Wolfram *Mathematica*® Tutorial Collection

J/LINK™ USER GUIDE

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Content authored by:
Todd Gayley

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Contents

Introduction to *J/Link*

Welcome to *J/Link*, a product that integrates *Mathematica* and Java. *J/Link* lets you call Java from *Mathematica* in a completely transparent way, and it also lets you use and control the *Mathematica* kernel from a Java program. For *Mathematica* users, *J/Link* makes the whole universe of existing and future Java classes an automatic extension to the *Mathematica* environment. For Java programmers, *J/Link* turns *Mathematica* into a scripting shell that lets you experiment with, build, and test Java classes a line at a time. It also makes Java the ideal language for writing programs that use the computational services of *Mathematica*.

J/Link's most unique feature is that it lets you load arbitrary Java classes into *Mathematica* and then create Java objects, call methods, and access fields directly from the *Mathematica* language. Thus, you can use *Mathematica* to "script" the functionality of an arbitrary Java program—in effect, writing a Java program in *Mathematica*. Essentially anything you can do from Java you can now do from *Mathematica*, perhaps even more easily because you are working in a true interpreted environment.

For example, you can now create a Java-based user interface entirely with *Mathematica* code. This could be anything from a simple progress bar for a long computation to a dialog box or sophisticated wizard that walks users through a calculation. Such an interface is completely portable and can make full use of AWT, Swing, or any other user-interface class library.

- Call Java methods from *Mathematica*
- Write Java programs that use *Mathematica* services
- Create alternative front ends for *Mathematica*
- Create dialog boxes and other popup user interface elements for *Mathematica* programs
- Write applets that use *Mathematica* kernels on the client or server
- Write servlets that make *Mathematica* services available to HTTP clients

Some uses for *J/Link.*

Java is a fast, robust, and portable general-purpose programming language. It is not just an "internet" language, although it does have many useful internet features. Java is also emphati-

cally *not* just a language for writing applets. Applets are a powerful use for Java, but Java is good for much more than that. In fact, applets have already been relegated to a relatively minor category of Java programs. Today, Java is everywhere—on the client, server, browser, database, device, and desktop. And *J/Link* lets you put *Mathematica* and Java together in any way you want.

J/Link is designed for end-users and developers alike. The same features that let *Mathematica* users transparently call any Java method also let developers create sophisticated commercial add-ons to *Mathematica*. Programmers who want to write custom front ends for *Mathematica*, or use *Mathematica* as a computational engine for another program, will find using Java with *J/Link* is easier than using the traditional *MathLink* interface from C or C++.

Finally, *J/Link* comes with full source code. This includes the components written in *Mathematica*, Java, and C. You can examine the code to supplement the documentation, get tips for your own programs, better understand how to use advanced features, or just see how it works.

Some familiarity with both Java and *Mathematica* is assumed in this manual. Even if you do not know Java, *J/Link* is easy to use as a means to call existing Java classes from *Mathematica*. This only requires learning what classes and methods are available—the syntax and intricacies of the Java language are irrelevant, since you will be writing *Mathematica* programs, not Java programs.

J/Link and *MathLink*

The underlying glue that makes this all work is *MathLink*, Wolfram Research's protocol for sending data and commands back and forth between *Mathematica* and other programs. At its core, *J/Link* is a *MathLink* developer's kit for Java, although it goes far beyond this. In fact, *J/Link*'s best feature is that for a large class of uses, it hides *MathLink* completely, so users and programmers do not need to know anything about it. This class corresponds to the so-called "installable" or "template" *MathLink* programs, which plug into *Mathematica* and extend its functionality. For all types of *MathLink* programs, *J/Link* provides a higher-level layer of functionality than the traditional C *MathLink* programming interface. This makes Java the easiest and most convenient language for writing programs to interact with *Mathematica*.

Calling Java from *Mathematica*

Preamble

J/Link provides *Mathematica* users with the ability to interact with arbitrary Java classes directly from *Mathematica*. You can create objects and call methods directly in the *Mathematica* language. You do not need to write any Java code, or prepare in any way the Java classes you want to use. You also do not need to know anything about *MathLink*. In effect, all of Java becomes a transparent extension to *Mathematica*, almost as if every existing and future Java class were written in the *Mathematica* language itself.

This facility is called "installable Java" because it generalizes the ability that *Mathematica* has always had to plug in extensions written in other languages through the `Install` function. You will see later how *J/Link* vastly simplifies this procedure for Java compared to languages like C or C++. In fact, *J/Link* makes the procedure go away completely, which is why Java becomes a transparent extension to *Mathematica*.

Although Java is often referred to as an interpreted language, this is really a misnomer. To use Java you must write a complete program, compile it, and then execute it (some environments exist that let you interactively execute lines of Java code, but these are special tools, and similar tools exist for traditional languages like C). *Mathematica* users have the luxury of working in a true interpreted, interactive environment that lets them experiment with functions and build and test programs a line at a time. *J/Link* brings this same productive environment to Java programmers. You could say that *Mathematica* becomes a scripting language for Java.

To *Mathematica* users, then, the "installable Java" feature of *J/Link* opens up the expanding universe of Java classes as an extension to *Mathematica*; for Java users, it allows the extraordinarily powerful and versatile *Mathematica* environment to be used as a shell for interactively developing, experimenting with, and testing Java classes.

Loading the *J/Link* Package

The first step is to load the *J/Link* package file.

```
Needs["JLink`"]
```

Launching the Java Runtime

InstallJava

The next step is to launch the Java runtime and "install" it into *Mathematica*. The function for this is `InstallJava`.

`InstallJava[]`	launch the Java runtime and prepare it for use from *Mathematica*
`ReinstallJava[]`	quit and restart the Java runtime if it is already running
`JavaLink[]`	give the `LinkObject` that is being used to communicate with the Java runtime

Launching the Java runtime.

```
In[1]:= InstallJava[]

Out[1]= LinkObject[d:\jdk122\bin\java, 5, 2]
```

`InstallJava` can be called more than once in a session. On every call after the first, it does nothing. Thus, it is safe to call `InstallJava` in any program you write, without considering whether the user has already called it.

`InstallJava` creates a command line that is used to launch the Java runtime (typically called "java") and specify some initial arguments for it. In rare cases you will need to control what is on this command line, so `InstallJava` takes a number of options for this purpose. Most users will not need to use these options, and in fact you should avoid them. Programmers should not assume that they have the ability to control the launch of the Java runtime, as it might already be running. If for some reason you absolutely must apply options to control the launch of the Java runtime, use `ReinstallJava` instead of `InstallJava`.

`ClassPath->None`	use the default class path of your Java runtime
`ClassPath->"dirs"`	use the specified directories and jar files
`CommandLine->"cmd"`	use the specified command line to launch the Java runtime, instead of "java"

Options for `InstallJava`.

Controlling the Command Used to Launch Java

An important option to InstallJava and ReinstallJava is CommandLine. This specifies the first part of the command line used to launch Java. One use for this option is if you have more than one Java runtime installed on your system, and you want to invoke a specific one:

```
ReinstallJava[CommandLine → "d:\\full\\path\\to\\java.exe"]
```

By default, InstallJava will launch the Java runtime that is bundled with *Mathematica* 4.2 and later. If you have an earlier version of *Mathematica,* the default command line that will be used is java on most systems. If the java executable is not on your system path, you can use InstallJava to point at it. Another use for this option is to specify arguments to Java that are not covered by other options. Here is an example that specifies verbose garbage collection and defines a property named foo to have the value bar.

```
ReinstallJava[CommandLine → "/path/to/java -verbosegc -Dfoo=bar"]
```

Overriding the Class Path

The class path is the set of directories in which the Java runtime looks for classes. When you launch a Java program from your system's command line, the class path used by Java includes some default locations and any locations specified in the CLASSPATH environment variable, if it exists. If you use the –classpath command-line option to specify a set of locations, however, then the CLASSPATH environment variable is ignored. The ClassPath option to InstallJava and ReinstallJava works the same way. If you leave it at the default value, Automatic, then *J/Link* will include the contents of the CLASSPATH environment variable in its class search path. If you set it to None or a string, then the contents of CLASSPATH are not used. If you set it to be a string, use the same syntax that you would use for setting the CLASSPATH environment variable, which is different for Windows and Unix:

```
ReinstallJava[ClassPath → "c:\\my\\java\\dir;d:\\MyJavaStuff.jar"] (* Windows *)

ReinstallJava[ClassPath → "/my/java/dir:/home/me/MyJavaStuff.jar"]
(* Unix/Linux *)
```

J/Link has its own mechanism for controlling the class search path that is very flexible. Not only does *J/Link* automatically search for classes in *Mathematica* application directories, it also lets you dynamically add new search locations while the Java runtime is running. This means that

using the ClassPath option to configure the class path when Java first launches is not very important. One setting for the ClassPath option that is sometimes useful is None, to prevent *J/Link* from finding any classes from the contents of CLASSPATH. You might want to do this if you had an experimental version of some class in a development directory and you wanted to make sure that *J/Link* used that version in preference to an older one that was present on your CLASSPATH. "The Java Class Path" presents a complete treatment of the subject of how *J/Link* searches for classes, and how to add locations to this search path.

Loading Classes

LoadJavaClass

LoadJavaClass["classname"]	load the specified class into Java and *Mathematica*
LoadClass["classname"]	deprecated name from earlier versions of *J/Link*; use LoadJavaClass instead

Loading classes.

To use a Java class in *Mathematica*, it must first be loaded into the Java runtime and certain definitions must be set up in *Mathematica*. This is accomplished with the LoadJavaClass function. LoadJavaClass takes a string specifying the fully qualified name of the class (i.e., the full hierarchical name with all the periods):

```
In[1]:= urlClass = LoadJavaClass["java.net.URL"]
Out[1]= JavaClass[java.net.URL]
```

The return value is an expression with head JavaClass. This JavaClass expression can be used in many places in *J/Link*, so you might want to assign it to a variable as done here. Virtually everywhere in *J/Link* where a class needs to be specified as an argument, you can use either a JavaClass expression, the fully qualified class name as a string, or an object of the class. Note that you cannot create a valid JavaClass expression by simply typing it in—it must be returned by LoadJavaClass.

When a class has been loaded, you can call static methods in the class, create objects of the class, and invoke methods and access fields of these objects. You can use any *public* constructors, methods, or fields of a class.

`StaticsVisible->True`	make static methods and fields accessible by just their names, not in a special context
`AllowShortContext->False`	make static methods and fields accessible only in their fully qualified class context
`UseTypeChecking->False`	suppress the type checking that is normally inserted in definitions for calls into Java

Options for `LoadJavaClass`.

"The Java Class Path" discusses the details of how and where *J/Link* finds classes. *J/Link* will be able to find classes on the class path, in the special Java extensions directory, and in a set of extra directories that users can control even while *J/Link* is running.

When to Call LoadJavaClass

It is often the case that you do not need to explicitly load a class with `LoadJavaClass`. As described later, when you create a Java object with `JavaNew`, you can supply the class name as a string. If the class has not already been loaded, `LoadJavaClass` will be called internally by `JavaNew`. In fact, anytime a Java object is returned to *Mathematica* its class is loaded automatically if necessary. This would seem to imply that there is little reason to use `LoadJavaClass`. There are a number of reasons why you would want or need to use `LoadJavaClass` explicitly:

- You need to call a static method of a class and you will not create, or have not yet created, an object of that class. A class must be loaded before any of its static methods can be called.
- You need to use one of the options to `LoadJavaClass`. When `LoadJavaClass` is called internally by `JavaNew`, it is called with the default option settings.
- You want to see errors associated with loading a class reported at a well-defined time.
- You want to control where your users experience the initial delay associated with loading a class. Loading a class can take several seconds if it or one of its parent classes is very large (although it rarely takes that long). You might want to avoid a mysterious delay in a function that users expect to be very quick.
- You want to hang on to the `JavaClass` expression returned by `LoadJavaClass` to use it in other functions. Although all functions that take a `JavaClass` can also take a class name string, you might prefer to use a named `JavaClass` variable for readability purposes. It is also slightly faster than using a string, but this will not be perceptible unless you are using it many times in a loop.
- You feel that it makes your code more self-documenting.

The operation of loading a class in *J/Link* is only done once in a *J/Link* session (a session is the period between `InstallJava` and `UninstallJava`). You can call `LoadJavaClass` on a given class as many times as you want, and every call after the first one immediately returns the `JavaClass` expression without doing any work. This is important, as it means that you never have to worry whether a class has been loaded already—if you are not sure, call `LoadJavaClass`.

Developers writing code for a wide audience should always call `LoadJavaClass` on any classes they need in every function that needs them. It is not suitable to call `LoadJavaClass` in the body of your package code when it is read in, as the user may quit and restart the Java runtime (i.e., `UninstallJava` and `InstallJava`) after your package was read. To be safe, every user-level function that uses *J/Link* should call `InstallJava` and `LoadJavaClass` (if `LoadJavaClass` is necessary; see the following). Both calls execute very quickly if they are not needed.

As mentioned already, loading a class can take several seconds in some cases. When a class is loaded, all of its superclasses are loaded in succession, walking up the inheritance hierarchy. Because a given class is only actually loaded once, if you load another class that shares some of the same superclasses as a previously loaded class, these superclasses will not have to be loaded again. This means that loading the second class will be much quicker than the first if any of the shared superclasses were large. An example of this is loading classes in the java.awt package. The class `java.awt.Component` is very large, so the first time you load a class that inherits from it, say `java.awt.Button`, there will be a noticeable delay. Subsequent loading of other classes derived from `Component` will be much quicker.

Contexts and Visibility of Static Members

LoadJavaClass has two options that let you control the naming and visibility of static methods and fields. To understand these options, you need to understand the problems they help to solve. This explanation gets a bit ahead since how to call Java methods has not been discussed. When a class is loaded, definitions are created in *Mathematica* that allow you to call methods and access fields of objects of that class. Static members are treated quite differently from nonstatic ones. None of these issues arise for nonstatic members, so only static members are discussed in this section. Say you have a class named com.foobar.MyClass that contains a static method named foo. When you load this class, a definition must be set up for foo so that it can be called by name, something like foo[*args*]. The question becomes: In what context do you want the symbol foo defined, and do you want this context to be visible (i.e., on $ContextPath)?

J/Link always creates a definition for foo in a context that mirrors its fully qualified classname: com`foobar`MyClass`foo. This is done to avoid conflicting with symbols named foo that might be present in other contexts. However, you might find it clumsy to have to call foo by typing the full context name every time, as in com`foobar`MyClass`foo[*args*]. The option AllowShortContext -> True (this is the default setting) causes *J/Link* to also make definitions for foo accessible in a shortened context, one that consists of just the class name without the hierarchical package name prefix. In the example, this means that you could call foo as simply MyClass`foo[*args*]. If you need to avoid use of the short context because there is already a context of the same name in your *Mathematica* session, you can use AllowShortContext -> False. This forces all names to be put only in the "deep" context. Note that even with AllowShortContext -> True, names for statics are also put into the deep context, so you can always use the deep context to refer to a symbol if you desire.

AllowShortContext, then, lets you control the context where the symbol names are defined. The other option, StaticsVisible, controls whether this context is made visible (put on $ContextPath) or not. The default is StaticsVisible -> False, so you have to use a context name when referring to a symbol, as in MyClass`foo[*args*]. With StaticsVisible -> True, MyClass` will be put on $ContextPath, so you could just write foo[*args*]. Having the default be True would be a bit dangerous—every time you load a class a potentially large number of

names would suddenly be created and made visible in your *Mathematica* session, opening up the possibility for all sorts of "shadowing" problems if symbols of the same names were already present. This problem is particularly acute with Java, because method and field names in Java typically begin with a lowercase letter, which is also the convention for user-defined symbols in *Mathematica*. Some Java classes define static methods and fields with names like `x`, `y`, `width`, and so on, so shadowing errors are very likely to occur (see "Contexts" for a discussion of contexts and shadowing problems).

For these reasons `StaticsVisible -> True` is recommended only for classes that you have written, or ones whose contents you are familiar with. In such cases, it can save you some typing, make your code more readable, and prevent the all-too-easy bug of forgetting to type the package prefix. A classic example would be implementing the venerable "addtwo" *MathLink* example program. In Java, it might look like this:

```
public class AddTwo {
    public static int addtwo(int i, int j) {return i + j;}
}
```

With the default `StaticsVisible -> False`, you would have to call addtwo as `AddTwo`addtwo[3, 4]`. Setting `StaticsVisible -> True` lets you write the more obvious `addtwo[3, 4]`.

Be reminded that these options are only for *static* methods and fields. As discussed later, non-statics are handled in a way that makes context and visibility issues go away completely.

Inner Classes

Inner classes are public classes defined inside another public class. For example, the class `javax.swing.Box` has an inner class named `Filler`. When you refer to the `Filler` class in a Java program, you typically use the outer class name, followed by a period, then the inner class name:

```
Box.Filler f = new Box.Filler(…);
```

You can use inner classes with *J/Link*, but you need to use the true internal name of the class, which has a `$`, not a period, separating the outer and inner class names:

```
filler = JavaNew["java.swing.Box$Filler", …]
```

If you look at the class files produced by the Java compiler, you will see these `$`-separated class names for inner classes.

Conversion of Types Between Java and *Mathematica*

Before you encounter the operations of creating Java objects and calling methods, you should examine the mapping of types between *Mathematica* and Java. When a Java method returns a result to *Mathematica*, the result is automatically converted into a *Mathematica* expression. For example, Java integer types (e.g., byte, short, int, and so on), are converted into *Mathematica* integers, and Java real number types (float, double) are converted into *Mathematica* reals. The following table shows the complete set of conversions. These conversions work both ways—for example, when a *Mathematica* integer is sent to a Java method that requires a byte value, the integer is automatically converted to a Java byte.

Java type	*Mathematica type*
`byte, char, short, int, long`	`Integer`
`Byte, Character, Short, Integer, Long, BigInteger`	`Integer`
`float, double`	`Real`
`Float, Double, BigDecimal`	`Real`
`boolean`	`True or False`
`String`	`String`
`array`	`List`
controlled by user (see "Complex Numbers")	`Complex`
`Object`	`JavaObject`
`Expr`	any expression
`null`	`Null`

Corresponding types in Java and *Mathematica*.

Java arrays are mapped to *Mathematica* lists of the appropriate depth. Thus, when you call a method that takes a `double[]`, you might pass it `{1.0, 2.0, N[Pi], 1.23}`. Similarly, a method that returns a two-deep array of integers (i.e., `int[][]`) might return to *Mathematica* the expression `{{1, 2, 3}, {5, 3, 1}}`.

In most cases, *J/Link* will let you supply a *Mathematica* integer to a method that is typed to take a real type (`float` or `double`). Similarly, a method that takes a `double[]` could be

passed a list of mixed integers and reals. The only times when you cannot do this are the rare cases where a method has two signatures that differ only in a real versus integer type at the same argument slot. For example, consider a class with these methods:

```
public void foo(byte b, Object obj);
public void foo(float f, Object obj);
public void bar(float f, Object obj);
```

J/Link would create two *Mathematica* definitions for the method `foo`—one that required an integer for the first argument and invoked the first signature, and one that required a real number for the first argument and invoked the second signature. The definition created for the method `bar` would accept an integer or a real for the first argument. In other words, *J/Link* will automatically convert integers to reals, except in cases where such conversion makes it ambiguous as to which signature of a given method to invoke. This is not strictly true, though, as *J/Link* does not try as hard as it possibly could to determine whether real versus integer ambiguity is a problem at every argument position. The presence of ambiguity at one position will cause *J/Link* to give up and require exact type matching at all argument positions. This is starting to sound confusing, but you will find that in most cases *J/Link* allows you to pass integers or lists with integers to methods that take reals or arrays of reals, respectively, as arguments. In cases where it does not, the call will fail with an error message, and you will have to use *Mathematica*'s `N` function to convert all integers to reals explicitly.

Creating Objects

To instantiate Java objects, use the `JavaNew` function. The first argument to `JavaNew` is the object's class, specified either as a `JavaClass` expression returned from `LoadJavaClass` or as a string giving the fully qualified class name (i.e., having the full package prefix with all the periods). If you wish to supply any arguments to the object's constructor, they follow as a sequence after the class.

`JavaNew[`*cls*`,`*arg1*`,…]`	construct a new object of the specified class and return it to *Mathematica*
`JavaNew["`*classname*`",`*arg1*`,…]`	construct a new object of the specified class and return it to *Mathematica*

Constructing Java objects.

For example, this will create a new `Frame`.

```
frm = JavaNew["java.awt.Frame"]
```

```
«JavaObject[java.awt.Frame] »
```

The return value from `JavaNew` is a strange expression that looks like it has the head `JavaObject`, except that it is enclosed in angle brackets. The angle brackets are used to indicate that the form in which the expression is displayed is quite different from its internal representation. These expressions will be referred to as `JavaObject` expressions. `JavaObject` expressions are displayed in a way that shows their class name, but you should consider them opaque, meaning that you cannot pick them apart or peer into their insides. You can only use them in *J/Link* functions that take `JavaObject` expressions. For example, if *obj* is a `JavaObject`, you cannot use `First[`*obj*`]` to get its class name. Instead, there is a *J/Link* function, `ClassName[`*obj*`]`, for this purpose.

`JavaNew` invokes a Java constructor appropriate for the types of the arguments being passed in, and then returns to *Mathematica* what is, in effect, a reference to the object. That is how you should think of `JavaObject` expressions—as references to Java objects very much like object references in the Java language itself. What is returned to *Mathematica* is not large no matter what type of object you are constructing. In particular, the object's data (that is, its fields) are not sent back to *Mathematica*. The actual object remains on the Java side, and *Mathematica* gets a reference to it.

The `Frame` class has a second constructor, which takes a title in the form of a string. Here is how you would call that constructor.

```
frm = JavaNew["java.awt.Frame", "My Example Frame"]
```

```
«JavaObject[java.awt.Frame] »
```

Note that simply constructing a `Frame` does not cause it to appear. That requires a separate step (calling the frame's `show` or `setVisible` methods will work, but as you will see later, *J/Link* provides a special function, `JavaShow`, to make Java windows appear and come to the foreground).

The previous examples specified the class by giving its name as a string. You can also use a `JavaClass` expression, which is a special expression returned by `LoadJavaClass` that identifies a class in a particularly efficient manner. When you specify the class name as a string, the class is loaded if it has not already been.

```
frameClass = LoadJavaClass["java.awt.Frame"];
frm = JavaNew[frameClass, "My Example Frame"];
```

JavaNew is not the only way to get a reference to a Java object in *Mathematica*. Many methods and fields return objects, and when you call such a method, a JavaObject expression is created. Such objects can be used in the same way as ones you explicitly construct with JavaNew.

At this point, you may be wondering about things like reference counts and how objects returned to *Mathematica* get cleaned up. These issues are discussed in "Object References in *Mathematica*".

J/Link has two other functions for creating Java objects, called MakeJavaObject and MakeJavaExpr. These specialized functions are described in the section "MakeJavaObject and MakeJavaExpr".

Calling Methods and Accessing Fields

Syntax

The *Mathematica* syntax for calling Java methods and accessing fields is very similar to Java syntax. The following box compares the *Mathematica* and Java ways of calling constructors, methods, fields, static methods, and static fields. You can see that *Mathematica* programs that use Java are written in almost exactly the same way as Java programs, except *Mathematica* uses [] instead of () for arguments, and *Mathematica* uses @ instead of Java's . (dot) as the "member access" operator.

An exception is that for static methods, *Mathematica* uses the context mark ` in place of Java's dot. This parallels Java usage also, as Java's use of the dot in this circumstance is really as a scope resolution operator (like :: in C++). Although *Mathematica* does not use this terminology, its scope resolution operator is the context mark. Java's hierarchical package names map directly to *Mathematica*'s hierarchical contexts.

	constructors
Java:	`MyClass obj=new MyClass (args);`
Mathematica:	`obj=JavaNew["MyClass",args];`
	methods
Java:	`obj.methodName (args);`
Mathematica:	`obj@methodName[args]`
	fields
Java:	`obj.fieldName=1;` `value=obj.fieldName;`
Mathematica:	`obj@fieldName=1;` `value=obj@fieldName;`
	static methods
Java:	`MyClass.staticMethod (args);`
Mathematica:	`MyClass`staticMethod[args];`
	static fields
Java:	`MyClass.staticField=1;` `value=MyClass.staticField;`
Mathematica:	`MyClass`staticField=1;` `value=MyClass`staticField;`

Java and *Mathematica* syntax comparison.

You may already be familiar with `@` as a *Mathematica* operator for applying a function to an argument: `f@x` is equivalent to the more commonly used `f[x]`. *J/Link* does not usurp `@` for some special operation—it is really just normal function application slightly disguised. This means that you do not have to use `@` at all. The following are equivalent ways of invoking a method:

```
(* These are equivalent *)
obj@method[args];
obj[method[args]];
```

The first form preserves the natural mapping of Java's syntax to *Mathematica*'s, and it will be used exclusively in this tutorial.

When you call methods or fields and get results back, *J/Link* automatically converts arguments and results to and from their *Mathematica* representations according to the table in "Conversion of Types between Java and *Mathematica*".

Method calls can be chained in *Mathematica* just like in Java. For example, if `meth1` returns a Java object, you could write in Java `obj.meth1().meth2()`. In *Mathematica*, this becomes `obj@meth1[]@meth2[]`. Note that there is an apparent problem here: *Mathematica*'s `@` operator groups to the right, whereas Java's dot groups to the left. In other words, `obj.meth1().meth2()` in Java is really `(obj.meth1()).meth2()` whereas `obj@meth1[]@meth2[]` in *Mathematica* would normally be `obj@(meth1[]@meth2[])`. I say "normally" because *J/Link* automatically causes chained calls to group to the left like Java. It does this by defining rules for `JavaObject` expressions, not by altering the properties of the `@` operator, so the global behavior of `@` is not affected. This chaining behavior only applies to method calls, not fields. You cannot do this:

```
(* These are incorrect. You cannot chain calls after a field access. *)
x = obj@field@method[args];
x = obj@field1@field2;
```

You would have to split these up into two lines. For example, the second line above would become:

```
temp = obj@field1;
x = temp@field2;
```

In Java, like other object-oriented languages, method and field names are scoped by the object on which they are called. In other words, when you write `obj.meth()`, Java knows that you are calling the method named `meth` that resides in `obj`'s class, even though there may be other methods named `meth` in other classes. *J/Link* preserves this scoping for *Mathematica* symbols so that there is never a conflict with existing symbols of the same name. When you write `obj@meth[]`, there is no conflict with any other symbols named `meth` in the system—the symbol `meth` used by *Mathematica* in the evaluation of this call is the one set up by *J/Link* for this class. Here is an example using a field. First, you create a `Point` object.

```
pt = JavaNew["java.awt.Point"]
```

```
«JavaObject[java.awt.Point] »
```

The `Point` class has fields named `x` and `y`, which hold its coordinates. A user's session is also likely to have symbols named `x` or `y` in it, however. You set up a definition for `x` that will tell you when it is evaluated.

```
x := Print["gotcha"]
```

Now set a value for the field named `x` (this would be written as `pt.x = 42` in Java).

```
pt@x = 42;
```

You will notice that "gotcha" was not printed. There is no conflict between the symbol `x` in the `Global`` context that has the `Print` definition and the symbol `x` that is used during the evaluation of this line of code. *J/Link* protects the names of methods and fields on the right-hand side of `@` so that they do not conflict with, or rely on, any definitions that might exist for these symbols in visible contexts. Here is a method example that demonstrates this issue differently.

```
frm = JavaNew["java.awt.Frame"];
frm@show[]
```

Even though a new symbol `show` is being created here, the `show` that is used by *J/Link* is the one that resides down in the `java`awt`Frame` context, which has the necessary definitions set up for it.

In summary, for nonstatic methods and fields, you never have to worry about name conflicts and shadowing, no matter what context you are in or what the `$ContextPath` is at the moment. This is not true for static members, however. Static methods and fields are called by their full name, without an object reference, so there is no object out front to scope the name. Here is a simple example of a static method call that invokes the Java garbage collector. You need to call `LoadJavaClass` before you call a static method to make sure the class has been loaded.

```
LoadJavaClass["java.lang.Runtime"];
Runtime`gc[];
```

The name scoping issue is not usually a problem with statics, because they are defined in their own contexts (`Runtime`` in this example). These contexts are usually not on `$ContextPath`, so you do not have to worry that there is a symbol of the same name in the `Global`` context or in a package that has been read. There is more discussion of this issue in the section on `LoadJavaClass`, because `LoadJavaClass` takes options that determine the contexts in which static methods are defined and whether or not they are put on `$ContextPath`. If there is already a context named `Runtime`` in your session, and it has its own symbol `gc`, you can always avoid a conflict by using the fully hierarchical context name that corresponds to the full class name for a static member.

```
java`lang`Runtime`gc[];
```

Finally, just as in Java, you can call a static method on an object if you like. In this case, since there is an object out front, you get the name scoping. Here you call a static method of the `Runtime` class that returns the current `Runtime` object (you cannot create a `Runtime` object with `JavaNew`, as `Runtime` has no constructors). You then invoke the (static) method `gc` on the object, and you can use `gc` without any context prefix.

```
runtime = Runtime`getRuntime[];
runtime@gc[];
```

Underscores in Java Names

Java names can have characters in them that are not legal in *Mathematica* symbols. The only common one is the underscore. *J/Link* maps underscores in class, method, and field names to "U". Note that this mapping is only used where it is necessary—when names are used in symbolic form, not as strings. For example, assume you have a class named `com.acme.My_Class`. When you refer to this class name as a string, you use the underscore.

```
LoadJavaClass["com.acme.My_Class"];
JavaNew["com.acme.My_Class"];
```

But when you call a static method in such a class, the hierarchical context name is symbolic, so you must convert the underscore to `U`.

```
com`acme`MyUClass`staticMethod[];
MyUClass`staticMethod[];
```

The same rule applies to method and field names. Many Java field names have underscores in them, for example `java.awt.Frame.TOP_ALIGNMENT`. To refer to this method in code, use the `U`.

```
LoadJavaClass["java.awt.Frame"];
Frame`TOPUALIGNMENT
0.
```

In cases where you supply a string, leave the underscore.

```
Fields["java.awt.Frame", "*_ALIGNMENT"]
static final float BOTTOM_ALIGNMENT
static final float CENTER_ALIGNMENT
static final float LEFT_ALIGNMENT
static final float RIGHT_ALIGNMENT
static final float TOP_ALIGNMENT
```

Getting Information about Classes and Objects

J/Link has some useful functions that show you the constructors, methods, and fields available for a given class or object.

`Constructors[`*cls*`]`	return a table of the public constructors and their arguments
`Constructors[`*obj*`]`	constructors for this object's class
`Methods[`*cls*`]`	return a table of the public methods and their arguments
`Methods[`*cls*`,"`*pat*`"]`	show only methods whose names match the string pattern *pat*
`Methods[`*obj*`]`	show methods for this object's class
`Fields[`*cls*`]`	return a table of the public fields
`Fields[`*cls*`,"`*pat*`"]`	show only fields whose names match the string pattern *pat*
`Fields[`*obj*`]`	show fields for this object's class
`ClassName[`*cls*`]`	return, as a string, the name of the class represented by *cls*
`ClassName[`*obj*`]`	return, as a string, the name of this object's class
`GetClass[`*obj*`]`	return the `JavaClass` representing this object's class
`ParentClass[`*obj*`]`	return the `JavaClass` representing this object's parent class
`InstanceOf[`*obj*`,`*cls*`]`	return `True` if this object is an instance of *cls*, `False` otherwise
`JavaObjectQ[`*expr*`]`	return `True` if *expr* is a valid reference to a Java object, `False` otherwise

Getting information about classes and objects.

You can give an object or a class to `Constructors`, `Methods`, and `Fields`. The class can be specified either by its full name as a string, or as a `JavaClass` expression:

```
urlClass = LoadJavaClass["java.net.URL"];
urlObject = JavaNew["java.net.URL", "http://www.wolfram.com"];
(* The next three lines are equivalent *)
Methods[urlClass]
Methods[urlObject]
Methods["java.net.URL"]
```

The declarations returned by these functions have been simplified by removing the Java keywords `public`, `final` (removed only for methods, not fields), `synchronized`, `native`, `volatile`, and `transient`. The declarations will always be public, and the other modifiers are probably not relevant for use via *J/Link*.

Methods and Fields take one option, Inherited, which specifies whether to include members inherited from superclasses and interfaces or show only members declared in the class itself. The default is Inherited -> True.

Inherited->False	show only members that are declared in the class itself, not inherited from superclasses or interfaces

Option for Methods and Fields.

There are additional functions that give information about objects and classes. These functions are ClassName, GetClass, ParentClass, InstanceOf, and JavaObjectQ. They are self-explanatory, for the most part. The InstanceOf function mimics the Java language's instanceof operator. JavaObjectQ is useful for writing patterns that match only valid Java objects:

```
Stringify[obj_?JavaObjectQ] := obj[toString[]]
```

JavaObjectQ returns True if and only if its argument is a valid reference to a Java object or if it is the symbol Null, which maps to Java's null object.

Quitting or Restarting Java

When you are finished with using Java in a *Mathematica* session, you can quit the Java runtime by calling UninstallJava[].

UninstallJava[]	quit the Java runtime
ReinstallJava[]	restart the Java runtime

Quitting the Java runtime.

In addition to quitting Java, UninstallJava clears out the many symbols and definitions created in *Mathematica* when you load classes. All outstanding JavaObject expressions will become invalid when Java is quit. They will no longer satisfy JavaObjectQ, and they will show up as raw symbols like JLink`Objects`JavaObject12345678 instead of << JavaObject[classname] >>.

Most users will have no reason to call UninstallJava. You should think of the Java runtime as an integral part of the *Mathematica* system—start it up, and then just leave it running. All code that uses *J/Link* shares the same Java runtime, and there may be packages that you are using

that make use of Java without you even knowing it. Shutting down Java might compromise their functionality. Developers writing packages should *never* call `UninstallJava` in their packages. You cannot assume that when your application is done with *J/Link*, your users are done with it as well.

About the only common reason to need to stop and restart Java is when you are actively developing Java classes that you want to call from *Mathematica*. Once a class is loaded into the Java runtime, it cannot be unloaded. If you want to modify and recompile your class, you need to restart Java to reload the modified version. Even in this circumstance, though, you will not be calling `UninstallJava`. Instead, you will call `ReinstallJava`, which simply calls `UninstallJava` followed by `InstallJava` again.

Version Information

J/Link provides three symbols that supply version information. These symbols provide the same type of information as their counterparts in *Mathematica* itself, except that they are in the `JLink`Information`` context, which is not on `$ContextPath`, so you must specify them by their full names.

`JLink`Information`$Version`	a string giving full version information
`JLink`Information`$VersionNumber`	a real number giving the current version number
`JLink`Information`$ReleaseNumber`	an integer giving the release number (the last digit in a full x.x.x version specification)
`ShowJavaConsole[]`	the console window will show version information for the Java runtime and the *J/Link* Java component

J/Link version information.

```
JLink`Information`$Version
```

```
J/Link Version 4.0.1
```

```
JLink`Information`$VersionNumber
```

```
4.
```

```
JLink`Information`$ReleaseNumber
```

```
1
```

The ShowJavaConsole[] function, described in "The Java Console Window", will also display some useful version information. It shows the version of the Java runtime being used and the version of the portion of *J/Link* that is written in Java. The version of the *J/Link* Java component should match the version of the *J/Link Mathematica* component.

Controlling the Class Path: How *J/Link* Finds Classes

The Java Class Path

The class path tells the Java runtime, compiler, and other tools where to find third-party and user-defined classes—classes that are not Java "extensions" or part of the Java platform itself. The class path has always been a source of confusion among Java users and programmers.

Java can find classes that are part of the standard Java platform (so-called "bootstrap" classes), classes that use the so-called "extensions" mechanism, and classes on the class path, which is controlled by the CLASSPATH environment variable or by command-line options when Java is launched. *J/Link* can load and use any classes that the Java runtime can find through these normal mechanisms. In addition, *J/Link* can find classes, resources, and native libraries that are in a set of extra locations, beyond what is specified on the class path at startup. This set of extra locations can be added to while Java is running.

J/Link provides two ways to alter the search path Java uses to find classes. The first way is via the ClassPath option to ReinstallJava. The second way, which is superior to modifying the class path at startup, is to add new directories and jar files to the special set of extra locations that *J/Link* searches. These two methods will be described in the next two subsections.

Overriding the Startup Class Path

For a class to be accessible via the standard Java class path, one of the following must apply:

- It is inside a .zip or .jar file that is itself named on the class path.
- It is a loose class file that is in an appropriately nested directory beneath a directory that is on the class path.

"Appropriately nested" means that the class file must be in a directory whose hierarchy mirrors the full package name of the class. For example, assume that the directory c:\MyClasses is on the class path. If you have a class that is not in a package (there is no package statement at the beginning of the code), its class file should be put directly into c:\MyClasses. If you have a class that is in the package `com.acme.stuff`, its class file would need to be in the directory c:\MyClasses\com\acme\stuff. Note that jar and zip files must be explicitly named on the class path—you cannot just toss them into a directory that is itself named on the class path. Directory issues are not relevant for jar and zip files, meaning that regardless of how hierarchically organized the classes inside a jar file are, you simply name the jar file itself on the class path and all the classes inside it can be found.

If you want to specify paths for classes that are not part of the standard Java platform or extensions, you can use the `ClassPath` option to `ReinstallJava`. The value that you supply for the `ClassPath` option is a string that names the desired directories and zip or jar files. This string is platform-dependent; the paths are specified in the native style for your platform, and the separator character is a colon on Unix and a semicolon on Windows. Here are typical specifications:

```
ReinstallJava[ClassPath → "c:\\MyJavaDir\\MyPackage.jar;c:\\MyJavaDir"]
(* Windows *)

ReinstallJava[ClassPath → "~/MyJavaDir/MyPackage.jar:~/MyJavaDir"]
(* Unix *)
```

The default setting for `ClassPath` is `Automatic`, which means to use the value of the `CLASSPATH` environment variable. If you set `ClassPath` to something else, then J/Link will ignore the `CLASSPATH` environment variable—it will not be able to find those classes. In other words, if you use a `ClassPath` specification, you lose the `CLASSPATH` environment variable. This is similar to the behavior of the `-classpath` command-line option to the Java runtime and compiler, if you are familiar with those tools.

It is recommended that users avoid the `ClassPath` option. If you need the dynamic control that the `ClassPath` option provides, you should use the more powerful and convenient `AddToClassPath` mechanism, described in the next section. The most common reason for using the `ClassPath` option is if you want to specifically prevent the contents of the `CLASSPATH` environment variable from being used. To do this, set `ClassPath -> None`.

Dynamically Modifying the Class Path

One thing that is inconvenient about the standard Java class path is that it cannot be changed after the Java runtime has been launched. *J/Link* has its own class loader that searches in a set of special locations beyond the standard Java class path. This gives *J/Link* an extremely powerful and flexible means of finding classes. To add locations to this extra set, use the `AddToClassPath` function.

`AddToClassPath["`*location*`",…]`	add the specified directories or jar files to *J/Link*'s class search path

Adding classes to the search path.

After Java has been started, you can call `AddToClassPath` whenever you wish, and it will take effect immediately. One convenient feature of this extra class search path is that if you add a directory, then any jar or zip files in that directory will be searched. This means that you do not have to name jar files individually, as you need to do with the standard Java class path. For loose class files, the nesting rules are the same as for the class path, meaning that if a class is in the package `com.acme.stuff`, and you called `AddToClassPath["d:\\myClasses"]`, then you would need to put the class file into d:\MyClasses\com\acme\stuff.

Changes to the search path that you make with `AddToClassPath` only apply to the current Java session. If you quit and restart java, you will need to call `AddToClassPath` again.

In addition to the locations you add yourself with `AddToClassPath`, *J/Link* automatically includes any Java subdirectories of any directories in the standard *Mathematica* application locations ($UserBaseDirectory/AddOns/Applications, $BaseDirectory/AddOns/Applications, *< Mathematica dir >*/AddOns/Applications, and *< Mathematica dir >*/AddOns/ExtraPackages). This feature is designed to provide extremely easy deployment for developers who create applications for *Mathematica* that use Java and *J/Link* for part of their implementation. This is described in "Deploying Applications that use *J/Link*" in more detail, but even casual Java programmers who are writing classes to use with *J/Link* can take advantage of it. Just create a subdirectory of AddOns/Applications, say MyStuff, create a Java subdirectory within it, and toss class or jar files into it. *J/Link* will be able to find and use them. Of course, loose class files have to be placed into an appropriately nested subdirectory of the Java directory, corresponding to their package names (if any), as described.

The AddToClassPath function was introduced in *J/Link* 2.0. Previous versions of *J/Link* had a variable called $ExtraClassPath that specified a list of extra locations. You could add to this list like this:

```
AppendTo[$ExtraClassPath, "d:\\MyClasses"];
```

$ExtraClassPath was deprecated in *J/Link* 2.0, but it still works. One advantage of $ExtraClassPath over using AddToClassPath is that changes made to $ExtraClassPath persist across a restart of the Java runtime.

Examining the Class Path

The JavaClassPath function returns the set of directories and jar files in which *J/Link* will search for classes. This includes all locations added with AddToClassPath or $ExtraClassPath, as well as Java subdirectories of application directories in any of the standard *Mathematica* application locations. It does not display the jar files that make up the standard Java platform itself, or jar files in the Java extensions directory. Those classes can always be found by Java programs.

JavaClassPath[]	gives the complete set of directories and jar files in which *J/Link* will search for classes

Inspecting the class search path.

Using J/Link's Class Loader Directly

As stated earlier, *J/Link* uses its own class loader to allow it to find classes and other resources in a dynamic set of locations beyond the startup class path. Essentially all the classes that you load using *J/Link* that are not part of the Java platform itself will be loaded by this class loader. One consequence of this is that calling Java's Class.forName() method from *Mathematica* will often not work.

```
LoadJavaClass["java.lang.Class"];
cls = Class`forName["some.class.that.only.JLink.can.find"]
```

```
Java::excptn : A Java exception occurred: java.lang.ClassNotFoundException:
    some.class.that.only.JLink.can.find
      at java.net.URLClassLoader$1.run(Unknown Source)
      at java.security.AccessController.doPrivileged(Native Method)
      at java.net.URLClassLoader.findClass(Unknown Source)
      at java.lang.ClassLoader.loadClass(Unknown Source)
      at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
      at java.lang.ClassLoader.loadClass(Unknown Source)
      at java.lang.ClassLoader.loadClassInternal(Unknown Source)
      at java.lang.Class.forName0(Native Method)
      at java.lang.Class.forName(Unknown Source)
      at
    sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
      at
    sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source).
$Failed
```

The problem is that `Class.forName()` finds classes using a default class loader, not the *J/Link* class loader, and this default class loader does not know about the special directories in which *J/Link* looks for classes (in fact, it does not even know about the startup class path, because of details of how *J/Link* launches Java). If you are translating Java code into *Mathematica*, or if you just want to get a `Class` object for a given class, watch out for this problem. The fix is to force *J/Link*'s class loader to be used. One way to do this is to use the three-argument form of `Class.forName()`, which allows you to specify the class loader to be used:

```
LoadJavaClass["com.wolfram.jlink.JLinkClassLoader"];
cls = Class`forName["some.class.that.only.JLink.can.find",
  True, JLinkClassLoader`getInstance[]]
```

An easier way is to use the static `classFromName` method of `JLinkClassLoader`:

```
cls = JLinkClassLoader`classFromName["some.class.that.only.JLink.can.find"]
```

You should think of this `classFromName()` method as being the replacement for `Class.forName()`. When you find yourself wanting to obtain a `Class` object from a class name given as a string, remember to use `JLinkClassLoader.classFromName()`.

`Class.forName()` is not very commonly found in Java code. One reason it is used is when an object needs to be created, but its class was not known at compile time. For example, the class name might come from a preferences file or be determined programmatically in some other way. Often, the very next line creates an instance of the class, like this:

```
// Java code
Class cls = Class.forName("SomeClassThatImplementsInterfaceX");
X obj = (X) cls.newInstance();
```

If you are translating code like this into a *Mathematica* program, this operation can be performed simply by calling `JavaNew`:

```
obj = JavaNew["SomeClassThatImplementsInterfaceX"]
```

The point here is that for a very common usage of `Class.forName()`, you do not have to translate it line-by-line into *Mathematica*—you can duplicate the functionality by calling `JavaNew`.

Performance Issues

Overhead of Calls to Java

The speed of Java programs is highly dependent on the Java runtime. On certain types of programs, for example, ones that spend most of their time in a tight number-crunching loop, the speed of Java can approach that of compiled, optimized C.

Java is a good choice for computationally intensive programs. Your mileage may vary, but do not rule out Java for any type of program before you have done some simple speed testing. For less demanding programs, where every ounce of speed is not necessary, the simplicity of using *J/Link* instead of programming traditional *MathLink* "installable" programs with C makes Java an obvious choice.

The speed issues with *J/Link* are not, for the most part, the speed of Java execution. Rather, the bottleneck is the rate at which you can perform calls into Java, which is itself limited mainly by the speed of *MathLink* and the processing that must be done in *Mathematica* for each call into Java. The maximum rate of calls into Java is highly dependent on which operating system and which Java runtime you use. A fast Windows machine can perform more than 5000 Java method calls per second, and considerably more if they are static methods, which require less

preprocessing in *Mathematica*. On some operating systems the results will be less. You should keep in mind that there is a more or less fixed cost of a call into Java regardless of what the call does, and on slow machines this cost could be as much as .001 seconds. Many Java methods will execute in considerably less time than this, so the total time for the call is often dominated by the fixed turnaround time of a *J/Link* call, not the speed of Java itself.

For most uses, the overhead of a call into Java is not a concern, but if you have a loop that calls into Java 500,000 times, you will have a problem (unless your program takes so long that the *J/Link* cost is negligible, in which case you have an even bigger problem!). If your *Mathematica* program is structured in a way that requires a great many calls into Java, you may need to refactor it to do more on the Java side and thus reduce the number of times you need to cross the Java-*Mathematica* boundary. This will probably involve writing some Java code, which unfortunately defeats the *J/Link* premise of being able to use *Mathematica* to script the functionality of an arbitrary Java program. There are uses of Java that just cannot be feasibly scripted in this way, and for these you will need to write more of the functionality in Java and less in *Mathematica*.

Speeding Up Sending Large Arrays

You can send and receive arrays of most "primitive" Java types (e.g., `byte`, `short`, `int`, `float`, `double`) nearly as fast as in a C-language program. The set of types that can be passed quickly corresponds to the set of types for which the *MathLink* C API has single functions to put arrays. The Java types `long` (these are 64 bits), `boolean`, and `String` do not have fast *MathLink* functions, and so sending or receiving these types is much slower. Try to avoid using extremely large arrays of these types (say, more than 100,000 elements) if possible.

A setting that has a big effect on the speed of moving multidimensional arrays is the one used to control whether "ragged" arrays are allowed. As discussed in "Ragged Arrays", the default behavior of *J/Link* is to require that all arrays be fully rectangular. But Java does not require that arrays conform to this restriction, and if you want to send or receive ragged arrays, you can call `AllowRaggedArrays[True]` in your *Mathematica* session. This causes *J/Link* to switch to a much slower method for reading and writing arrays. Avoid using this setting unless you need it, and switch it off as soon as you no longer require it.

When you load a class with a method that takes, say, an `int[][]`, the definition in *Mathematica* that *J/Link* creates for calling this method uses a pattern test that requires its argument to be a two-dimensional array of integers. If the array is quite large, say on the order of 500 by 500, this test can take a significant amount of time, probably similar to the time it takes to actually transfer the array to Java. If you want to avoid the time taken by this testing of array arguments, you can set the variable `$RelaxedTypeChecking` to `True`. If you do this, you are on your own to ensure that the arrays you send are of the right type and dimensionality. If you pass a bad array, you will get a *MathLink* error, but this will not cause any problems for *J/Link* (other than that the call will return `$Failed`).

You probably do not want to leave `$RelaxedTypeChecking` set to `True` for a long time, and if you are writing code for others to use you certainly do not want to alter its value in their session. `$RelaxedTypeChecking` is intended to be used in a `Block` construct, where it is given the value of `True` for a short period:

```
Block[{$RelaxedTypeChecking = True}, obj[meth[someLargeArray]]]
```

`$RelaxedTypeChecking` only has an effect for arrays, which are the only types for which the pattern test that *J/Link* creates is expensive relative to the actual call into Java.

Another optimization to speed up *J/Link* programs is to use `ReturnAsJavaObject` to avoid unnecessary passing of large arrays or strings back and forth between *Mathematica* and Java. `ReturnAsJavaObject` is discussed in the section "ReturnAsJavaObject".

An Optimization Example

Next examine a simple example of steps you might take to improve the speed of a *J/Link* program. Java has a powerful `DecimalFormat` class you can use to format *Mathematica* numbers in a desired way for output to a file. Here you create a `DecimalFormat` object that will format numbers to exactly four decimal places.

```
fmt = JavaNew["java.text.DecimalFormat", "#.0000"];
```

To use the `fmt` object, you call its `format()` method, supplying the number you want formatted.

```
fmt@format[12.34]
```

```
12.3400
```

This returns a string with the requested format. Now suppose you want to use this ability to format a list of 20000 numbers before writing them to a file.

```
data = Table[Random[], {40 000}];

Map[fmt@format[#] &, data];
```

The `Map` call, which invokes the format method 40000 times, takes 46 seconds on a certain PC (this is wall clock time, not the result of the `Timing` function, which is not accurate for *MathLink* programs on most systems). Clearly this is not acceptable. As a first step, you try using `MethodFunction` because you are calling the same method many times.

```
methodFunc = MethodFunction[fmt, format];
```

Note that you use `fmt` as the first argument to `MethodFunction`. The first argument merely specifies the class; as with virtually all functions in *J/Link* that take a class specification, you can use an object of the class if you desire. The `MethodFunction` that is created can be used on any object of the `DecimalFormat` class, not just the `fmt` object.

```
Map[methodFunc[fmt, #] &, data];
```

Using `methodFunc`, this now takes 36 seconds. There is a slight speed improvement, much less than in earlier versions of *J/Link*. This means you are getting about 1100 calls per second, and it is still not fast enough to be useful. The only thing to do is to write your own Java method that takes an *array* of numbers, formats them all, and returns an array of strings. This will reduce the number of calls from *Mathematica* into Java 40000 down to one.

Here is the code for the trivial Java class necessary. Note that there is nothing about this code that suggests it will be called from *Mathematica* via *J/Link*. This is exactly the same code you would write if you wanted to use this functionality within Java.

```
public class FormatArray {
    public static String[] format(java.text.DecimalFormat fmt,double[] d) {
        String[] result=new String[d.length];
        for (int i = 0; i < d.length; i++)
            result[i] = fmt.format(d[i]);
        return result;
    }
}
```

This new version takes less than 2 seconds.

```
LoadJavaClass["FormatArray"];
FormatArray`format[fmt, data];
```

Reference Counts and Memory Management

Object References in Mathematica

The earlier treatment of JavaObject expressions avoided discussing deeper issues such as reference counts and uniqueness. Every time a Java object reference is returned to *Mathematica*, either as a result of a method or field or an explicit call to JavaNew, *J/Link* looks to see if a reference to this object has been sent previously in this session. If not, it creates a JavaObject expression in *Mathematica* and sets up a number of definitions for it. This is a comparatively time-consuming process. If this object has already been sent to *Mathematica*, in most cases *J/Link* simply creates a JavaObject expression that is identical to the one created previously. This is a much faster operation.

There are some exceptions to this last rule, meaning that sometimes when an object is returned to *Mathematica* a new and different JavaObject expression is created for it, even though this same object has previously been sent to *Mathematica*. Specifically, any time an object's hashCode() value has changed since the last time it was seen in *Mathematica*, the JavaObject expression created will be different. You do not really need to be concerned with the details of this, except to remember that SameQ is not a valid way to compare JavaObject expressions to decide whether they refer to the same object. You must use the SameObjectQ function.

SameObjectQ[*obj1*,*obj2*]	return True if the JavaObject expressions *obj1* and *obj2* refer to the same Java object, False otherwise

Comparing JavaObject expressions.

Here is an example.

```
pt = JavaNew["java.awt.Point", 1, 1]
```

```
«JavaObject[java.awt.Point]»
```

The variable pt refers to a Java Point object. Now put it into a container so you can get it back out later.

```
vec = JavaNew["java.util.Vector"];
vec@add[pt];
```

Now change the value of one of its fields. For a `Point` object, changing the value of one of its fields changes its `hashCode()` value.

```
pt@x = 2;
```

Now you compare the `JavaObject` expression given by `pt` and the `JavaObject` expression created when you ask for the first element of the `Vector` to be returned to *Mathematica*. Even though these are both references to the same Java object, the `JavaObject` expressions are different.

```
pt === vec@elementAt[0]
```

```
False
```

Because you cannot use `SameQ` (`===`) to decide whether two object references in *Mathematica* refer to the same Java object, *J/Link* provides a function, `SameObjectQ`, for this purpose.

```
SameObjectQ[pt, vec@elementAt[0]]
```

```
True
```

You may be wondering why the `SameObjectQ` function is necessary. Why not just call an object's `equals()` method? It certainly gives the correct result for this example.

```
pt@equals[vec@elementAt[0]]
```

```
True
```

The problem with this technique is that `equals()` does not always compare object references. Any class is free to override `equals()` to provide any desired behavior for comparing two objects of that class. Some classes make `equals()` compare the "contents" of the objects, such as the `String` class, which uses it for string comparison. Java provides two distinct equality operations, the == operator and the `equals()` method. The == operator always compares references, returning `true` if and only if the references point to the same object, but `equals()` is often overridden for some other type of comparison. Because there is no method call in Java that mimics the behavior of the language's == operator as applied to object references, *J/Link* needs a `SameObjectQ` function that provides that behavior for *Mathematica* programmers.

In an unusual case where you need to compare object references for equality a very large number of times, the comparative slowness of `SameObjectQ` compared to `SameQ` could become an issue. The only thing that could cause two `JavaObject` expressions that refer to the exact same Java object to be not `SameQ` is if the `hashCode()` value of the object changed between

the times that the two JavaObject expressions were created. If you know this has not happened, then you can safely use SameQ as the test whether they refer to the same object.

ReleaseJavaObject

The Java language has a built-in facility called "garbage collection" for freeing up memory occupied by objects that are no longer in use by a program. Objects become eligible for garbage collection when no references to them exist anywhere, except perhaps in other objects that are also unreferenced. When an object is returned to *Mathematica*, either as a result of a call to JavaNew or as the return value of a method call or field access, the *J/Lin*k code holds a special reference to the object on the Java side to ensure that it cannot be garbage-collected while it is in use by *Mathematica*. If you know that you no longer need to use a given Java object in your *Mathematica* session, you can explicitly tell *J/Link* to release its reference. The function that does this is ReleaseJavaObject. In addition to releasing the *Mathematica*-specific reference in Java, ReleaseJavaObject clears out internal definitions for the object that were created in *Mathematica*. Any subsequent attempt to use this object in *Mathematica* will fail.

```
frm = JavaNew["java.awt.Frame"]
```

```
«JavaObject[java.awt.Frame] »
```

Now tell Java that you no longer need to use this object from *Mathematica*.

```
ReleaseJavaObject[frm]
```

It is now an error to refer to frm.

ReleaseJavaObject[*obj*]	let Java know that you are done using *obj* in *Mathematica*
ReleaseObject[*obj*]	deprecated; replaced by ReleaseJavaObject in *J/Link* 2.0
JavaBlock[*expr*]	all novel Java objects returned to *Mathematica* during the evaluation of *expr* will be released when *expr* finishes
BeginJavaBlock[]	all novel Java objects returned to *Mathematica* between now and the matching EndJavaBlock[] will be released
EndJavaBlock[]	release all novel objects seen since the matching BeginJavaBlock[]
LoadedJavaObjects[]	return a list of all objects that are in use in *Mathematica*
LoadedJavaClasses[]	return a list of all classes loaded into *Mathematica*

J/Link memory management functions.

Calling ReleaseJavaObject will not necessarily cause the object to be garbage-collected. It is quite possible that other references to it exist in Java. ReleaseJavaObject does not tell Java to throw the object away, only that it does not need to be kept around solely for *Mathematica*'s sake.

An important fact about the references that *J/Link* maintains for objects sent to *Mathematica* is that only one reference is kept for each object, no matter how many times it is returned to *Mathematica*. It is your responsibility to make sure that after you call ReleaseJavaObject, you never attempt to use that object through any reference that might exist to it in your *Mathematica* session.

```
frm = JavaNew["java.awt.Frame"];
b1 = JavaNew["java.awt.Button"];
```

The add() method of the Frame class returns the object added, so b2 refers to the same object as b1:

```
b2 = frm@add[b1];
```

If you call ReleaseJavaObject[b1], it is not the *Mathematica* symbol b1 that is affected, but the Java object that b1 refers to. Therefore, using b2 is also an error (or any other way to refer to this same Button object, such as %).

Calling ReleaseJavaObject is often not necessary in casual use. If you are not making heavy use of Java in your session then you will usually not need to be concerned about keeping track of what objects may or may not be needed anymore—you can just let them pile up. There are special times, though, when memory use in Java will be important, and you may need the extra control that ReleaseJavaObject provides.

JavaBlock

ReleaseJavaObject is provided mainly for developers who are writing code for others to use. Because you can never predict how your code will be used, developers should always be sure that their code cleans up any unnecessary references it creates. Probably the most useful function for this is JavaBlock.

JavaBlock automates the process of releasing objects encountered during the evaluation of an expression. Often, a *Mathematica* program will need to create some Java objects with JavaNew, operate with them, perhaps causing other objects to be returned to *Mathematica* as the results

of method calls, and finally return some result such as a number or string. Every Java object encountered by *Mathematica* during this operation is needed only during the lifetime of the program, much like the local variables provided in *Mathematica* by `Block` and `Module`, and in C, C++, Java, and many other languages by block scoping constructs (e.g., {}). `JavaBlock` allows you to mark a block of code as having the property that any new objects returned to *Mathematica* during the evaluation are to be treated as temporary, and released when `JavaBlock` finishes.

It is important to note that the preceding sentence said "new objects". `JavaBlock` will not cause every object encountered during the evaluation to be released, only those that are being encountered for the first time. Objects that have already been seen by *Mathematica* will not be affected. This means that you do not have to worry that `JavaBlock` will aggressively release an object that is not truly temporary to that evaluation.

It is not enough simply to call `ReleaseJavaObject` on every object you create with `JavaNew`, because many Java method calls return objects. You may not be interested in these return values, or you may never assign them to a named variable because they may be chained together with other calls (as in `obj@returnsObject[]@foo[]`), but you still need to release them. Using `JavaBlock` is an easy way to be sure that all novel objects are released when a block of code finishes.

`JavaBlock[`*expr*`]` returns whatever *expr* returns.

Many *J/Link Mathematica* programs will have the following structure:

```
MyFunc[args__] :=
     JavaBlock[
          Module[{locals},
               ...
           ]
     ]
```

It is very common to write a function that creates and manipulates a number of `JavaObject` expressions, and then returns one of them, the rest being temporary. To facilitate this, if the return value of a `JavaBlock` is a single `JavaObject`, it will not be released.

```
MyOtherFunc[args__] :=
    JavaBlock[
        Module[{obj},
            ...
            obj = JavaNew["java.awt.Frame"];
            ...
            Return[obj]
   (* OK: obj will not be released when JavaBlock finishes. *)
        ]
    ]
```

New in *J/Link* 2.1 is the KeepJavaObject function, which allows you to specify an object or sequence of objects that should not be released when the JavaBlock ends. Calling KeepJavaObject on a single object or sequence of objects means they will not be released when the first enclosing JavaBlock ends. If there is an outer enclosing JavaBlock, the objects will be freed when *it* ends, however, so if you want the objects to escape a nested set of JavaBlock expressions, you must call KeepJavaObject at each level. Alternatively, you can call KeepJavaObject[*obj*, Manual], where the Manual argument tells *J/Link* that the object should not be released by any enclosing JavaBlock expressions. The only way such object will be released is if you manually call ReleaseJavaObject on it. Here is an example that uses KeepJavaObject to allow you to return a list of two objects without them being released:

```
MyOtherFunc[args__] :=
    Module[{obj1, obj2, obj3},
        JavaBlock[
            obj1 = JavaNew["java.awt.Frame"];
            obj2 = JavaNew["java.awt.Button"];
            obj3 = JavaNew["SomeTemporaryObject"];
            ...
            KeepJavaObject[obj1, obj2];
            {obj1, obj2}
        ]
     ]
```

BeginJavaBlock and EndJavaBlock can be used to provide the same functionality as JavaBlock across more than one evaluation. EndJavaBlock releases all novel Java objects returned to *Mathematica* since the previous matching BeginJavaBlock. These functions are mainly of use during development, when you might want to set a mark in your session, do some work, and then release all novel objects returned to *Mathematica* since that point. BeginJavaBlock and EndJavaBlock can be nested. Every BeginJavaBlock should have a matching EndJavaBlock, although it is not a serious error to forget to call EndJavaBlock, even if you have nested levels of them—you will only fail to release some objects.

LoadedJavaObjects and LoadedJavaClasses

LoadedJavaObjects[] returns a list of all Java objects that are currently referenced in *Mathematica*. This includes all objects explicitly created with JavaNew and all those that were returned to *Mathematica* as the result of a Java method call or field access. It does not include objects that have been released with ReleaseJavaObject or through JavaBlock. LoadedJavaObjects is intended mainly for debugging. It is very useful to call it before and after some function you are working on. If the list grows, your function leaks references, and you need to examine its use of JavaBlock and/or ReleaseJavaObject.

LoadedJavaClasses[] returns a list of JavaClass expressions representing all classes loaded into *Mathematica*. Like LoadedJavaObjects, LoadedJavaClasses is intended mainly for debugging. Note that you do not have to determine if a class has already been loaded before you call LoadJavaClass. If the class has been loaded, LoadJavaClass does nothing but return the appropriate JavaClass expression.

Exceptions

How Exceptions Are Handled

J/Link handles Java exceptions automatically. If an uncaught exception is thrown during any call into Java, you will get a message in *Mathematica*. Here is an example that tries to format a real number as an integer.

```
LoadClass["java.lang.Integer"];
Integer`parseInt["1234.5"]
```

```
Java::excptn :
    A Java exception occurred : java.lang.ArrayIndexOutOfBoundsException.
$Failed
```

If the exception is thrown before the method returns a result to *Mathematica*, as in the example, the result of the call will be $Failed. As discussed later in "Manually Returning a Result to *Mathematica*", it is possible to write your own methods that manually send a result to *Mathematica* before they return. In such cases, if an exception is thrown after the result is sent to *Mathematica* but before the method returns, you will get a warning message reporting the exception, but the result of the call will be unaffected.

If the Java code was compiled with debugging information included, the *Mathematica* message you get as a result of an exception will show the full stack trace to the point where the exception occurred, with the exact line numbers in each file.

The JavaThrow Function

In some cases, you may want to cause an exception to be thrown in Java. This can be done with the JavaThrow function. JavaThrow is new in *J/Link* 2.0 and should be considered experimental. Its behavior might change in future versions.

JavaThrow[*exceptionObj*]	throw the given exception object in Java

Throwing Java exceptions from *Mathematica.*

You will only want to use JavaThrow in *Mathematica* code that is itself called from Java. It is quite common for *J/Link* programs written in *Mathematica* to involve both calls from *Mathematica* into Java and calls from Java back to *Mathematica*. Such "callbacks" to *Mathematica* are used extensively in *Mathematica* programs that create Java user interfaces, as described in detail later in the section "Creating Windows and Other User Interface Elements". For example, you can associate a *Mathematica* function to be called when the user clicks a Java button. This *Mathematica* function is called directly from Java, and you might want it to behave just like a Java method, including having the ability to throw Java exceptions.

An example of throwing an exception in a callback from a user interface action like clicking a button is not very realistic because there is typically nothing in Java to catch such exceptions; thus they are essentially ignored. A more meaningful example would be a program that involved a mix of Java and *Mathematica* code where, for flexibility and ease of development reasons, you have a *Mathematica* function being called to implement the "guts" of a Java method that can throw an exception. As a concrete example, say you are doing XML processing with Java and *Mathematica* using the SAX (Simple API for XML) API. SAX processing is based on a set of handler methods that are called as certain events occur during parsing of the XML document. Each such method can throw a SAXException to indicate an error and halt the parsing. You want to implement these handler methods in *Mathematica* code, and thus you want a way to throw a SAXException from *Mathematica*. Here is a hypothetical example of one such handler method, the startDocument() method, which is invoked by the SAX engine when document processing starts:

```
startDocument[] :=
 If[!readyToAcceptParsingEvents, JavaThrow[JavaNew["org.xml.sax.SAXException",
    "Mathematica code has not been initialized"]]]
```

After a call to JavaThrow, the rest of the *Mathematica* function executes normally, but there is no result returned to Java.

Returning Objects "by Value" and "by Reference"

References and Values

J/Link provides a mapping between certain *Mathematica* expressions and their Java counterparts. What this means is that these *Mathematica* expressions are automatically converted to and from their Java counterparts as they are passed between *Mathematica* and Java. For example, Java integer types (`long`, `short`, `int`, and so on) are converted to *Mathematica* integers and Java real types (`float` and `double`) are converted to *Mathematica* real numbers. Another mapping is that Java objects are converted to `JavaObject` expressions in *Mathematica*. These `JavaObject` expressions are *references* to Java objects—they have no meaning in *Mathematica* except as they are manipulated by *J/Link*. However, some Java objects are things that have meaningful values in *Mathematica*, and these objects are by default converted to values. Examples of such objects are strings and arrays.

You could say, then, that Java objects are by default returned to *Mathematica* "by reference", except for a few special cases. These special cases are strings, arrays, complex numbers (discussed later), `BigDecimal` and `BigInteger` (discussed later), and the "wrapper" classes (e.g., `java.lang.Integer`). You could say that these exceptional cases are returned "by value". The table in "Conversion of Types between Java and *Mathematica*" shows how these special Java object types are mapped into *Mathematica* values.

In summary, every Java object that has a meaningful value representation in *Mathematica* is converted into this value, simply because that is the most useful behavior. There are times, however, when you might want to override this default behavior. Probably the most common reason for doing this is to avoid unnecessary traffic of large expressions over *MathLink*.

`ReturnAsJavaObject[`*expr*`]`	a Java object returned by *expr* will be in the form of a reference
`ByRef[`*expr*`]`	deprecated; replaced by `ReturnAsJavaObject` in *J/Link* 2.0
`JavaObjectToExpression[`*obj*`]`	give the value of the Java object *obj* as a *Mathematica* expression
`Val[`*obj*`]`	deprecated; replaced by `JavaObjectToExpression` in *J/Link* 2.0

"By reference" and "by value" control.

ReturnAsJavaObject

Consider the case where you have a static method in class `MyClass` called `arrayAbs()` that takes an array of doubles and returns a new array where each element is the absolute value of the corresponding element in the argument array. The declaration of this method thus looks like `double[] arrayAbs(double[] a)`. This is how you would call such a method from *Mathematica*.

```
LoadJavaClass["MyClass", StaticsVisible → True];
arrayAbs[{1., -2., 3., 4.}]
{1., 2., 3., 4.}
```

The example showed how you probably want the method to work: you pass a *Mathematica* list and get back a list. Now assume you have another method named `arraySqrt()` that acts like `arrayAbs()` except that it performs the `sqrt()` function instead of `abs()`.

```
arraySqrt[arrayAbs[{1., -2., 3., 4.}]]
{1., 1.41421, 1.73205, 2.}
```

In this computation, the original list is sent over *MathLink* to Java and a Java array is created with these values. That array is passed as an argument to `arrayAbs()`, which itself creates and returns another array. This array is then sent back to *Mathematica* via *MathLink* to create a list, which is then promptly sent back to Java as the argument for `arraySqrt()`. You can see that it was a waste of time to send the array data back to *Mathematica*—you had a perfectly good array (the one returned by the `arrayAbs()` method) living on the Java side, ready to be passed to `arraySqrt()`, but instead you sent its contents back to *Mathematica* only to have it immediately come back to Java again as a new array with the same values! For this example, the cost is negligible, but what if the array has 200,000 elements?

What is needed is a way to let the array data remain in Java and return only a reference to the array, not the actual data itself. This can be accomplished with the `ReturnAsJavaObject` function.

```
ReturnAsJavaObject[arrayAbs[{1., -2., 3., 4.}]]
«JavaObject[[D] »
```

Note that the class name of the `JavaObject` is `"[D"`, which, although a bit cryptic, is the actual Java class name of a one-dimensional array of doubles. Here is how the computation looks using `ReturnAsJavaObject`:

```
arraySqrt[ReturnAsJavaObject[arrayAbs[{1., -2., 3., 4.}]]]
{1., 1.41421, 1.73205, 2.}
```

Earlier you saw `arraySqrt()` being called with an argument that was a *Mathematica* list of reals. Here it is being called with a reference to a Java object that is a one-dimensional array of doubles. All methods and fields that take an array can be called from *Mathematica* with either a *Mathematica* list or a reference to a Java array of the appropriate type.

Strings are the other type for which `ReturnAsJavaObject` is useful. Like arrays, strings have the two properties that (1) they are represented in Java as objects but also have a meaningful *Mathematica* value, and (2) they can be large, so it is useful to be able to avoid passing their data back and forth unnecessarily. As an example, say your class `MyClass` has a static method that appends to a string a digit taken from an external device that you are controlling from Java. It takes a string and returns a new one, so its signature is `static String appendDigit (String s)`. You have a *Mathematica* variable named `veryLongString` that holds a long string, and you want to append to this string 100 times. This code will cause the string contents to make 100 round trips between *Mathematica* and Java.

```
Do[veryLongString = appendString[veryLongString], {100}];
```

Using `ReturnAsJavaObject` lets the strings remain on the Java side, and thus it generates virtually no *MathLink* traffic.

```
Do[veryLongString = ReturnAsJavaObject[appendString[veryLongString]], {100}];
```

This example is somewhat contrived, since repeatedly appending to a growing string is not a very efficient style of programming, but it illustrates the issues.

When the Do loop is executed, veryLongString gets assigned values that are not *Mathematica* strings, but JavaObject expressions that refer to strings residing in Java. That means that appendString[] gets called with a *Mathematica* string the very first iteration, but with a JavaObject expression thereafter. As is the case with arrays, any Java method or field that takes a string can be called in *Mathematica* either with a string or a JavaObject expression that refers to one. The veryLongString variable started out holding a string, but at the end of the loop it holds a JavaObject expression.

```
veryLongString
«JavaObject[java.lang.String]»
```

At some point, you probably want an actual *Mathematica* string, not this string object reference. How do you get the value back? You will visit this example again later when the JavaObjectToExpression function is introduced.

In summary, the ReturnAsJavaObject function causes methods and fields that return objects that would normally be converted into *Mathematica* values to return references instead. It is an optimization to avoid unnecessarily passing large amounts of data between *Mathematica* and Java, and as such it will be useful primarily for very large arrays and strings. As with all optimizations, you should not concern yourself with ReturnAsJavaObject unless you have some code that is running at an unacceptable speed, or you know ahead of time that the code you are writing will benefit measurably from it. Objects of most Java classes have no meaningful "by value" representation in *Mathematica*, and they are always returned "by reference". ReturnAsJavaObject will have no effect in these cases.

Finally, note that ReturnAsJavaObject has no effect on methods in which the Java programmer manually sends the result back to *Mathematica* (this topic is discussed later in this User Guide). Manually returning a result bypasses the normal result-handling routines in *J/Link*, so there is no chance for the ReturnAsJavaObject request to be accommodated.

JavaObjectToExpression

In the previous section, you saw how the ReturnAsJavaObject function can be used to cause objects normally returned to *Mathematica* by value to be returned by reference. It is necessary to have a function that does the reverse—takes a reference and converts it to its value representation. That function is JavaObjectToExpression.

Returning to the earlier appendString example, you used ReturnAsJavaObject to avoid costly passing of string data back and forth over *MathLink*. The result of this was that the veryLongString variable now held a JavaObject expression, not a literal *Mathematica* string. JavaObjectToExpression can be used to get the value of this string object as a *Mathematica* string.

```
JavaObjectToExpression[veryLongString]
```

```
037118086362644534489492294928989287822791948284089742269122236592851667829700627394053209887
   2893368
```

The majority of Java objects have no meaningful value representation in *Mathematica*. These objects can only be represented in *Mathematica* as JavaObject expressions, and using JavaObjectToExpression on them has no effect.

The ReturnAsJavaObject function is not the only way to get a JavaObject expression for an object that is normally returned to *Mathematica* as a value. The JavaNew function always returns a reference.

```
JavaNew["java.lang.String", "a string"]
```

```
«JavaObject[java.lang.String] »
```

```
JavaObjectToExpression[%]
```

```
a string
```

The next section introduces the MakeJavaObject function, which is easier than using JavaNew to construct Java objects out of *Mathematica* strings and arrays.

MakeJavaObject and MakeJavaExpr

Preamble

In addition to JavaNew, which calls a class constructor, *J/Link* provides two convenience functions for creating Java objects from *Mathematica* expressions. These functions are MakeJavaObject and MakeJavaExpr. Do not get them confused, despite their similar names. MakeJavaObject is a commonly used function for constructing objects of some special types. MakeJavaExpr is an advanced function that creates an object of *J/Link*'s Expr class representing an arbitrary *Mathematica* expression.

MakeJavaObject

`MakeJavaObject[`*val*`]`	construct an object of the appropriate type to represent the *Mathematica* expression *val* (numbers, strings, lists, and so on)

`MakeJavaObject`.

When you call a Java method from *Mathematica* that takes, say, a Java `String` object, you can call it with a *Mathematica* string. The internals of *J/Link* will construct a Java string that has the same characters as the *Mathematica* string, and pass that string to the Java method. Sometimes, however, you want to pass a string to a method that is typed to take `Object`. You cannot call such a method from *Mathematica* with a string as the argument because although *J/Link* recognizes that a *Mathematica* string corresponds to a Java string, it does not recognize that a *Mathematica* string corresponds to a Java `Object`. It does this deliberately, for the sake of imposing as much type safety as possible on calls into Java. For this example, assume that the class `MyClass` has a method with the following signature:

```
void foo(Object obj);
```

Assume also that `theObj` is an object of this class, created with `JavaNew`. Try to call `foo` with a literal string.

```
theObj@foo["this is a string"]
```

```
Java::argxs :
    The method foo was called with an incorrect number or type of arguments.
```

```
$Failed
```

It fails for the reason given above. To call a Java method that is typed to take an `Object` with a string, you must first explicitly create a Java string object with the appropriate value. You can do this using `JavaNew`.

```
javaStr = JavaNew["java.lang.String", "this is a string"]
```

```
«JavaObject[java.lang.String] »
```

Now it works, because the argument is a `JavaObject` expression.

```
theObj@foo[javaStr]
```

To avoid having to call JavaNew to create a Java string object, *J/Link* provides the MakeJavaObject function.

```
javaStr = MakeJavaObject["this is a string"];
```

In the case of a string, MakeJavaObject just calls JavaNew for you. Of course, it would not be of much use if it could only construct String objects. The same issue arises with other Java objects that are direct representations of *Mathematica* values. This includes the "wrapper" classes like java.lang.Integer, java.lang.Boolean, and so on, and the array classes. If you want to call a Java method that takes a java.lang.Integer as an argument, you can call it from *Mathematica* with a raw integer. But if you want to pass an integer to a method that is typed to take an Object, you must explicitly create an object of type java.lang.Integer—*J/Link* will not construct one automatically from an integer argument. It is simpler to call MakeJavaObject than JavaNew for this.

```
MakeJavaObject[42]
```

```
«JavaObject[java.lang.Integer] »
```

When given an integer argument, MakeJavaObject always constructs a java.lang.Integer, never a java.lang.Short, java.lang.Long, or other "integer" Java wrapper object. Similarly, if you call MakeJavaObject with a real number, it creates a java.lang.Double, never a java.lang.Float. If you require an object of one of these other types, you will have to call JavaNew explicitly.

MakeJavaObject also works for Boolean values.

```
MakeJavaObject[True]
```

```
«JavaObject[java.lang.Boolean] »
```

If MakeJavaObject were only a shortcut for calling JavaNew, it would not be all that useful. It becomes indispensable, however, for creating objects of an array class. Recall that in Java, arrays are objects and they belong to a class. These classes have cryptic names, but if you know them you can create array objects with JavaNew. When creating array objects, the second argument to JavaNew is a list giving the length in each dimension. Here you create a 2×3 array of ints.

```
intArray2D = JavaNew["[[I", {2, 3}]
```

```
«JavaObject[[[I] »
```

JavaNew lets us create array objects, but it does not let us supply initial values for the elements of the array. MakeJavaObject, on the other hand, takes a *Mathematica* list and converts it into a Java array object with the same values.

```
intArray2D = MakeJavaObject[{{1, 2, 3}, {4, 5, 6}}]
«JavaObject[[[I] »
```

Thus, MakeJavaObject is particularly useful for creating array objects, because it lets you supply the initial values for the array elements, and it frees you from having to learn and remember the names of the Java array classes ([[I for a two-dimensional array of ints, [D for a one-dimensional array of doubles, and so on). MakeJavaObject can create arrays up to three dimensions deep of integers, doubles, strings, Booleans, and objects.

The JavaObjectToExpression function is discussed in the section "JavaObjectToExpression", and you can think of MakeJavaObject as being the inverse of JavaObjectToExpression. MakeJavaObject takes a *Mathematica* expression that has a corresponding Java object that can represent its value, and creates that object. It literally "makes it into a Java object". The JavaObjectToExpression function goes the other way—it takes a Java object that has a meaningful *Mathematica* representation and converts it into that expression. It will always be the case that, for any x that MakeJavaObject can operate on,

```
JavaObjectToExpression[MakeJavaObject[x]] === x
```

Remember that MakeJavaObject is not a commonly used function. You do not need to explicitly construct Java objects from *Mathematica* strings, arrays, and so on, just to pass them to Java methods—*J/Link* does this automatically for you. But even though *J/Link* will create objects automatically from certain arguments in most circumstances, it will not do so when an argument is typed as a generic Object. Then you must create a JavaObject yourself, and MakeJavaObject is the easiest way to do this.

The code for the SetInternetProxy function discussed in the section SetInternetProxy provides a concrete example of using MakeJavaObject. To specify proxy information (for users behind firewalls), you need to set some system properties using the Properties class. This class is a subclass of Hashtable, so it has a method with the signature

```
Object put(Object key, Object value);
```

You should always specify keys and values for `Properties` in the form of strings. Thus, you might try this from *Mathematica*.

```
LoadJavaClass["java.lang.System"];
System`getProperties[]@put["proxySet", "true"]
```

```
Java::argx :
  Method named put defined in class java.util.Properties was called with
    an incorrect number or type of arguments. The
    arguments, shown here in a list, were {proxySet, true}.
$Failed
```

For this to work, you need to use `MakeJavaObject` to create Java `String` objects:

```
System`getProperties[]@put[MakeJavaObject["proxySet"], MakeJavaObject["true"]]
```

MakeJavaExpr

To understand the `MakeJavaExpr` function, you need to understand the motivation for *J/Link*'s `Expr` class, which is discussed in detail in "Motivation for the Expr Class". Basically, an `Expr` is a Java object that can represent an arbitrary *Mathematica* expression. Its main use is as a convenience for Java programmers who want to examine and operate on *Mathematica* expressions in Java. Sometimes it is useful to have a way of creating `Expr` objects in the *Mathematica* language instead of from Java. `MakeJavaExpr` is the function that fills this need.

`MakeJavaExpr[`*expr*`]`	construct an object of *J/Link*'s `Expr` class that represents the *Mathematica* expression

`MakeJavaExpr.`

Note that if you are calling a Java method that is typed to take an `Expr`, then you do not have to call `MakeJavaExpr` to construct an `Expr` object. *J/Link* will automatically convert any expression you supply as the argument to an `Expr` object, as it does with other automatic conversions. Like `MakeJavaObject`, `MakeJavaExpr` is used in cases where you are calling a method that takes a generic `Object`, not an `Expr`, and therefore *J/Link* will not perform any automatic conversion for you. In such cases you need to explicitly create an `Expr` object out of some *Mathematica* expression. One reason you might want to do this is to store a *Mathematica* expression in Java for retrieval later.

Here is a simple example of `MakeJavaExpr`. This demonstrates a few methods from the `Expr` class, which has a number of *Mathematica*-like methods for examining, modifying, and extracting portions of expressions. Of course, this is a highly contrived example—if you wanted to know the length of an expression you would just call *Mathematica*'s `Length[]` function. The `Expr` methods demonstrated here are typically called from Java, not *Mathematica*.

```
e = MakeJavaExpr[1 + 2 x + x^2]
```

```
«JavaObject[com.wolfram.jlink.Expr] »
```

```
e@length[]
```

```
3
```

```
e@part[3]
```

$$x^2$$

```
e@insert[x^3, -1]
```

$$1 + 2\,x + x^2 + x^3$$

Note that `Expr` objects, like *Mathematica* expressions, are immutable. The above call to `insert()` did not modify `e`; instead, it returned a new `Expr`.

```
JavaObjectToExpression[e]
```

$$1 + 2\,x + x^2$$

If you are having trouble understanding why you might want to use `MakeJavaExpr` in a *Mathematica* program, do not worry. It is an advanced function that few programmers will have any use for.

Creating Windows and Other User Interface Elements

Preamble

One of the most useful applications for *J/Link* is to write user interface elements for *Mathematica* programs. Examples of such elements would be a progress bar monitoring the completion of a computation, a window that displays an image or animation, a dialog box that prompts the user for input or helps them compose a proper call of an unfamiliar function, or a mini-application that leads the user through the steps of an analysis. These types of user interfaces are distinct from what you might write for a Java program that uses *Mathematica* in the background in that they "pop up" when the user invokes some *Mathematica* code. They do not replace the notebook front end, they just augment it. In this way, they are like an extension of the palettes and other specialty notebook elements you can create in the front end.

Mathematica with *J/Link* is an extremely powerful and productive environment for creating user interfaces. The complexity of user interface code is ideally suited to the interactive line-at-a-time nature of *J/Link* development. You can literally build, modify, and experiment with your user interface *while it is running*.

Anyone considering writing user interfaces for *Mathematica* programs should also look at *GUIKit. GUIKit* is built on top of *J/Link*, and provides an extremely high-level means of creating interfaces. Further discussion of *GUIKit* is beyond the scope of this manual, but be aware that *GUIKit* was specifically designed to provide an easier means of creating user interfaces than writing in "raw" *J/Link*, as described here.

Interactive and Non-Interactive Interfaces

To write *Mathematica* programs that create Java windows you need to understand important distinctions between several types of such user interfaces. These distinctions relate to how they interact with the *Mathematica* kernel.

At the highest level of categorization, there is a distinction between "interactive" and "non-interactive" interfaces. The interactiveness under consideration here is with the *Mathematica*

kernel, not with the user. What we are calling non-interactive user interfaces have no need to communicate back to *Mathematica*, although they typically are controlled by *Mathematica*. Such interfaces often accept no user input at all—they are created, manipulated, and destroyed by *Mathematica* code. An example of this type is a window that shows a progress bar (a complete progress bar program is presented in "A Progress Bar"). A progress bar does not return a result to *Mathematica* and it does not need to respond to user actions, at least not by interacting with *Mathematica*. In other words, the window may go away when its close box is clicked (a user action), but this is not relevant to *Mathematica* because it does not return a result or trigger a call back into *Mathematica*. A progress bar is completely driven by a *Mathematica* program. The flow of information is in one direction only.

Such user interfaces typically have lifetimes that are encompassed by a single *Mathematica* program, as is the case with a progress bar. This is not required, however. Hosting an applet in its own window, as described in "Hosting Applets", is an example where the window lives on after the code that created it ends execution. The applet window is only dismissed when the user clicks in its close box. Again, though, the important property is that the applet does not need to interact with *Mathematica*.

This type of user interface, which requires no interaction back with *Mathematica*, poses no special issues that need to be discussed in this section. A program that creates, runs, and destroys such an interface is very much like a non-GUI Java computation that is accomplished with a series of calls into Java. It just happens to produce a visual effect. You can examine the progress bar code in "A Progress Bar" if you want to see a fully fleshed out example.

The more common “interactive” type of user interface needs to communicate back to *Mathematica*. This might be to return a result, like a typical modal input dialog, or to initiate a computation as a consequence of the user clicking a button. To understand the special problem this imposes, it is useful to examine some basic considerations about the kernel’s “main loop”, whereby it acquires input, evaluates it, and sends off any output.

When the *Mathematica* kernel is being used from the front end, it spends most of its life waiting for input to arrive on the *MathLink* that it uses to communicate with the front end. This *MathLink* is given by `$ParentLink`, and it is `$ParentLink` that has the kernel’s “attention”. When input arrives on `$ParentLink`, it is evaluated, any results are sent back on the link, and the kernel goes back to waiting for more input on `$ParentLink`. When *J/Link* is being used, the

kernel has another *MathLink* open, the one that connects to the Java runtime. When you execute some code that calls into Java, the kernel sends something to Java and then blocks waiting for the return value from Java. During this period when the kernel is waiting for a return value from Java, the Java link has the kernel's attention. It is only during this period of time that the kernel is paying attention to the Java link. A more general way of saying this is that the kernel is only listening for input arriving from Java when it has been specifically instructed to do so. The rest of the time it is listening only to `$ParentLink`, which is typically the notebook front end.

Consider what happens when the user clicks on a button in your Java window and that button tries to execute some code that calls into *Mathematica*. The Java side sends something to *Mathematica* and then waits for the result, but the kernel will never get the request because it is not paying attention to the Java link. It is necessary to use some means to tell the kernel to look for input arriving on the Java link. *J/Link* provides three different ways to manage the kernel's attention to the Java link, and thereby control its readiness to accept requests for evaluations initiated by the Java side.

These three ways can be called "modal", "modeless", and "manual". In modal interaction, characterized by the use of the `DoModal` *Mathematica* function, the kernel is pointed at the Java link until the Java side releases it. The kernel is a complete slave to the Java side, and is unavailable for any other computations. In modeless interaction, characterized by the use of the `ShareKernel` *Mathematica* function, the kernel is kept in a state where it is receptive to evaluation requests arriving from either the notebook front end or Java, evenly sharing its attention between these two programs. Lastly, there is a manual mode, characterized by the use of the `ServiceJava` *Mathematica* function, which in some ways is intermediate between modal and modeless operation. Here, you manually instruct the kernel to allow single requests from Java while in the middle of running a larger program. The next few sections are devoted to further exploration of these types of user interfaces.

Before continuing, it is important to remember that all these issues about how to prepare the kernel for computations arriving from Java are only relevant for computations *initiated* in Java, typically by user actions like clicking a button. Calls from Java to *Mathematica* that are part of a back-and-forth series of calls that involve a call from *Mathematica* into Java are not a problem. Any time *Mathematica* has called into Java, *Mathematica* is actively listening for results arriving from Java. This may sound confusing, but that is mostly because it is only in a much later

section that discusses writing your own Java methods to be called from *Mathematica*; such methods can call back to *Mathematica* for computations before they return their result (typical examples are to print something in the notebook window or display a message). These are true *callbacks* into *Mathematica*, and *Mathematica* is always ready to handle them. In contrast, calls to *Mathematica* that occur as the result of a user action in the Java side are, in effect, a surprise to *Mathematica*, and it is not normally in a state where it is ready to accept them.

Modal versus Modeless Operation

A common type of user interface element is like a modal dialog: once it is displayed, the *Mathematica* program hangs waiting for the user to dismiss the window. Typically, this is because the window returns a result to *Mathematica*, so it is not meaningful for *Mathematica* to continue until the window is closed. An example of such a window is a simple input window that asks the user for some value, which it returns to *Mathematica* when the **OK** button is clicked.

It is important to understand our slightly generalized use of the term "modal" to describe these windows. They may not be modal in the traditional sense that they must be dismissed before anything else can be done in the user interface. Rather, they are modal with respect to the *Mathematica* kernel—the kernel cannot do anything else until they are closed. A Java window that you create might not be modal with respect to other Java windows on the screen (i.e., a dialog might not have the `isModal` property set), but it ties up the kernel's attention until it is dismissed.

Another type of user interface element is like a modeless dialog: after it is displayed, the *Mathematica* program that created it will finish, leaving the window visible and usable while the user continues working in the notebook front end. This sounds a lot like the first type of user interface element described earlier, but these windows are distinguished by the fact that they can initiate interactions with *Mathematica* while they are visible. An example would be a window that lets users load packages into *Mathematica* by selecting them from a scrolling list. You write a *J/Link* program that creates this window, displays it, and returns. The window is left open and usable until the user clicks in its close box. In the meantime, the user is free to continue working in the front end, going back to use this Java window whenever it is convenient.

Such a window is almost like another type of notebook or palette window in the front end. You can have any number of front end or Java windows open at once, and active, meaning that they

can be used to initiate computations in *Mathematica*. They are each their own little interface onto the same kernel. What is different about the Java window is that it is much more general than a notebook window, and, importantly, it lives in a different application layer than the front end. This last fact makes the Java window in effect a second front end, rather than an extension of the notebook front end. To accommodate such a second front end, the kernel must be kept in a special state that allows it to handle requests for evaluations arriving from either the notebook front end or Java.

Before presenting examples of how to implement modal and modeless windows, it is necessary to jump ahead a little bit and explain the main mechanism by which Java user interface elements can communicate with *Mathematica*.

Handling Events with Mathematica Code: The "MathListener" Classes

User interface elements typically have active components like buttons, scrollbars, menus, and text fields that need to trigger some action when they are clicked. In the Java event model, components fire events in response to user actions, and other components indicate their interest in these events by registering as event listeners. In practice, though, components do not usually act as event listeners directly. Instead, the programmer writes an adapter class that implements the desired event-listener interface and calls certain methods in the component in response to various events. This avoids having to subclass the responding component just to have it act as an event listener. The only specialty code goes into the adapter class, allowing the components that fire and respond to events to be generic.

As an example, say you are writing a standard Java program and you have a button that you want to use to control the appearance of a text area. Clicking the button should toggle between black text on a white background and white text on a black background. Buttons fire `ActionEvent`s when they are clicked, and a class that wants to receive notifications of clicks must implement the `ActionListener` interface, and register with the button by calling its `addActionListener` method. You would write a class, perhaps called `MyActionAdapter`, that implements `ActionListener`. In its `actionPerformed()` method, which is what will be called when the button is clicked, you would call the appropriate methods to set the foreground and background colors of the text area.

If you have ever used a Java GUI builder that lets you create an application by dropping components on a form and then wiring them together via events, the code that is being generated for you consists in large part of adapter classes that manage the logic of calling certain methods in the target objects when events are fired by the source objects.

What all this is leading up to is simply that the wiring of components in a GUI typically involves writing a lot of Java code in the form of classes that implement various event-listener interfaces. *J/Link* programmers want to write GUIs that use the standard Java event model, and they should not have to write Java code to do it. The solution is simple: *J/Link* provides a complete set of classes that implement the standard event-listener interfaces and whose actions are to call back into *Mathematica* to execute user-defined code. This brings all the event-handling logic down into *Mathematica*, where it can be scripted like every other part of the program.

Not only does this solution preserve the "pure *Mathematica*" property of even complex Java GUIs, it is vastly more flexible than writing a traditional application in Java. When you write in Java, or use a fancy drag-and-drop GUI builder, you hard-code the event logic. You have to decide at compile time what every click, scroll, and keystroke will do. But when you use *J/Link*, you decide how your program is wired together at run time. You can even change the behavior on the fly simply by typing a few lines of code.

J/Link provides implementations of all the standard AWT event-listener classes. These classes are named after the interfaces they implement, with "Math" prepended. Thus, the class that implements `ActionListener` is `MathActionListener`. (Perhaps these classes would be better named `MathXXXAdapter`.) The following table shows a summary of all the `MathListener` classes, the methods they implement, and the arguments they send to your *Mathematica* handler function.

class	*methods*	*arguments to Mathematica handler*
MathActionListener	actionPerformed (ActionEvent e)	e, e.getActionCommand () (String)
MathAdjustmentListener	adjustmentValueChanged (AdjustmentEvent e)	e, e.getAdjustmentType (), (Integer) e.getValue () (Integer)
MathComponentListener	componentHidden (ComponentEvent e) componentShown (ComponentEvent e) componentResized (ComponentEvent e) componentMoved (ComponentEvent e)	e
MathContainerListener	componentAdded (ContainerEvent e) componentRemoved (ContainerEvent e)	e
MathFocusListener	focusGained (FocusEvent e) focusLost (FocusEvent e)	e
MathItemListener	itemStateChanged (ItemEvent e)	e, e.getStateChange () (Integer)
MathKeyListener	keyPressed (KeyEvent e) keyReleased (KeyEvent e) keyTyped (KeyEvent e)	e, e.getKeyChar (),(Integer) e.getKeyCode () (Integer)
MathMouseListener	mouseClicked (MouseEvent e) mouseEntered (MouseEvent e) mouseExited (MouseEvent e) mousePressed (MouseEvent e) mouseReleased (MouseEvent e)	e, e.getX (), (Integer) e.getY (), (Integer) e.getClickCount () (Integer)

MathMouseMotionListener	mouseMoved (MouseEvent e) mouseDragged (MouseEvent e)	e, e.getX (), (Integer) e.getY (), (Integer) e.getClickCount () (Integer)
MathPropertyChangeListener	propertyChanged (PropertyChangeEvent e)	e
MathTextListener	textValueChanged (TextEvent e)	e
MathVetoableChangeListener	vetoableChange (PropertyChangeEvent e)	e (veto the change by returning False from your handler)
MathWindowListener	windowOpened (WindowEvent e) windowClosed (WindowEvent e) windowClosing (WindowEvent e) windowActivated (WindowEvent e) windowDeactivated (WindowEvent e) windowIconified (WindowEvent e) windowDeiconified (WindowEvent e)	e

Listener classes provided with *J/Link*.

As an example of how to read this table, take the `MathKeyListener` class. `MathKeyListener` implements the `KeyListener` interface, which contains the methods `keyPressed()`, `keyReleased()`, and `keyTyped()`. If you register a `MathKeyListener` object with a component that fires `KeyEvent`s, then these three methods will be called in response to the key events they are named after. When any of these methods are called, they will call into *Mathematica* and execute a user-defined function, passing it three arguments: the `KeyEvent` object itself, followed by two integers that are the results of the event object's `getKeyChar()` and `getKeyCode()` methods. All the `MathListener` classes pass your handler function the event object itself, and a few, like this one, pass additional integer arguments that are commonly needed values. This just saves you the overhead of having to call back into Java to get these additional values.

To specify the *Mathematica* function associated with any of the methods of a MathListener object, call the object's setHandler() method. setHandler() takes two strings, the first of which is the name of the event-handler method (e.g., "actionPerformed" or "keyPressed"), and the second of which is the *Mathematica* function that should be called in response. The *Mathematica* function can be a name, as in "myButtonFunction" or a pure function (specified as a string). The reason for supplying the name of the actual Java method in the listener interface is that many of the listeners have multiple methods. setHandler() returns True if the handler was set correctly and False otherwise (for example, if the method you named is not spelled correctly).

obj@setHandler["*methodName*","*funcName*"]	set the *Mathematica* function that will be called when the MathListener object *obj*'s event-handler method methodName() is called.

Assigning the *Mathematica* function that will be called in response to an event notification.

The use of these classes will become clear in the simple examples that follow for modal and modeless windows, and in the more fully worked-out examples in the sections "A Simple Modal Input Dialog" and "A Piano Keyboard".

You are not required to use the *J/Link* MathListener classes for creating calls into *Mathematica* triggered by user actions. They are provided simply as a convenience. You could write your own classes to handle events and put calls into *Mathematica* directly into their code. All the "MathListener" classes in *J/Link* are derived from an abstract base class called, appropriately, MathListener. The code in MathListener takes care of all of the details of interacting with *Mathematica*, and it also provides the setHandler() methods that you use to associate events with *Mathematica* code. Users who want to write their own classes in MathListener style (for example, for one of the Swing-specific event-listener interfaces, which *J/Link* does not provide) are strongly encouraged to make their classes subclasses of MathListener to inherit all this functionality. You should examine the source code for one of the concrete classes derived from MathListener (MathActionListener is probably the simplest one) to see how it is written. You can use this as a starting point for your own implementation. If you do not make your class a subclass of MathListener, and instead choose instead to write your own event-handler code that calls into *Mathematica*, you *must* read "Writing Your Own Event Handler Code".

Bringing Java Windows to the Foreground

If you are creating a Java window with a *Mathematica* program, you probably want that window to pop up in front of the notebook the user is working in, so that its presence becomes apparent. You might expect that the `toFront()` method of Java's `Window` class is what you would use for this, but this does not work on the Macintosh, and it works slightly differently on different Java runtimes on Windows. As a result of these differences, it is difficult to write a *Mathematica* program that behaves identically on all platforms and all Java virtual machines with respect to making Java windows visible in front of all other windows the user might see.

As a result of these unfortunate differences, *J/Link* provides a *Mathematica* function, `JavaShow`, which performs the proper steps on all configurations. You should use `JavaShow[`*window*`]` in place of `window@setVisible[True]`, `window@show[]`, or `window@toFront[]`. You will see `JavaShow` used in all the example programs. The argument to `JavaShow` must be a Java object that is an instance of a class that can represent a top-level window. Specifically, it must be of class `java.awt.Window` or a subclass. This includes the AWT `Frame` and `Dialog` windows, and also the Swing classes used for top-level windows (`JFrame`, `JWindow`, and `JDialog`).

`JavaShow[`*windowObj*`]`	make the specified Java window visible and bring it in front of all other windows, including notebook windows

Bringing a Java window to the foreground.

Modal Windows

Here is an example of a simple "modal" window. The window contains a button and a text field. The text field starts out displaying the value 1, and each time the button is clicked the value is incremented. The `com.wolfram.jlink.MathFrame` class is used for the enclosing window. `MathFrame` is a simple extension to `java.awt.Frame` that calls `dispose()` on itself when its close box is clicked (the standard `Frame` class does nothing).

```
frm = JavaNew["com.wolfram.jlink.MathFrame"];

button = JavaNew["java.awt.Button"];
textField = JavaNew["java.awt.TextField"];

frm@setLayout[JavaNew["java.awt.GridLayout"]];
frm@add[button];
frm@add[textField];
frm@pack[];
JavaShow[frm];
```

At this point, you should see a small frame window with a button on the left and a text field on the right. Now label the button and set the starting text for the field.

```
button@setLabel["++"];
textField@setText["1"];
```

Now you want to add behavior to the button that causes it to increment the text field value. Buttons fire `ActionEvent`s, so you need an instance of `MathActionListener`.

```
buttonListener = JavaNew["com.wolfram.jlink.MathActionListener"];
```

It must be registered with the button by calling `addActionListener`.

```
button@addActionListener[buttonListener];
```

At this point, if you were to click the **++** button, the `actionPerformed()` method of your `MathActionListener` would be called (do not click the button yet!). You know from the `MathListener` table in the previous subsection that the `actionPerformed()` method will call a user-defined *Mathematica* function with two arguments: the `ActionEvent` object itself and the integer value that results from the event's `getActionCommand()` method.

You have not yet set the user-defined code to be called by the `actionPerformed()` method. That is done for all the `MathListener` classes with the `setHandler()` method. This method takes two strings, the first being the name of the method in the event-listener interface, and the second being the function you want called.

```
buttonListener@setHandler["actionPerformed", "buttonFunc"];
```

Now you need to define `buttonFunc`. It must be written to take two arguments, but in this example you are not interested in either argument.

```
buttonFunc[_, _] :=
 Module[{curText, newVal},
       curText = textField@getText[];
       newVal = ToExpression[curText] + 1;
       textField@setText[ToString[newVal]]
 ]
```

You are still not quite ready to try the button. If you click the button now, the Java user interface thread will hang because it will call into *Mathematica* trying to evaluate `buttonFunc` and wait for the result, but the result will never come because the kernel is not waiting for input to arrive on the Java link. What you need is a way to put the kernel into a state where it is continuously reading from the Java link. This is what makes the window "modal"—the kernel cannot do anything else until the window is closed. The function that implements this modal state is `DoModal`.

`DoModal[]`	put the kernel into a state where its attention is solely directed at the Java link
`EndModal[]`	what the Java program must call to make the `DoModal` function return, ending the modal state

Entering and exiting the modal state.

`DoModal` will not return until the Java program calls back into *Mathematica* to evaluate `EndModal[]`. While `DoModal` is executing, the kernel is ready to handle callbacks from Java—for example, from `MathListener` objects. The way to get the Java side to call `EndModal[]` is typically to use a `MathListener`. For example, if your window had **OK** and **Cancel** buttons, these should dismiss the window, so you would create `MathActionListener` objects and register them with these two buttons. These `MathActionListener` objects would be set to call `EndModal[]` in their `actionPerformed()` methods.

`DoModal` returns whatever the block of code that calls `EndModal[]` returns. You would typically use this return value to determine how the window was closed—for example, whether it was the **OK** or **Cancel** button. You could then take appropriate action. See "A Simple Modal Input Dialog" for an example of using the return value of `DoModal`.

In the present example, the only way to close the window is by clicking its close box. Clicking the close box fires a `windowClosing` event, so you use a `MathWindowListener` to receive notifications.

```
windowListener = JavaNew["com.wolfram.jlink.MathWindowListener"];
frm@addWindowListener[windowListener];
```

Now you assign the *Mathematica* function to be called when the close box is clicked. All you need it to do is call `EndModal[]`, so you can specify a pure function that ignores its arguments and does nothing but execute `EndModal[]`.

```
windowListener@setHandler["windowClosing", "EndModal[]&"];
```

The preceding few lines are a fine example of how to use a MathWindowListener to trigger a call to EndModal[] when a window's close box is clicked. You would use something similar to this, except with a MathActionListener, if you wanted to have an explicit **Close** button. In this example, though, there is an easier way. Mentioned earlier is that the MathFrame class is just a normal AWT Frame except that it calls dispose() on itself when its close box is clicked. Actually it has another useful property—it can also execute EndModal[] when its close box is clicked. Thus, if you use MathFrame as the top-level window class for your interfaces, you will not have to manually create a MathWindowListener to terminate the modal loop every time. To enable this behavior of MathFrame, you need to call its setModal method:

```
(***
 This is even easier than using the MathWindowListener above.
 We won't call it here, though, because we have already arranged
 for EndModal to be called, and bad things will happen if we try
 to call it twice.

frm@setModal[]

***)
```

You must not call setModal if you are not using DoModal. This is because after setModal has been called, the MathFrame will try to call into *Mathematica* when it is closed (to execute EndModal), and *Mathematica* needs to be in a state where it is ready for calls originating in Java. The same issue exists for any MathListener you create yourself.

Now that everything is ready, you can enter the modal state and use the window.

```
DoModal[]
```

When you are done playing with the window, click the close box in the frame, which will trigger a callback into *Mathematica* that calls EndModal[]. DoModal then returns, and the kernel is ready to be used from the front end. DoModal[] returns Null if you click the close box of a MathFrame.

Here is how the entire example looks when packaged into a single program. (The code for SimpleModal is also available as SimpleModal.nb in the JLink/Examples/Part1 directory.)

```
SimpleModal[] :=
    JavaBlock[
        Module[{frm, button, textField, windowListener,
                buttonListener, buttonFunc},

            (* Create the GUI components. *)
            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            button = JavaNew["java.awt.Button"];
            textField = JavaNew["java.awt.TextField"];

            (* Configure their properties. *)
            frm@setLayout[JavaNew["java.awt.GridLayout"]];
            frm@add[button];
            frm@add[textField];
            button@setLabel["++"];
            textField@setText["1"];
            frm@pack[];

            (* Create the listener and set its handler function. *)
            buttonListener =
                JavaNew["com.wolfram.jlink.MathActionListener"];
            buttonListener@setHandler["actionPerformed", ToString[buttonFunc]];
            button@addActionListener[buttonListener];

            (* Define buttonFunc. *)
            buttonFunc[_, _] :=
                JavaBlock[
                    Module[{curText, newVal},
                        curText = textField@getText[];
                        newVal = ToExpression[curText] + 1;
                        textField@setText[ToString[newVal]]
                    ]
                ];

            (* Make the window visible and bring it in front of any
               notebook windows. *)
            JavaShow[frm];

            (* Tell the frame to end the modal loop when it is closed. *)
            frm@setModal[];

            (* Enter the modal loop. *)
            DoModal[];
        ]
]
```

Remember that `DoModal` will not return until the Java side calls `EndModal`. You have to be a little careful when you call `DoModal` that you have already established a way for the Java side to trigger a call to `EndModal`. As explained earlier, you will typically have done this by using a `MathFrame` as the frame window and calling its `setModal` method, or by creating and registering a `MathListener` of your own that will call `EndModal` in response to a user action (such as clicking an **OK** or **Cancel** button). Once `DoModal` has begun, the kernel is not responsive to the front end and thus it is too late to set anything up. If you call `DoModal` and realize that for some reason you cannot end it from Java, you can abort it from the front end by selecting **Evaluation ▸ Interrupt Evaluation** in the menu, and then in the resulting dialog, clicking the button labeled **Abort**.

There is one subtlety you might notice in the code for SimpleModal that is not directly related to *J/Link*. In the line that calls buttonListener@setHandler, you pass the name of the button function not as the literal string "buttonFunc", but as ToString[buttonFunc]. This is because buttonFunc is a local name in a Module, and thus its real name is not buttonFunc, but something like buttonFunc$42. To make sure you capture its true run-time name, you call ToString on the symbolic name. You could avoid this by simply not making the name buttonFunc local to the Module, but the way you have done it automatically cleans up the definition for buttonFunc when the Module finishes.

MathFrame and MathJFrame

You encountered the MathFrame class in this section, which is a useful top-level window class for *J/Link* programmers because it has three special properties. You have already encountered two of them: it calls dispose() on itself when it is closed, and it has the setModal() method, which gives it easy support for use with DoModal. The third property is that it has an onClose() method that you can use to specify *Mathematica* code that will be executed when the window is closed. The onClose() method is used in the Palette example in "Sharing the Front End: Palette-Type Buttons". *J/Link* also has a MathJFrame class, which is a subclass of the Swing JFrame class, and it also has these three special properties. Programmers who want to create interfaces with Swing components instead of AWT ones can use MathJFrame as their top-level window class.

Modeless Windows: Sharing the Kernel with Java

The previous subsection demonstrated how to write *J/Link* programs that display Java windows and then how to use the DoModal function to cause the kernel to wait until the window is closed. During the time that DoModal is running, the kernel is able to receive and process requests for computations that originate from the Java side. The word "modal" is used in this context to refer to the fact that the kernel is busy servicing the Java link, and thus the notebook front end cannot use the kernel until DoModal returns.

This arrangement works fine for many types of Java windows, and it is required for those that return a result to *Mathematica*, because the kernel cannot sensibly proceed until the window is dismissed. Unfortunately, it is too restrictive for a large class of user interface elements. Con-

sider trying to duplicate the general concept of a front end palette window in Java. You want to have a window of buttons that, when clicked, cause some computation to occur in *Mathematica*. Like a front end palette window, you want this window to be created and remain visible and active indefinitely. It would not be of much use if every time you wanted to click one of the buttons you had first to execute `DoModal[]` (and you would also have to arrange for each button to call `EndModal[]` as part of the computation it triggers). You want to be able to go back and forth between notebook windows in the front end and our Java window without needing manually to switch the kernel into and out of some special state each time.

What is needed is a way for the kernel to automatically pay attention to input arriving from the Java link in addition to the notebook front end link. What you really have here is two front ends vying for the kernel's attention. *J/Link* solves this problem by introducing a simple way in which the kernel can be put into a state where it is simultaneously listening for input on any number of links. The function that accomplishes this is `ShareKernel`.

Important Note: In *Mathematica* 5.1 and later, the kernel is always shared with Java. This means that the functions `ShareKernel` and `UnshareKernel` are not necessary and, in fact, do nothing at all. If you are writing program that only need to run in *Mathematica* 5.1 and later, you never need to call `ShareKernel` or `UnshareKernel` (`ShareFrontEnd` and `UnshareFrontEnd` are still useful, however). If your programs need to work on all versions of *Mathematica*, then you will need to use `ShareKernel` and `UnshareKernel` as described next.

`ShareKernel[]`	begin sharing the kernel with Java
`ShareKernel[`*link*`]`	begin sharing the kernel with *link*
`UnshareKernel[`*id*`]`	unregisters the request for sharing (that is, the call to `ShareKernel`) that returned *id*; kernel sharing will not be turned off unless no other requests are outstanding
`UnshareKernel[`*link*`]`	end sharing of the kernel with *link*
`UnshareKernel[]`	end sharing of the kernel with Java
`KernelSharedQ[]`	`True` if the kernel is currently being shared; `False` otherwise
`SharingLinks[]`	a list of the links currently sharing the kernel

Sharing the kernel.

ShareKernel takes a LinkObject as an argument and initiates sharing of the kernel between that link and the current $ParentLink (typically, the notebook front end). If you call ShareKernel with no arguments, it assumes you mean the link to Java. Most users will call it with no arguments.

```
In[]:= ShareKernel[];

In[]:= 2 + 2
Out[]= 4
```

Note that while the kernel is being shared, the input prompt has "(sharing)" prepended to it. The string that is prepended is specified by the SharingPrompt option to ShareKernel.

Sharing is transparent to the user. Other than the changed input prompt, there is nothing to suggest that anything different is going on. Input sent from either the front end or a Java program to the kernel will be evaluated and the result sent back to the program that sent the input. Each link is the kernel's $ParentLink during the time that the kernel is computing input that arrived from that link. In other words, ShareKernel takes care of shuffling the $ParentLink value back and forth between links as input arrives on each.

It is safe to call ShareKernel if the kernel is already being shared. This means that programs you write can call it without your having to worry that a user might already have initiated sharing. When you are finished with the need to share the kernel with Java, you can call UnshareKernel. This restores the kernel to its normal mode of operation, paying attention only to the front end.

```
In[]:= UnshareKernel[]
```

When called with no arguments, UnshareKernel shuts down sharing. This is not a desirable thing in most cases, because it might be that some other Java-based program is running that requires sharing. If you are writing code for others to use, you certainly cannot shut down sharing on your users just because your code is done with it. To solve this problem, ShareKernel returns a token (it is just an integer, but you should not be concerned with its representation) that reflects a request for sharing functionality. In other words, calling ShareKernel registers a request for sharing, turns it on if it is not on already, and returns a token that represents that particular request. When you call UnshareKernel, you pass it the token to "unregister" that particular request for sharing. Only if there are no other outstanding requests will sharing actually be turned off.

A quirk of ShareKernel is that you cannot call ShareKernel and UnshareKernel in the same cell. Doing so will cause the kernel to hang. Of course, there is no reason to ever do this, as kernel sharing is only relevant when it spans multiple evaluations (more precisely, the evaluation of multiple cells). There would be no point to turning sharing on and off within the scope of a single computation.

An example of a nontrivial user interface that uses ShareKernel is presented in "Real-Time Algebra: A Mini-Application".

Sharing the Front End

One goal of *J/Link* was to have Java user interface elements be as close as possible to first-class citizens of the notebook front end environment, in the way that notebooks and palettes are. The ability to share the kernel mimics one important aspect of this citizenship, hiding the fact that the Java runtime is a separate program and the kernel is normally only waiting for input from the front end.

There is one more important thing that palettes can do that would be nice to do from Java, and that is interact with the front end. You can create a palette button that, when clicked, evaluates the code Print["hello"]. You can do this easily with *J/Link* also, but with one big difference: when you click the palette button, hello appears in the active notebook, but when you click the Java button, the "hello" gets sent back to the Java program (which is, after all, the kernel's $ParentLink at that moment). Even if you persuaded the kernel to write the TextPacket that contains "hello" to the front end link instead of the Java link, nothing useful would happen because the front end is not paying attention to the kernel link when the front end is not waiting for the result of a computation. Poking some output at the front end while it is idle simply will not work.

J/Link provides the ShareFrontEnd function as the solution to this problem. ShareFrontEnd[] causes Print output and graphics generated by a Java user-interface element to appear in the front end. It also lets the Java side call *Mathematica* functions that manipulate elements of notebooks and have them work properly in the front end (for example, NotebookRead, NotebookWrite, SelectionEvaluate, and so on). While sharing is on, the front end behaves normally, and you can continue to use it for editing, calculations, or whatever. The sharing is transparent.

`ShareFrontEnd[]`	begin sharing the front end with Java
`UnshareFrontEnd[`*id*`]`	unregisters the request for sharing (that is, the call to `ShareFrontEnd`) that returned *id*; front end sharing will not be turned off unless no other requests are outstanding
`UnshareFrontEnd[]`	end sharing of the front end with Java
`FrontEndSharedQ[]`	`True` if the front end is currently being shared with Java; `False` otherwise

Sharing the notebook front end.

`ShareFrontEnd` currently does not work with a remote kernel; the same machine must be running the kernel and the front end.

`ShareFrontEnd` is as close as you currently can come to having Java user interfaces hosted directly by the notebook front end itself, as if they were special types of notebook windows. This type of tight integration might be possible in the future.

Note that `Print` output, graphics, and messages generated by a *modal* Java window will appear in the front end without needing to call `ShareFrontEnd`. This is because `$ParentLink` remains the front end link during `DoModal` (these "side effect" packets always get sent to `$ParentLink`), and also because the front end is able to handle various packets arriving from the kernel because the front end *is* in the middle of a computation—it is waiting for the result of the code that called `DoModal`. `ShareFrontEnd` is a way to restore a feature that was lost when you gained the ability to create *modeless* interfaces via `ShareKernel`. That is how to think of `ShareFrontEnd`—as a step beyond `ShareKernel` that allows side effect output generated by computations triggered in Java to appear in the notebook front end. `ShareFrontEnd` is particularly useful when developing code that needs to use `ShareKernel`, even if the code does not need the extra functionality of `ShareFrontEnd`. This is because *Mathematica* error messages generated by computations triggered by Java events get lost with `ShareKernel`. The messages will show up in the front end if front end sharing is turned on.

When you are done with the need to share the front end, call `UnshareFrontEnd`. Like the `ShareKernel`/`UnshareKernel` pair of functions, `ShareFrontEnd` returns a token that you should pass to `UnshareFrontEnd` to unregister the request for front end sharing. Only when all calls to `ShareFrontEnd` have been unregistered by calls to `UnshareFrontEnd` will front end sharing be turned off. You can force front end sharing to be shut down immediately by calling

UnshareFrontEnd with no arguments, but although this is convenient when you are developing code of your own, it should never be called in code that is intended for others to use. Just because your code is done with front end sharing does not mean that your users are done with it. Instead, save the token returned from ShareFrontEnd and pass it to UnshareFrontEnd.

ShareFrontEnd requires that the kernel be shared, so it calls ShareKernel internally. Calling UnshareKernel with no arguments forces kernel sharing to stop immediately, and this turns off front end sharing as well. Thus, you can use UnshareKernel[] as a quick shortcut to immediately shut down all sharing.

An example of some simple palette-type buttons that use ShareFrontEnd is presented in "Sharing the Front End: Palette-Type Buttons".

An important use for ShareFrontEnd is to allow a popup Java user interface to display graphics containing typeset expressions. When the kernel is asked to produce a graphic containing typeset expressions, say a plot with PlotLabel -> Sqrt[z], it crunches out PostScript for the plot itself, but when it comes time to produce PostScript for the typeset label, it cannot do this. Instead, it sends a special request back to the front end, asking it for the PostScript representation. Because dealing with typeset expressions is a skill possessed only by the notebook front end, when any other interface is driving the kernel, the interface must be careful to instruct the kernel to not attempt to typeset anything in a graphic (ShareKernel handles this automatically for you). This works fine, but you lose the ability to get pictures of typeset expressions in your Java interface.

ShareFrontEnd does two things to overcome this limitation: it fools the kernel into thinking that the Java runtime is a notebook front end and, therefore, capable of handling the special "convert to PostScript" requests; and it gives Java the ability to make good on this promise by forwarding the requests to the front end. "GraphicsDlg: Graphics and Typeset Output in a Window" describes an example of a Java dialog box that displays typeset expressions using ShareFrontEnd.

Summary of Modal and Modeless Operation

The previous discussion of modal and modeless operation, `ShareKernel`, and `ShareFrontEnd` may have seemed complex. In fact, the principles and uses of these techniques are simple. This will become clear upon seeing some more examples. Many of the example programs in "Example Programs" use `ShareKernel` or `ShareFrontEnd`. The important thing is to understand the capabilities they provide so that you can begin to see how to use them in your own programs.

If you want your user-interface element (typically a window) to tie up the kernel until the user dismisses it, then you will use the `setModal/DoModal/EndModal` suite. Because the internal workings of the modal state are simpler than the modeless state, you should use this style unless your program needs the features of a modeless window. You will always want to use this type of window if you need to return a result to a running *Mathematica* program, such as if you are creating a dialog box into which the user will enter values and then click **OK**. "A Simple Modal Input Dialog" gives an example of this type of dialog.

If you want your window to remain visible and active while the user returns to work in the front end, you must run your window in a "modeless" fashion. This requires calling `ShareKernel` to put the kernel into a state where it is simultaneously receptive to input arriving from either the notebook front end or Java. At this point the kernel is dividing its attention between two independent and essentially equivalent front ends. One drawback (or feature, depending on your point of view) of this state is that all side effect output like `Print` output, messages, or plots triggered by Java code is sent to Java instead of the front end (and the standard Java `MathListener` classes just throw all this output away). Thus, you could not create a button that prints something in a notebook window when it is clicked, like you can with a palette button in the front end. If you want to give your Java program the ability to interact with the front end the way that notebook and palette windows themselves can, you must instead use `ShareFrontEnd`, which you can think of as an extension to `ShareKernel`.

A very common mistake is to create a Java window, wire up a `MathListener` class that calls back to *Mathematica* on some event, and then trigger the event before you have called `DoModal` or `ShareKernel`. This will cause the Java user interface thread to hang. A symptom that the UI thread is hanging is that the controls in your Java window are visually unresponsive (for example, buttons will not appear to depress when you click them). If you do inadvertently get into this state, you can just call `ShareKernel` to allow the queued-up call(s) from Java to proceed.

"Manual" Interfaces: The ServiceJava Function

In addition to the modal and modeless types of interfaces just discussed, there is another type that in some ways is intermediate. Consider the following scenario. You want to create a *Mathematica* program that puts up a Java window and displays something in it that changes over the course of the program. So far, this sounds like an example of a "non-interactive" interface, which was discussed way back at the beginning of this section, the progress bar example being a classic case. Now, though, you want to add some interactivity to the window, meaning that you want user actions in the window to trigger calls into *Mathematica*. Keeping with the progress bar example, say you want to add an **Abort** button that stops the program. How do you manage to get the kernel's attention directed at the Java side so that Java events can trigger calls to *Mathematica*?

The modal type of interface will not work, because in the modal state the kernel is executing `DoModal`, not your computation—the kernel is doing nothing but paying attention to Java. The modeless type of interface will not work either, because the modeless technique causes the kernel to pay attention to the front end and Java alternately, letting each perform a full computation in turn. There is no sharing within the context of a single computation.

The obvious answer is the there needs to be a function that allows the kernel to service a single computation arriving from Java, if there is one waiting. That function is `ServiceJava`. Calling `ServiceJava` in a program will cause the kernel to accept one request for a computation from the Java side. It performs the computation and then returns control to your program. If there is no request waiting, `ServiceJava` returns immediately.

Here is some pseudocode showing the structure of a program that displays a progress bar with an **Abort** button and periodically calls `ServiceJava` to handle user clicks on that button, stopping the computation if requested.

```
    ... create progress bar ...
    progressBar@addActionListener[
        JavaNew["com.wolfram.jlink.MathActionListener", "(userCancelled =
True)&"]
    ];
    JavaShow[progressBar];
    While[i < 100 && !userCancelled,
        ... compute one iteration ...
        ... update progress bar ...
        ServiceJava[];
        i++
    ];
    ... destroy progress bar ...
```

You might recognize that `ServiceJava` is closely related to `DoModal`, and although this is not the actual implementation, you can think of `DoModal` as being written in terms of `ServiceJava` as follows:

```
(* Not the actual implementation of DoModal, but the principle is correct.
*)
DoModal[] :=
    While[!endModal,
        ServiceJava[]
    ]
```

Seen in this way, `DoModal` is a special case of the use of `ServiceJava`, where *Mathematica* is doing nothing but servicing requests from Java. Sometimes you need something else to be going on in *Mathematica*, but still need to be able to handle requests arriving from Java. That is when you call `ServiceJava` yourself. Like `DoModal`, there is no shifting of `$ParentLink` when `ServiceJava` is called. Thus, side-effect output like graphics, messages, and `Print` output triggered by Java computations appear in the notebook, just as if they were hard-coded into the *Mathematica* program that called `ServiceJava`.

The BouncingBalls example program presented in "BouncingBalls: Drawing in a Window" uses `ServiceJava`.

Using a GUI Builder

The preceding discussion on modal and modeless interfaces featured examples that were created entirely with *Mathematica* code. For complex user interfaces, you might find it more convenient to lay out your windows and wire up events with a drag-and-drop GUI builder like the ones present in most commercial Java development environments. You are free to write as much or as little of the code for your interface in native Java. If you want events in your GUI to trigger calls into *Mathematica*, then you can use any of the `MathListener` classes from Java code just as they are used from *Mathematica* code. Alternatively, you could write your own Java code that calls into *Mathematica* at appropriate times. See the section "Writing Your Own Installable Java Classes" for information about how to write Java code that calls back into *Mathematica*. "GraphicsDlg: Graphics and Typeset Output in a Window" gives a simple example of a dialog box that was created with a GUI builder and is then invoked and controlled by *Mathematica* code.

Drawing and Displaying *Mathematica* Images in Java Windows

The MathCanvas and MathGraphicsJPanel classes

J/Link makes it easy to draw into Java windows from *Mathematica*, and also display *Mathematica* graphics and typeset expressions. The `MathCanvas` and `MathGraphicsJPanel` classes are provided for this purpose. You can use these classes in pure Java programs that use the *Mathematica* kernel, as described in "Writing Java Programs that use *Mathematica*", but it is also handy for Java windows that are created and scripted from *Mathematica*. Note that the `MathGraphicsJPanel` class is new in *J/Link* 2.0.

`MathCanvas` is a subclass of the AWT `Canvas` class, and `MathGraphicsJPanel` is a subclass of the Swing `JPanel` class. In terms of their special added *Mathematica* graphics capabilities, they are identical. These classes provide two ways to supply the image to be displayed. The first way is by providing a fragment of *Mathematica* code whose output will be displayed. The output can either be a graphics object, or a nongraphics expression that will be typeset. This makes it

trivial to display *Mathematica* graphics or typeset expressions in a Java window. The second way to control the display is to provide a Java `Image` object that will be painted. This `Image` will typically be created by *Mathematica* code, such as code that creates a bitmap out of raw *Mathematica* data, or code that draws something using calls to Java's graphics routines.

Because `MathCanvas` and `MathGraphicsJPanel` are Java classes and can be used from Java programs as well as *Mathematica* programs, there is full JavaDoc format documentation for them in the JLink/Documentation/JavaDoc directory. You can browse that documentation for more details.

Showing Mathematica Graphics and Typeset Expressions

Here is a simple example of displaying a window that shows a *Mathematica* plot. This example uses `MathCanvas`, but the relevant parts would look the same if you used `MathGraphicsJPanel`. You will be using this window throughout this section, so do not close it if you are evaluating the code as you read this section.

```
frame = JavaNew["com.wolfram.jlink.MathFrame"];
frame@setLayout[JavaNew["java.awt.BorderLayout"]];
mathCanvas = JavaNew["com.wolfram.jlink.MathCanvas"];
frame@add["Center", mathCanvas];
frame@setSize[400, 400];
frame@layout[];
mathCanvas@setMathCommand["Plot[x, {x,0,1}]"];
JavaShow[frame];
```

As you can see, it is as simple as calling the canvas' `setMathCommand()` method. The argument to `setMathCommand()` is a string giving the code to be evaluated. This code must *return* a graphics expression, not just cause one to be produced. For example, `setMathCommand["Plot[x,{x,0,1}];"]` will not work because the trailing semicolon causes the expression to evaluate to `Null`. The image is automatically rendered at the correct size, and centered in the canvas if the actual image size produced by *Mathematica* does not completely fill the requested area (as is often the case with typeset output).

Calling `setMathCommand()` again resets the image.

```
mathCanvas@setMathCommand["Plot3D[Sin[x Cos[y]], {x,0,2Pi}, {y,0,2Pi}]"];
```

If the plotting command depends on variables in your *Mathematica* session, you can call recompute() to cause the graphic to be recomputed and rendered. For example, this displays a slow animation in the window.

```
n = 1.0;
mathCanvas@setMathCommand["Plot3D[Sin[n x Cos[y]], {x,0,2Pi}, {y,0,2Pi}]"];
Do[n += 0.1; mathCanvas@recompute[]; Pause[1], {10}]
```

Because you supply the expression as a string, remember to escape any quote marks inside the string with a backslash.

```
mathCanvas@setMathCommand["Plot[x, {x,0,1}, PlotLabel->\"This is a plot\"]"];
```

A MathCanvas can also display typeset expressions. The default behavior of MathCanvas is to expect that the expression supplied in setMathCommand() will evaluate to a graphics object, which should be rendered. To get it to instead typeset the return value, call the setImageType() method, supplying the constant TYPESET.

```
mathCanvas@setImageType[MathCanvas`TYPESET];
mathCanvas@setMathCommand["Integrate[Sqrt[x] Sqrt[1+x], x]"];
```

To switch back to displaying graphics, call mathCanvas@setImageType[MathCanvas`GRAPHICS]. The default format for typeset output is StandardForm. To switch to TraditionalForm, use the setUsesTraditionalForm() method. You call recompute() here because changing the output type does not force the image to be redrawn.

```
mathCanvas@setUsesTraditionalForm[True];
mathCanvas@recompute[];
```

Graphics are rendered using *Mathematica*'s Display command, which is fast and does not require the notebook front end to be running. For higher quality, though, particularly for 3D graphics, an alternative method is available that uses the front end for rendering services. You can switch to using this technique by calling the setUsesFE() method.

```
(* First, change back to graphics mode from typeset mode. *)
mathCanvas@setImageType[MathCanvas`GRAPHICS];

mathCanvas@setUsesFE[True];
mathCanvas@setMathCommand["Plot3D[Sin[x Cos[y]], {x,0,2Pi}, {y,0,2Pi}]"];
```

You might want to compare the resulting plot with setUsesFE[True] and setUsesFE[False].

An important point about using the front end for rendering is that when the computation to produce the image is performed, the front end must be in a state where it is receptive to requests for services from the kernel. There are two times when this is the case: either a cell in the front end is currently evaluating (as will be the case when you are calling `setMathCommand()` or `recompute()` from a *Mathematica* program), or `ShareFrontEnd` has been called. Looking at it from the other direction, the only time it will not work is if `ShareKernel` is in use, but not `ShareFrontEnd`, and the computation is triggered by an event in Java. The rule is that if you want to involve the front end for rendering, and you want to call `setMathCommand()` or `recompute()` from Java in response to a user action in a modeless interface, you need to use `ShareFrontEnd`; `ShareKernel` is not enough. Modal and modeless interfaces and `ShareFrontEnd` are discussed in the section "Creating Windows and Other User Interface Elements".

Drawing Using Java's Graphics Functions

You saw that the `setMathCommand()` method of the `MathCanvas` and `MathGraphicsJPanel` classes lets you supply a *Mathematica* expression whose output is to be displayed. You can also use a `MathCanvas` or `MathGraphicsJPanel` to display a Java `Image` by using the `setImage()` method instead of `setMathCommand()`.

Now look at a simple example of drawing into a Java window from *Mathematica*. You will continue to use the same window and `MathCanvas` you have been working with. If this program used a `MathGraphicsJPanel` instead, the portions of the code related to drawing would look exactly the same. To draw into the `MathCanvas`, you create an offscreen image of the same dimensions, get a graphics context for drawing onto it, draw, and then use the `setImage()` method of `MathCanvas` to cause the offscreen image to be displayed. Drawing into an offscreen image and then blitting it to the screen is a standard technique for flicker-free drawing.

```
offscreen = mathCanvas@createImage[mathCanvas@getSize[]@width,
                                   mathCanvas@getSize[]@height];
g = offscreen@getGraphics[];
g@drawRect[100, 100, 200, 150];
mathCanvas@setImage[offscreen];
```

Programs that want to draw manually into a Java window from *Mathematica* will generally all have this same structure. It takes just a few more lines of code to turn our `MathCanvas` into a scribble program. Here is the complete program (this code is also provided as the file Scribble.nb in the JLink/Examples/Part1 directory).

```
Scribble[] :=
    JavaBlock[
        Module[{frame, mathCanvas, offscreen, g, mml, pts},
            frame = JavaNew["com.wolfram.jlink.MathFrame"];
            frame@setLayout[JavaNew["java.awt.BorderLayout"]];
            mathCanvas = JavaNew["com.wolfram.jlink.MathCanvas"];
            frame@add["Center", mathCanvas];
            frame@setSize[400, 400];
            frame@layout[];
            JavaShow[frame];
            (* Now create the offscreen image and the graphics context
               for drawing into it.
            *)
            offscreen = mathCanvas@createImage[mathCanvas@getSize[]@width,
                                               mathCanvas@getSize[]@height];
            g = offscreen@getGraphics[];
            (* Now create the MathMouseMotionListener that will do the drawing
               and set its mouseDragged event handler callback.
            *)
            mml = JavaNew["com.wolfram.jlink.MathMouseMotionListener"];
            mml@setHandler["mouseDragged", "mouseDraggedFunc"];
            mathCanvas@addMouseMotionListener[mml];
            mouseDraggedFunc[_, x_, y_, _] :=
                (g@drawLine[pts[[-1, 1]], pts[[-1, 2]], x, y];
                 mathCanvas@setImage[offscreen];
                 mathCanvas@repaintNow[];
                 AppendTo[pts, {x,y}];);
            (* Initialize the pts list and run the program modally. *)
            pts = {{0,0}};
            frame@setModal[];
            DoModal[];
            pts
        ]
    ]
```

Run the program, then click and drag the mouse to draw in the window. Close the window to end the program and the `Scribble` function will return the list of points drawn.

```
pts = Scribble[];
```

If you examine the list of points returned, you will see that they are based on Java's coordinate system, which has (0, 0) in the upper left. If you want to plot the points in a *Mathematica* graphic, you have to invert the *y* values. This is demonstrated in the Scribble.nb example notebook.

There is one new MathCanvas method demonstrated in this program, repaintNow(). In a computation-intensive program like this, where events are being fired on the user interface thread very quickly, and the handlers for these events take a nontrivial amount of time to execute, Java will sometimes delay repainting the window. The drawing becomes very chunky, with no visual effect for a while and then suddenly all the lines drawn in the last few seconds will appear. Even calling the standard repaint() method after every new point will not ensure that the window is updated in a timely manner. To solve this problem, the repaintNow() method is provided, which forces an immediate redraw of the canvas. If your program relies on smooth visual feedback from user events that fire rapidly, you should call repaintNow() also, even if it does not seem necessary on your system. There can be very significant differences between different platforms and different Java runtimes on the responsiveness of the screen updating mechanism.

The ability to draw in response to events in a MathCanvas or MathGraphicsJPanel opens up the possibility for some impressive interactive demonstrations, tutorials, and so on. Two of the larger example programs provided draw into a MathCanvas from *Mathematica:* BouncingBalls (in the section "BouncingBalls: Drawing in a Window") and Spirograph (in the section "Spirograph").

Bitmaps

You have seen how to draw into a MathCanvas or MathGraphicsJPanel by using an offscreen image. Another type of image that you can create with *Mathematica* code and display using setImage() is a bitmap. In this example you will create an indexed-color bitmap out of *Mathematica* data and display it. You will use an 8-bit color table, meaning that every data point in the image will be treated as an index into a 256-element list of colors. You could use a larger color table if desired.

You closed the frame window in the Scribble example, so you must first create a new frame and canvas for the bitmap.

```
frame = JavaNew["com.wolfram.jlink.MathFrame"];
frame@setLayout[JavaNew["java.awt.BorderLayout"]];
mathCanvas = JavaNew["com.wolfram.jlink.MathCanvas"];
frame@add["Center", mathCanvas];
frame@setSize[450, 450];
frame@layout[];
JavaShow[frame];
```

Here is the color table. It is an array of {r,g,b} triplets, with each color component being in the range 0..255. In this example, colors with low indices are mostly blue, and ones with high indices are mostly red.

```
colors = Table[{i, 0, 255 - i}, {i, 0, 255}];
```

The data is a 400×400 matrix of integers in the range 0..255 (because they are indices into the 256-element color table). In a real application, this data might be read from a file or computed in some more sophisticated way. If the range of numbers in the data did not span 0..255, you would have to scale it into that range, or a larger range if you wanted to use a deeper color table.

```
data = Table[Round[255 (0.5 + Sin[x]Cos[y]/2)],
            {x, Pi/100., 4Pi, Pi/100.}, {y, Pi/100., 4Pi, Pi/100.}];
```

Here you create the Java objects that represent the color model and bitmap. You can read the standard Java documentation on these classes for more information.

```
colorModel = JavaNew["java.awt.image.IndexColorModel", 8, 256,
                              Flatten[colors], 0, False];
bitmap = JavaNew["java.awt.image.MemoryImageSource", 400, 400,
                           colorModel, Flatten[data], 0, 400];
```

Now create an `Image` out of the bitmap and display it.

```
image = frame@getToolkit[]@createImage[bitmap];
mathCanvas@setImage[image];
```

The Java Console Window

J/Link provides a convenient means to display the Java "console" window. Any output written to the standard `System.out` and `System.err` streams will be directed to this window. If you are calling Java code that writes diagnostic information to `System.out` or `System.err`, then you can see this output while your program runs. Like most *J/Link* features, the console window can be used easily from either *Mathematica* or Java programs (its use from Java code is described in "Writing Java Programs that use *Mathematica*"). To use it from *Mathematica*, call the `ShowJavaConsole` function.

`ShowJavaConsole[]`	display the Java console window and begin capturing output written to `System.out` and `System.err`
`ShowJavaConsole["stream"]`	display the Java console window and begin capturing output written to the specified stream, which should be "*stdout*" for `System.out` or "*stderr*" for `System.err`
`ShowJavaConsole[None]`	stop all capturing of output

Showing the console window.

```
ShowJavaConsole[]
```

```
«JavaObject[com.wolfram.jlink.ui.ConsoleWindow] »
```

Capturing of output only begins when you call `ShowJavaConsole`, so when the window first appears it will not have any content that might have been previously written to `System.out` or `System.err`. You will also note that the *J/Link* console window displays version information about the *J/Link* Java component and the Java runtime itself. Calling `ShowJavaConsole` when the window is already open will cause it to come to the foreground.

To demonstrate, you can write some output from *Mathematica*. If you executed the `ShowJavaConsole[]` given earlier, then you will see "Hello from Java" printed in the window.

```
LoadJavaClass["java.lang.System"];
System`out@println["Hello from Java"]
```

Although it is convenient to demonstrate writing to the window using *Mathematica* code like this, this is typically done from Java code instead. Actually, there is one common circumstance where it is quite useful to use the Java console window for diagnostic output written from *Mathematica* code. This is the case where you have a "modeless" Java user interface (as described in the section "Creating Windows and Other User Interface Elements") and you have not used the `ShareFrontEnd` function. Recall that in this circumstance, output from calls to `Print` in *Mathematica* will not appear in the notebook front end. If you write to `System.out` instead, as in the example, then you will always be able to see the output. You might want to do this in other circumstances just to avoid cluttering up your notebook with debugging output.

Using JavaBeans

JavaBeans is Java's component architecture. Beans are reusable components that can be manipulated visually in a builder tool. At the code level, a Bean is essentially just a normal Java class that conforms to a particular design pattern with respect to how its methods are named and how it supports events and persistence.

JavaBeans has not been mentioned up to this point because there really is not anything special to be said. Beans are just Java classes, and they can be used and called like any other classes. It is probably the case that many Java classes you use from *Mathematica* will be Beans, whether they advertise themselves to be or not. This is especially true for user interface components.

Beans are typically designed to be used in a visual builder tool, where the programmer is not writing code and calling named methods directly. Instead, a Bean exposes "properties" to the builder tool, which can be examined and set using a property editor window. In a typical simple example, a Bean might have methods named `setColor` and `getColor`, and by virtue of this it would be said to have a property named "color". A property editor would have a line showing the name "color" and an edit field where you could type in a color. It might even have a fancy editor that puts up a color picker window to let you visually select a desired color.

For the purposes of a visual builder tool or other type of automated manipulation, beans try to hide the low-level details of actual method names. If you want to call methods in a Bean class from *Mathematica* code, you call them by name in the usual way, without any consideration of the "Bean-ness" of the class.

Note that it would be quite possible to add *Mathematica* functions to *J/Link* that would provide explicit support for Bean properties. For example, a function `BeanSetProperty` could be written that would take a Bean object, a property name as a string, and the value to set the property to. Then, instead of writing what is currently required:

```
bean@setColor[Color`green]
```

you could write:

```
BeanSetProperty[bean, "color", Color`green]
```

The BeanSetProperty function lets you write code that manipulates nebulous things called properties instead of calling specific methods in the Bean class. If you do not see any particular advantage in the BeanSetProperty style, then you know why there is no special Bean support along these lines in *J/Link*. The advantages of working with properties versus directly calling methods accrues only when you are using a builder tool and not actually writing code by hand.

If you are interested, here are simplistic implementations of BeanSetProperty and BeanGetProperty:

```
BeanSetProperty[bean_?JavaObjectQ, propName_String, val_] :=
    Module[{methName = "set" <> ToUpperCase[StringTake[propName, 1]] <>
            StringDrop[propName, 1]},
        Through[(bean @@ ToHeldExpression[methName])[val]]
    ]

BeanGetProperty[bean_?JavaObjectQ, propName_String] :=
    Module[{methName = "get" <> ToUpperCase[StringTake[propName, 1]] <>
            StringDrop[propName, 1]},
        Through[(bean @@ ToHeldExpression[methName])[]]
    ]
```

To make use of events that a JavaBean fires, you can use one of the standard MathListener classes, as described in the section "Creating Windows and Other User Interface Elements". JavaBeans often fire PropertyChangeEvents, and you can arrange for *Mathematica* code to be executed in response to these events by using a MathPropertyChangeListener or a MathVetoableChangeListener.

Hosting Applets

J/Link gives you the ability to run most applets in their own window directly from *Mathematica*. Although this may seem immensely useful, given the vast number of applets that have been created, most applets do not export any useful public methods. They are generally standalone pieces of functionality, and thus they benefit little from the scriptability that *J/Link* provides. Still, there are many applets that may be useful to launch from a *Mathematica* program.

Note that this section is not about writing applets that use the *Mathematica* kernel. That topic is covered in "Writing Applets".

AppletViewer["*applet class*"]	runs the named applet class in its own window. The default width and height are 300 pixels
AppletViewer["*applet class*",*params*]	runs the named applet class in its own window, supplying it the given parameters, which is a list of "name=value" specifications like those used in an HTML page

Running applets.

J/Link includes an AppletViewer function for running applets. This function takes care of all the steps of creating the applet instance, providing a frame window to hold it, and starting it running. The first argument to AppletViewer is the fully qualified name of the applet class. The second argument is an optional list of parameters in "name=value" format, corresponding to the parameters supplied to an applet in an HTML page that hosts it. For example, if the <applet> tag in a web page that hosts an applet looks like this:

```
<applet code="SomeApplet.class" width=400 height=300>
     <param name=foo value=bar>
</applet>
```

you would call AppletViewer like this:

```
AppletViewer["SomeApplet", {"width=400", "height=300", "foo=bar"}];
```

You will typically supply at least "WIDTH=" and "HEIGHT=" specifications to control the width and height of the applet window. If you do not specify these parameters, the default width and height are 300 pixels.

An excellent example of an applet that is useful to *Mathematica* users is LiveGraphics3D, written by Martin Kraus. LiveGraphics3D is an interactive viewer for *Mathematica* 3D graphics. It gives you the ability to rotate and zoom images, view them in stereo, and more. If you want to try the following example, you will need to get the LiveGraphics3D materials, available from http://wwwvis.informatik.uni-stuttgart.de/~kraus/LiveGraphics3D/. Make sure you put live.jar onto your CLASSPATH before trying that example, or use the AddToClassPath feature of *J/Link* to make it available.

First, load the PolyhedronOperations` package and create the graphic to display. The LiveGraphics3D documentation gives a more general-purpose function for turning a *Mathematica* graphics expression into appropriate input for the LiveGraphics3D applet but, for many examples, using ToString, InputForm, and N is sufficient.

```
<< PolyhedronOperations`
dodec = ToString[InputForm[
    N[Graphics3D[Stellate[Normal[PolyhedronData["Dodecahedron", "Faces"]]]]]]];
```

You specify the image to be displayed via the `INPUT` parameter, which takes a string giving the `InputForm` representation of the graphic.

```
AppletViewer["Live", {"INPUT=" <> dodec, "WIDTH=400", "HEIGHT=400"}];
```

The Live applet has a number of keyboard and mouse controls for manipulating the image. You can read about them in the LiveGraphics3D documentation. Try Alt+S to switch into a stereo view.

When you are done with an applet, just click the window's close box.

If the applet needs to refer to other files, you should be aware that `AppletViewer` sets the document base to be the directory specified by the `"user.dir"` Java system property. This will normally be *Mathematica*'s current directory (given by `Directory[]`) at the time that `InstallJava` was called.

Most applets expose no public methods useful for controlling from *Mathematica*, so there is nothing to do but start them up with `AppletViewer` and then let the user close the window when they are finished. The Live applet is an exception—it provides a full set of methods to allow the view point, spin, and so on to be modified by *Mathematica* code. These methods are in the Live class, so to call them you need an instance of the Live class. The way you used `AppletViewer` earlier does not give us any instance of the applet class. The construction and destruction of the applet instance was hidden within the internals of `AppletViewer`. You can also call `AppletViewer` with an instance of an applet class instead of just the class name. This lets you manage the lifetime of the applet instance.

```
applet = JavaNew["Live"];
AppletViewer[applet, {"INPUT=" <> dodec, "WIDTH=400", "HEIGHT=400"}];
```

Now you can call methods on the applet instance. See the LiveGraphics3D documentation for the full set of methods. This scriptability opens up lots of possibilities, such as programming "flyby" views of objects, or creating buttons that jump the image into certain orientations or spins.

```
applet@setMagnification[0.5];
```

When you are done, you call `ReleaseJavaObject` to release the applet instance. This can be done before or after the applet window is closed.

```
ReleaseJavaObject[applet]
```

Periodical Tasks

The section "Creating Windows and Other User Interface Elements" described the `ShareKernel` function and how it allows Java and the notebook front end to share the kernel's attention. A side benefit of this functionality is that it becomes easy to provide a means whereby users can schedule arbitrary *Mathematica* programs to run at periodical intervals during a session. Say you have a source that provides continuously updated financial data and you want to have some variables in *Mathematica* constantly reflect the current values. You have written a program that goes out and reads from the source to get the information, but you have to manually run this program all the time while you are working. A better solution would be to set up a periodical task that pulls the data from the source and sets the variables every 15 seconds.

`AddPeriodical[`*expr*`,`*secs*`]`	cause *expr* to be evaluated every *secs* seconds while the kernel is idle
`RemovePeriodical[`*id*`]`	stop scheduling of the periodical represented by *id*
`Periodical[`*id*`]`	return a list `{HoldForm[`*expr*`],` *secs*`}` showing the expression and time interval associated with the periodical represented by *id*
`Periodicals[]`	return a list of the *id* numbers of all currently scheduled periodicals
`SetPeriodicalInterval[`*id*`]`	reset the periodical interval for the periodical task represented by *id*
`$ThisPeriodical`	holds the *id* of the currently executing periodical task

Controlling periodical tasks.

You can set up such a task with the `AddPeriodical` function.

```
id = AddPeriodical[updateFinancialData[], 15];
```

`AddPeriodical` returns an integer ID number that you must use to identify the task—for example, when it comes time to stop scheduling it by calling `RemovePeriodical`. `AddPeriodical` relies on kernel sharing, so it calls `ShareKernel` if it has not already been called. There is no limit on the number of periodicals that can be established.

After scheduling that task, `updateFinancialData[]` will be executed every 15 seconds while the kernel is idle. Note that periodical tasks are run only when the kernel is not busy—they do not interrupt other evaluations. If the kernel is in the middle of another evaluation when the allotted 15 seconds elapses, the task will wait to be executed until immediately after the computation finishes. Any such delayed periodicals are guaranteed to be executed as soon as the kernel finishes with the current computation. They cannot be indefinitely delayed if the user is busy with numerous computations in the front end or in Java. The converse to these facts is also true—if a periodical is executing when the user evaluates a cell in the front end, the evaluation will not be able to start until all periodicals finish, but it is guaranteed to start immediately thereafter.

To remove a single periodical task, use `RemovePeriodical`, supplying the ID number of the periodical as the argument. To remove all periodical tasks, use `RemovePeriodical[Periodicals[]]`. Periodical tasks are all removed if you call `UnshareKernel[]` with no arguments, which turns off all kernel sharing. You would then need to use `AddPeriodical` again to reestablish periodical tasks.

You can reset the scheduling interval for a periodical task by calling `SetPeriodicalInterval`, which is new in *J/Link* 2.0. This line makes the financial data periodical execute every 10 seconds, instead of 15 as shown earlier.

```
SetPeriodicalInterval[id, 10]
```

Sometimes you might want to change the interval for a periodical task or remove it entirely from within the code of the task itself. `$ThisPeriodical` is a variable that holds the ID of the currently executing periodical task. It will only have a value during the execution of a periodical task. You use `$ThisPeriodical` from within your periodical task to obtain its ID so that you can call `RemovePeriodical` or `SetPeriodicalInterval`.

Periodical tasks do not necessarily have anything to do with Java, nor do they need to use Java. Technically, Java does not even need to be running. However, because Java is used by the internals of `ShareKernel` to yield the CPU, if Java is not running then setting a periodical task will cause the kernel to keep the CPU continuously busy. Periodical task functionality is included in *J/Link* because it is a simple extension to `ShareKernel`, and it does have some nice uses in association with Java.

A final note about periodical tasks is that they do not cause output to appear in the front end. Look at this attempt.

```
id = AddPeriodical[Print["hello"], 10];
```

The programmer expects to get `hello` printed in his notebook every 10 seconds, but nothing happens. During the time when periodicals are executed, `$ParentLink` is not assigned to the front end (or Java). Results or side effects like `Print` output, messages, or graphics vanish into the ether.

Before proceeding, clean up the periodical tasks you created.

```
RemovePeriodical[Periodicals[]];
```

Some Special Number Classes

Preamble

There is a set of special number-related classes in Java that *J/Link* maps to their *Mathematica* numeric representation. Like strings and arrays, objects of these number classes have an important property: although they are objects in Java, they have a meaningful "by value" representation in *Mathematica*, so it is convenient for *J/Link* to automatically convert them to numbers as they are returned from Java to *Mathematica*, and back to objects as they are sent from *Mathematica* to Java.

These classes are the so-called "wrapper" classes that represent primitive types (`Byte`, `Integer`, `Long`, `Double`, and so on), `BigDecimal` and `BigInteger`, and any class used to represent complex numbers. The treatment of these classes is described in this section.

The "Wrapper" Classes: Integer, Float, Boolean, and Others

Java has a set of so-called "wrapper" classes that represent primitive types. These classes are `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `Boolean`. The wrapper classes hold single values of their respective primitive types, and are necessary to allow everything in Java to be represented as a subclass of Object. This lets various utility methods and data structures that deal with objects handle primitive types in a straightforward way. It is also necessary for Java's reflection capabilities.

If you have a Java method that returns one of these objects, it will arrive in *Mathematica* as an integer (for `Byte`, `Character`, `Short`, `Integer`, and `Long`), real number (for `Float` and `Double`

ble), or the symbols True or False (for Boolean). Likewise, a Java method that takes one of these objects as an argument can be called from *Mathematica* with the appropriate raw *Mathematica* value. The same rules hold true for arrays of these objects, which are mapped to lists of values.

In the unlikely event that you want to defeat these automatic "pass by value" semantics, you can use the ReturnAsJavaObject and JavaObjectToExpression functions, discussed in "References and Values".

Complex Numbers

You have seen that Java number types (e.g., byte, int, double) are returned to *Mathematica* as integers and reals, and integers and reals are converted to the appropriate types when sent as arguments to Java. What about complex numbers? It would be nice to have a Java class representing complex numbers that mapped directly to *Mathematica*'s Complex type, so that automatic conversions would occur as they were passed back and forth between *Mathematica* and Java. Java does not have a standard class for complex numbers, so *J/Link* lets you name the class that you want to participate in this mapping.

SetComplexClass["*classname*"]	set the class to be mapped to complex numbers in *Mathematica*
GetComplexClass[]	return the class currently used for complex numbers

Setting the class for complex numbers.

You can use any class you like as long as it has the following properties:

1. A public constructor that takes two doubles (the real and imaginary parts, in that order)
2. Methods that return the real and imaginary parts, having the following signatures:

```
public double re();
public double im();
```

Say that you are doing some computations with complex numbers in Java, and you want to interact with these methods from *Mathematica*. You like to use the complex number class available from netlib. This class is named ORG.netlib.math.complex.Complex and is available at http://www.netlib.org/java/. You use the SetComplexClass function to specify the name of the class:

```
SetComplexClass["ORG.netlib.math.complex.Complex"];
```

Now any method or field that takes an argument of type `ORG.netlib.math.complex.Complex` will accept a *Mathematica* complex number, and any object of class `ORG.netlib.math.complex.Complex` returned from a method or field will automatically be converted into a complex number in *Mathematica*. The same holds true for arrays of complex numbers.

Note that you must call `SetComplexClass` before you load any classes that use complex numbers, not merely before you call any methods of the class.

BigInteger and BigDecimal

Java has standard classes for arbitrary-precision floating-point numbers and arbitrary-precision integers. These classes are `java.math.BigDecimal` and `java.math.BigInteger`, respectively. Because *Mathematica* effortlessly handles such "bignums," *J/Link* maps `BigInteger` to *Mathematica* integers and `BigDecimal` to *Mathematica* reals. What this means is that any Java method or field that takes, say, a `BigInteger` can be called from *Mathematica* by passing an integer. Likewise, any method or field that returns a `BigDecimal` will have the value returned to *Mathematica* as a real number.

Ragged Arrays

Java allows arrays that are deeper than one dimension to be "ragged," or non-rectangular, meaning that they do not have the same length at every position at the same level. For example, `{{1,2,3},{4,5},{6,7,8}}` is a ragged two-dimensional array. *J/Link* allows you to send and receive ragged arrays, but it is not the default behavior. The reason for this is simply efficiency—the *MathLink* library has functions that allow very efficient transfer of rectangular arrays of most primitive types (e.g., `byte`, `int`, `double`, and so on), whereas ragged ones have to be picked apart tediously with a series of individual calls to get every piece. This all happens deep inside *J/Link*, so you do not have to be concerned with the mechanics of array passing, but it has a huge impact on speed. To maximize speed, *J/Link* assumes that arrays of primitive types are rectangular. You can toggle back and forth between allowing and rejecting ragged arrays by calling the `AllowRaggedArrays` function with either `True` or `False`.

`AllowRaggedArrays[True]`	allow ragged (i.e., nonrectangular) arrays to be sent to Java

Ragged array support.

With `AllowRaggedArrays[True]`, sending of arrays deeper than one dimension is greatly slowed. Here is an example of array behavior and how it is affected. Assume the class `Testing` has the following method, which takes a two-dimensional array of ints and simply returns it:

```
public static int[][] intArrayIdentity(int[][] a) {
    return a;
}
```

Look what happens if you call it with a ragged array.

```
LoadClass["Testing"];
Testing`intArrayIdentity[{{1, 2, 3}, {4, 5}}]
```

```
Java::argxs1 :
  The static method Testing`intArrayIdentity was called with an incorrect
    number or type of arguments. The argument was {{1,2,3},{4,5}}.
$Failed
```

An error occurs because the *Mathematica* definition for the `Testing`intArrayIdentity()` function requires that its argument be a two-dimensional rectangular array of integers. The call never even gets out of *Mathematica*.

Here you turn on support for ragged arrays, and the call works. This requires modifications in both the *Mathematica*-side type checking on method arguments and the Java-side array-reading routines.

```
AllowRaggedArrays[True]
Testing`intArrayIdentity[{{1, 2, 3}, {4, 5}}]
```

```
{{1, 2, 3}, {4, 5}}
```

It is a good idea to turn off support for ragged arrays as soon as you no longer need it, since it slows arrays down so much.

```
AllowRaggedArrays[False]
```

Implementing a Java Interface with *Mathematica* Code

You have seen how *J/Link* lets you write programs that use existing Java classes. You have also seen how you can wire up the behavior of a Java user interface via callbacks to *Mathematica* via the `MathListener` classes. You can think of any of these `MathListener` classes, such as `MathActionListener`, as a class that "proxies" its behavior to arbitrary user-defined *Mathematica* code. It is as if you have a Java class that has its implementation written in *Mathematica*. This functionality is extremely useful because it greatly extends the set of programs you can write purely in *Mathematica*, without resorting to writing our own Java classes.

`ImplementJavaInterface["`*interfaceName*`",{"`*methName*`"->"`*mathFunc*`",…}]`

create an instance of a Java class that implements the named Java interface by calling back to *Mathematica* according to the given mappings of Java methods to *Mathematica* functions

Implementing a Java interface entirely in *Mathematica*.

It would be nice to be able to take this behavior and generalize it, so that you could take *any* Java interface and implement its methods via callbacks to *Mathematica* functions, and do it all without having to write any Java code. The `ImplementJavaInterface` function, new in *J/Link* 2.0, lets you do precisely that. This function is easier to understand with a concrete example. Say you are writing a *Mathematica* program that uses *J/Link* to display a Java window with a Swing menu, and you want to script the behavior of the menu in *Mathematica.* The Swing `JMenu` class fires events to registered `MenuListeners`, so what you need is a class that implements `MenuListener` by calling into *Mathematica*. A quick glance at the section on MathListeners reveals that *J/Link* does not provide a `MathMenuListener` class for you. You could choose to write your own implementation of such a class, and in fact this would be very easy, even trivial, since you would make it a subclass of `MathListener` and inherit virtually all the functionality you would need. For the sake of this discussion, assume that you choose not to do that, perhaps because you do not know Java or you do not want to deal with all the extra steps required for that solution. Instead, you can use `ImplementJavaInterface` to create such a Java class with a single line of *Mathematica* code:

```
mathMenuListener =
    ImplementJavaInterface["javax.swing.event.MenuListener",
        {"menuSelected" -> "menuSelectedFunc",
         "menuCanceled" -> "menuCanceledFunc",
         "menuDeselected" -> "menuDeselectedFunc"}
    ];
myMenu@addMenuListener[mathMenuListener];

...

(* Later, define the three Mathematica event-handler functions: *)
menuSelectedFunc[menuEvent_] := ...

menuCanceledFunc[menuEvent_] := ...

menuDeselectedFunc[menuEvent_] := ...
```

The first argument to `ImplementJavaInterface` is the Java interface or list of interfaces you want to implement. The second argument is a list of rules that associate the name of a Java method from one of the interfaces with the name of a *Mathematica* function to call to implement that method. The *Mathematica* function will be called with the same arguments that the Java method takes. What `ImplementJavaInterface` returns is a Java object of a newly created class that implements the named interface(s). You use it just like any `JavaObject` obtained by calling `JavaNew` or through any other means. It is just as if you had written your own Java class that implemented the named interface by calling the associated *Mathematica* functions, and then called `JavaNew` to create an instance of that class.

It is not necessary to associate every method in the interface with a *Mathematica* function. Any Java methods you leave out of your list of mappings will be given a default Java implementation that returns `null`. If this is not an appropriate return value for the method (e.g., if the method returns an `int`) and the method gets called at some point an exception will be thrown. Generally, this exception will propagate to the top of the Java call stack and be ignored, but it is recommended that you implement all the methods in the Java interface.

The `ImplementJavaInterface` function makes use of the "dynamic proxy" capability introduced in Java 1.3. It will not work in Java versions earlier than 1.3. All Java runtimes bundled with *Mathematica* 4.2 and later are at Version 1.3 or later. If you have *Mathematica* 4.0 or 4.1, the `ImplementJavaInterface` function is another reason to make sure you have an up-to-date Java runtime for your system.

At first glance, the `ImplementJavaInterface` function might seem to give us the capability to write arbitrary Java classes in the *Mathematica* language, and to some extent that is true. One important thing you cannot do is extend, or subclass, an existing Java class. You also cannot

add methods that do not exist in the interface you are implementing. Event-handler classes are a good example of the type of classes for which this facility is useful. You might think that the `MathListener` classes are rendered obsolete by `ImplementJavaInterface`, and it is true that their functionality can be duplicated with it. The `MathListener` classes are still useful for Java versions earlier than 1.3, but most importantly they are useful for writing pure Java programs that call *Mathematica*. Using a class implemented in *Mathematica* via `ImplementJavaInterface` in a Java program that calls *Mathematica* would be possible, but quite cumbersome. If you want a dual-purpose class that is as easy to use from *Mathematica* as from Java, you should write your own subclass of `MathListener`. One *poor* reason for choosing to use `ImplementJavaInterface` instead of writing a custom Java class is that you are worried about complicating your application by requiring it to include its own Java classes in addition to *Mathematica* code. As explained in "Deploying Applications That Use *J/Link*", it is extremely easy to include supporting Java classes in your application. Your users will not require any extra installation steps nor will they need to modify the Java class path.

Writing Your Own Installable Java Classes

Preamble

The previous sections have shown how to load and use existing Java classes. This gives *Mathematica* programmers immediate access to the entire universe of Java classes. Sometimes, though, existing Java classes are not enough, and you need to write your own.

J/Link essentially obliterates the boundary between Java and *Mathematica*, letting you pass expressions of any type back and forth and use Java objects in *Mathematica* in a meaningful way. This means that when writing your own Java classes to call from *Mathematica*, you usually do not need to do anything special. You write the code in exactly the same way as you would if you wanted to use the class only from Java. (One important exception to this rule is that because it is comparatively slow to call into Java from *Mathematica*, you *might* need to design your classes in a way that will not require an excessive number of method calls from *Mathematica* to get the job done. This issue is discussed in detail in "Overhead of Calls to Java".)

In some cases, you might want to exert more direct control over the interaction with *Mathematica*. For example, you might want a method to return something different to *Mathematica* than what the method itself returns. Or you might want the method to not just return something, but also trigger a side effect in *Mathematica*—for example, printing something or displaying a message under certain conditions. You can even have an extended "dialog" with *Mathematica* before your method returns, perhaps invoking multiple computations in *Mathematica* and reading their results. You might also want to write a class of the `MathListener` type that calls into *Mathematica* as the result of some event triggered in Java.

If you do not want to do any of these things, then you can happily ignore this section. The whole point of *J/Link* is to make unnecessary the need to be concerned about the interaction with *Mathematica* through *MathLink*. Most programmers who want to write Java classes to be used from *Mathematica* will just write Java classes, period, without thinking about *Mathematica* or *J/Link*. Those programmers who want more control, or want to know more about the possibilities available with *J/Link*, read on.

The issues discussed in this section require some knowledge of *MathLink* programming (or, more precisely, *J/Link* programming using the Java methods that use *MathLink*), which is discussed in detail in "Writing Java Programs that use *Mathematica*". The fact that you meet some of these methods and issues here is a consequence of the false but useful dichotomy, noted in the Introduction, between using *MathLink* to write "installable" functions to be called from *Mathematica* and using *MathLink* to write front ends for *Mathematica*. *MathLink* is always used in the same way, it is just that virtually all of it is handled for you in the installable case. This section is about how to go beyond this default behavior, so you will be making direct *J/Link* calls to read and write to the link. Thus you will encounter concepts, classes, and methods in this section that are not explained until "Writing Java Programs That Use *Mathematica*".

Some of the discussion in this section will compare and contrast the process of writing an installable program in C. This is designed to help experienced *MathLink* programmers understand how *J/Link* works, and also to convince you that *J/Link* is a superior solution to using C, C++, or FORTRAN.

Installable Functions—The Old Way

Writing a so-called "installable" or "template" program in C requires a number of steps. If you have a file foo.c that contains a function `foo`, to call it from *Mathematica* you must first write a template (.tm) file that contains a template entry describing how you want `foo` to be called from *Mathematica*, what types of arguments it takes, and what it returns. You then pass this .tm file through a tool called `mprep`, which writes a file of C code that manages some, possibly all, of the *MathLink*-related aspects of the program. You also need to write a simple `main` routine, which is always the same. You then compile all of these files, resulting in an executable for just one platform.

Two big drawbacks of this method are that you need to write a template entry for every single function you want to call (imagine doing that for a whole function library), and the compiled program is not portable to other platforms. The biggest drawback, however, is that there is no automatic support for anything but the simplest types. If you want to do something as basic as returning a list of integers, you need to write the *MathLink* calls to do that yourself. And forget about object-oriented programming, as there is no way to pass "objects" to *Mathematica*.

Installable Functions in Java

J/Link makes all those steps go away. As you have seen all throughout this tutorial, you can literally call any method in any class, without any preparation.

It is only in cases where the default behavior of calling a method and receiving its result is not enough that you need to write specialty Java code. The rest of this section will examine some of the special techniques that can be used.

Setting Up Definitions in Mathematica When Your Class Is Loaded

Template entries in .tm files required by installable *MathLink* programs written in C have two features that might appear to be lost in *J/Link*. The first feature is the ability to specify arbitrary *Mathematica* code to be evaluated when the program is first "installed." This is done by using the `:Evaluate:` line in a template entry. The second feature is the ability to specify the way in which the function is to be called from *Mathematica*, including the name of the *Mathematica* function that maps to the C function, its argument sequence, how those arguments are mapped to the ones provided to the C function, and possibly some processing to be done on them before they are sent. This information is specified in the `:Pattern:` and `:Arguments:` lines of a template entry.

These two features are related to each other, because they both rely on the ability to specify *Mathematica* code that is loaded when an external program is installed. *J/Link* gives you this ability and more, through two special methods called `onLoadClass()` and `onUnloadClass()`. When a class is loaded into *Mathematica*, either directly through `LoadJavaClass` or indirectly by calling `JavaNew`, it is examined to see if it has a method with the following signature:

```
public static void onLoadClass(KernelLink ml);
```

If such a method is present, it will be called after all the method and field definitions for the class are set up in *Mathematica*. Because a class can only be loaded once in a Java session, this method will only be called once in the lifetime of a single Java runtime, although it may be called more than once in the lifetime of a single *Mathematica* kernel (because the user can repeatedly launch and quit the Java runtime). The `KernelLink` that is provided as an argument to this method is of course the link back to *Mathematica*.

A typical use for this feature would be to define the text for an error message issued by one of the methods in the class. Here is an example:

```
public static void onLoadClass(KernelLink ml) throwsMathLinkException {
    ml.evaluate("MyClass::sun = \"The foo() method can only be called on
Sunday.\"");
    ml.discardAnswer();
}
```

Note that this method throws MathLinkException. Your onLoadClass() method can throw any exceptions you like (a MathLinkException would be typical). This will not interfere with the matching of the expected signature for onLoadClass(). If an exception is thrown during onLoadClass, it will be handled gracefully, meaning that the normal operation of LoadJavaClass will not be affected. The only exception to this rule is if your code throws an exception while it is interacting with the link to the kernel, and more specifically, in the period between the time that it sends a computation to the kernel and the time that it begins to read the result. In other words, exceptions you throw will not break the LoadJavaClass mechanism, but it is up to you to make sure that you do not screw up the link's state by starting something you do not finish.

Another reason to use onLoadClass() would be if you wanted to create a *Mathematica* function for users to call that "wrapped" a static method call, providing it with a preferred name or argument sequence. If you have a class named MyClass with the method public static void myMethod(double[a]), the definition that will be automatically created for it in *Mathematica* will require that its argument be a list of real numbers or integers. Say you want to add a definition named MyMethod, having the traditional *Mathematica* capitalization, and you also want this function automatically to use N on its argument so that it will work for anything that will *evaluate* to a list of numbers, such as {Pi, 2Pi, 3Pi}. Here is how you would set up such an additional definition:

```
public static void onLoadClass(KernelLink ml) throwsMathLinkException {
    ml.evaluate("MyMethod[x_] := myMethod[N[x]]");
    ml.discardAnswer();
}
```

In other words, if you are not happy with the interface to the class that will automatically be created in *Mathematica*, you can use onLoadClass() to set up the desired definitions without changing the Java interface.

The *Mathematica* context that will be current when onLoadClass() is called is the context in which all the class' static methods and fields are defined. That is why in the preceding example the definition was made for MyMethod and not MyClass`MyMethod. This is important since you cannot know the correct context in your Java code because it is determined by the user via the AllowShortContext option to LoadJavaClass.

It is generally not a good idea to use `onLoadClass()` to send a lot of code to *Mathematica*. This will make the behavior of your class hard for people to understand because the *Mathematica* code is hidden, and also inflexible since you would have to recompile it to make changes to the embedded *Mathematica* code. If you have a lot of code that needs to accompany a Java class, it is better to put that code into a *Mathematica* package file that you or your users load. That is, rather than having users load a class that dumps a lot of code into *Mathematica*, you should have your users load a *Mathematica* package that loads your class. This will provide the greatest flexibility for future changes and maintenance.

Finally, there is no reason why your `onLoadClass()` method needs to restrict itself to making *J/Link* calls. You could perform operations specific to the Java side, for example, writing some debugging information to the Java console window, opening a file for writing, or whatever else you desire.

Similar to the handling of the `onLoadClass()` method, the `onUnloadClass()` method is called when a class is unloaded. Every loaded class is unloaded automatically by `UninstallJava` right before it quits the Java runtime. You can use `onUnloadClass()` to remove definitions created by `onLoadClass()`, or perform any other clean-up you would like. The signature of `onUnloadClass()` must be the following, although it can throw any exceptions:

```
public static void onUnloadClass(KernelLink ml);
```

Note that the meaning of loading and unloading classes here refers to being loaded by *Mathematica* with `LoadJavaClass` either directly or indirectly. It does not refer to the loading and unloading of classes internally by the Java runtime. Class loading by the Java runtime occurs when the class is first used, which may have occurred long before `LoadJavaClass` was called from *Mathematica*.

Manually Returning a Result to Mathematica

The default behavior of a Java method called from *Mathematica* is to return to *Mathematica* exactly what the method itself returns. There are times, however, when you want to return something else. For example, you might want to return an integer in some circumstances, and a symbol in others. Or you might want a method to return one thing when it is being called from Java, and return something different to *Mathematica*. In these cases, you will need to manually send a result to *Mathematica* before the method returns.

Say you are writing a file-reading class that you want to call from *Mathematica*. Because you want almost the identical behavior to the standard class `java.io.FileInputStream`, your class will be a subclass of it. The only changes you want to make are to provide some more *Mathematica*-like behavior. One example is that you want the `read` method to return not -1 when it reaches the end of the file, but rather the symbol `EndOfFile`, which is what *Mathematica*'s built-in file-reading functions return.

```
import java.io.*;
import com.wolfram.jlink.*;

public class MyFileReader extends FileInputStream {

    <<constructors, other methods deleted>>

    public int read() {

        int i = super.read();
        if (i == -1) {
            KernelLink link = StdLink.getLink();
            if (link != null) {
                link.beginManual();
                try {
                    link.putSymbol("EndOfFile");
                } catch (MathLinkException e) {}
            }
        }
        return i;
    }
}
```

If the file has reached the end, `i` will be -1, and you want to manually return something to *Mathematica*. The first thing you need to do is get a `KernelLink` object that can be used to communicate with *Mathematica*. This is obtained by calling the static method `StdLink.getLink()`. If you have written installable *MathLink* programs in C, you will recognize the choice of names here. A C program has a global variable named stdlink that holds the link back to *Mathematica*. *J/Link* has a `StdLink` class that has a few methods related to this link object.

The first thing you do is check whether `getLink()` returns `null`. It will never be `null` if the method is being called from *Mathematica*, so you can use this test to determine whether the method is being called from *Mathematica* or as part of a normal Java program. In this way, you can have a method that can be used from Java in the usual way when a *Mathematica* kernel is nowhere in sight. The `getLink()` call works no matter if the method is called directly from *Mathematica*, or indirectly as part of a chain of methods triggered by a call from *Mathematica*.

Once you have verified that a link back to the kernel exists, the first thing to do is inform *J/Link* that you will be sending the result back to *Mathematica* yourself, so it should not try automatically to send the method's return value. This is accomplished by calling the `beginManual()` method on the `KernelLink` object.

You *must* call `beginManual()` before you send any part of a result back to *Mathematica*. If you fail to do this, the link will get out of sync and the next *J/Link* call you make from *Mathematica* will probably hang. It is safe to call `beginManual()` more than once, so you do not have to worry that your method might be called from another method that has already called `beginManual()`.

Returning to the example program, the next thing after `beginManual()` is to make the required "put"-type calls to send the result back to *Mathematica* (in this case, just a single `putSymbol()`). As always, these calls can throw a `MathLinkException`, so you need to wrap them in a `try/catch` block. The `catch` handler is empty, since there really is not anything to do in the unlikely event of a *MathLink* error. The internal *J/Link* code that wraps all method calls will handle the cleanup and recovery from any *MathLink* error that might have occurred calling `putSymbol()`. You do not need to do anything for `MathLinkExceptions` that occur while you are putting a result manually. The method call will return `$Failed` to *Mathematica* automatically.

Installable programs written in C can also manually send results back. This is indicated by using the `Manual` keyword in the function's template entry. Thus for C programs the manual/automatic decision must be made at compile time, whereas with *J/Link* it is a runtime switch. You can have it both ways with *J/Link*—a normal automatic return in some circumstances and a manual return in others, as the preceding example demonstrates.

Requesting Evaluations by Mathematica

So far, you have seen only cases where a Java method has a very simple interaction with *Mathematica*. It is called and returns a result, either automatically or manually. There are many circumstances, however, where you might want to have a more complex interaction with *Mathematica*. You might want a message to appear in *Mathematica*, or some `Print` output, or you might want to have *Mathematica* evaluate something and return the answer to you. This is a completely separate issue from what you want to return to *Mathematica* at the *end* of your method—you can request evaluations from the body of a method whether it returns its final result manually or not.

In some sense, when you perform this type of interaction with *Mathematica* you are turning the tables on *Mathematica*, reversing the "master" and "slave" roles for a moment. When *Mathematica* calls into Java, the Java code is acting as the slave, performing a computation and returning control to *Mathematica*. In the middle of a Java method, however, you can call back into *Mathematica*, temporarily turning it into a computational server for the Java side. Thus you would expect to encounter essentially all the same issues that are discussed in "Writing Java Programs That Use *Mathematica*", and you would need to understand the full *J/Link* Java-side API.

The full treatment of the `MathLink` and `KernelLink` interfaces is presented in "Writing Java Programs That Use *Mathematica*". This section discusses a few special methods in `KernelLink` that are specifically for use by "installed" methods. You have already seen one, the `beginManual()` method. Now you will treat the `message()`, `print()`, and `evaluate()` methods.

The task of issuing a *Mathematica* message from a Java method and triggering some `Print` output are so commonly done that the `KernelLink` interface has special methods for these operations. The method `message()` performs all the steps of issuing a *Mathematica* message. It comes in two signatures:

```
public void message(String symtag, String arg);
public void message(String symtag, String[] args);
```

The first form is for when you just have a single string argument to be slotted into the message text, and the second form is for if the message text needs two or more arguments. You can pass `null` as the second argument if the message text needs no arguments.

The `print()` method performs all the steps necessary to invoke *Mathematica*'s `Print` function:

```
public void print(String s);
```

Here is an example method that uses both. Assume that the following messages are defined in *Mathematica* (this could be from loading a package or during this class' `onLoadClass()` method):

```
Foo::arg = "The `1` argument to foo must be greater than or equal to 0."
```

Here is the Java code:

```
public static double foo(double x, double y) {

    KernelLink link = StdLink.getLink();
    if (link != null) {
        link.print("inside foo");
        if (x < 0)
            link.message("Foo::arg", "first");
        if (y < 0)
            link.message("Foo::arg", "second");
    }
    return Math.sqrt(x) * Math.sqrt(y);
}
```

Note that `print()` and `message()` send the required code to *Mathematica* and also read the result from the link (it will always be the symbol `Null`). They do not throw `MathLinkException` so you do not have to wrap them in `try/catch` blocks.

Here is what happens when you call `foo()`:

```
LoadJavaClass["MyClass", StaticsVisible → True];
foo[1.0, -2.0]
```

```
inside foo
```

```
Foo::arg : The second argument to foo must be greater than or equal to 0.
```

```
Indeterminate
```

Note that you automatically get `Indeterminate` returned to *Mathematica* when a floating-point result from Java is NaN ("Not-a-Number").

The methods `print()` and `message()` are convenience functions for two special cases of the more general notion of sending intermediate evaluations to *Mathematica* before your method returns a result. The general means of doing this is to wrap whatever you send to *Mathematica* in `EvaluatePacket`, which is a signal to the kernel that this is not the final result, but rather something that it should evaluate and send the result back to Java. You can explicitly send the `EvaluatePacket` head, or you can use one of the methods in `KernelLink` that use `EvaluatePacket` for you. These methods are:

```
void evaluate (String s) throws MathLinkException;
String evaluateToInputForm (String s, int pageWidth);
String evaluateToOutputForm (String s, int pageWidth);
byte[] evaluateToImage (String s, int width, int height);
byte[] evaluateToTypeset (String s, int pageWidth, boolean useStdForm);
```

These methods are discussed in "Writing Java Programs that use *Mathematica*" (actually, they also come in several more flavors with other argument sequences). Here is a simple example:

```
public static double foo(double x, double y) {

    KernelLink link = StdLink.getLink();
    if (link != null) {
        try {
            link.evaluate("2+2");
            // Wait for, and then read, the answer.
            link.waitForAnswer();
            int sum1 = link.getInteger();

            // evaluateToOutputForm makes the result come back as a
            // string formatted in OutputForm, and all in one step
            // (no waitForAnswer call needed).
            String s = link.evaluateToOutputForm("3+3");
            int sum2 = Integer.parseInt(s);

            // If you want, put the whole evaluation piece by piece,
            // including the EvaluatePacket head.
            link.putFunction("EvaluatePacket");
            link.putFunction("Plus", 2);
            link.put(4);
            link.put(4);
            link.waitForAnswer();
            int sum3 = link.getInteger();
        } catch (MathLinkException e) {
            // The only type of mathlink error we are likely to get
            // is from a "get" function when what we are trying to
            // get is not the type of expression that is waiting. We
            // just clear the error state, throw away the packet we
            // are reading, and let the method finish normally.
            link.clearError();
            link.newPacket();
        }
    }
    return Math.sqrt(x) * Math.sqrt(y);
}
```

Throwing Exceptions

Any exceptions that your method throws will be handled gracefully by *J/Link*, resulting in the printing of a message in *Mathematica* describing the exception. This was discussed in "How Exceptions Are Handled". If you are sending computations to *Mathematica* as described in the previous section, you need to make sure that an exception does not interrupt your code unexpectedly. In other words, if you start a transaction with *Mathematica*, make sure you complete it or you will leave the link out of sync and future calls to Java will probably hang.

Making a Method Interruptible

If you are writing a method that may take a while to complete, you should consider making it interruptible from *Mathematica*. In C *MathLink* programs, a global variable named `MLAbort` is provided for this purpose. In *J/Link* programs, you call the `wasInterrupted()` method in the `KernelLink` interface:

```
public boolean wasInterrupted();
```

Here is an example method that performs a long computation, checking every 100 iterations whether the user tried to abort it (using the **Interrupt Evaluation** or **Abort Evaluation** commands in the **Evaluation** menu).

```
public int foo() {

    KernelLink link = StdLink.getLink();
    for (int i = 0; i < 10000, i++) {
        ... perform one step ...
        if (i % 100 == 0 && link.wasInterrupted())
            return 0;  // Return value will not be seen by Mathematica.
    }
    return 42;
}
```

This method returns 0 if it detects an attempt by the user to abort, but this value will never be seen by *Mathematica*. This is because *J/Link* causes a method or constructor call that is aborted to return `Abort[]`, whether or not you detect the abort in your code. Therefore, if you detect an abort and want to honor the user's request, just return some value right away. When *J/Link*

returns `Abort[]`, the user's entire computation is aborted, just as if the `Abort[]` was embedded in *Mathematica* code. This means that you do not have to be concerned with any details of propagating the abort back to *Mathematica*—all you have to do is return prematurely if you detect an abort request, and the rest is handled for you.

J/Link makes no distinction between an interrupt request and an abort request; they each cause `wasInterrupted()` to return `true`. Recall that *Mathematica* has separate commands for interrupting and aborting computations. The "Abort" operation (Alt+. on Windows) causes the entire computation to end as soon as possible and return `$Aborted`. The "Interrupt" operation (Alt+, on Windows) brings up a dialog box with further choices. If this **Interrupt** dialog box is triggered when a Java method is executing, it has a different set of buttons than when normal *Mathematica* code is executing. One of the options is **Send Abort to Linked Program** and another is **Send Interrupt to Linked Program**. Both of these choices have the same effect for Java methods, which is to cause `wasInterrupted()` to return `true` and the call to return `Abort[]` when it completes. The third button is **Kill Linked Program**, which will cause the Java runtime to quit. If you call a Java method that is not interruptible, killing the Java runtime in this way is the only way to make the method call terminate (you can also kill the Java runtime using process control features of your operating system).

Sometimes you might want a Java method to detect an abort and do something other than cause the entire *Mathematica* computation to abort. For example, you might want a loop to stop and return its results up to that point. Note that this is not generally recommended. Users expect a program to abort and return `$Aborted` when they issue an abort request. In some cases, however, especially if the code is not intended for use by a large community, you might find it useful to use an abort as a "message" to communicate some information to your Java code instead of just having the computation aborted. This idea is similar to *Mathematica*'s `CheckAbort` function, which allows you to detect an abort and absorb it so that it does not propagate further and abort the entire computation. To "absorb" the abort in your Java code so that *J/Link* does not return `Abort[]`, simply call the `clearInterrupt()` method:

```
public void clearInterrupt();
```

Here is an example:

```
public int foo() {

    KernelLink link = StdLink.getLink();
    for (int i = 0; i < 10000, i++) {
        ... perform one step ...
        if (i % 100 == 0 && link.wasInterrupted()) {
            link.clearInterrupt();
            return resultSoFar; // This is the value that will be returned
to Mathematica
        }
    }
    ...
    return 42;
}
```

Writing Your Own Event Handler Code

"Handling Events with *Mathematica* Code: The "MathListener" Classes" introduced the topic of triggering calls into *Mathematica* as a response to events fired in Java, such as clicking a button. A set of classes derived from `MathListener` is provided by *J/Link* for this purpose. You are not required to use the provided `MathListener` classes, of course. You can write your own classes to handle events and put calls into *Mathematica* directly into their code. All the event handler classes in *J/Link* are derived from the abstract base class `MathListener`, which takes care of all the details of interacting with *Mathematica*, and also provides the `setHandler()` methods that you use to associate events with *Mathematica* code. Users who want to write their own `MathListener`-style classes (for example, for one of the Swing-specific event listener interfaces, which *J/Link* does not provide) are strongly encouraged to make their classes subclasses of `MathListener` to inherit all this functionality. You should examine the source code for `MathListener`, and also one of the concrete classes derived from it (`MathActionListener` is probably the simplest one) to see how it is written. You can use this as a starting point for your own implementation.

There is a new feature of *J/Link* 2.0 that should be pointed out in this context. This is the `ImplementJavaInterface` *Mathematica* function, which lets you implement any Java interface entirely in *Mathematica* code. `ImplementJavaInterface` is described in more detail in

"Implementing a Java Interface with *Mathematica* Code", but a common use for it would be to create event-handler classes that implement a "Listener"-type interface for which *J/Link* does not have a built-in `MathListener`. This is discussed in more detail in "Implementing a Java Interface with *Mathematica* Code", and if you choose this technique, then you do not have to worry about any of the issues in this section because they are handled for you.

If you are going to write a Java class, and you choose not to derive your class from `MathListener`, there are two very important rules that *must* be adhered to when writing event-handler code that calls into *Mathematica*. To be more precise, these rules apply whenever you are writing code that needs to call into *Mathematica* at a point when *Mathematica* is not currently calling into Java. That may sound confusing, but it is really very simple. "Requesting Evaluations by *Mathematica*" showed how to request evaluations by *Mathematica* from within a Java method. In this case, *Mathematica* has called your Java method, and while *Mathematica* is waiting for the result, your code calls back to perform some computation. This works fine as described in that earlier section, because at the point the code calls back into *Mathematica*, *Mathematica* is in the middle of a call to Java. This is a true "callback"—*Mathematica* has called Java, and during the handling of this call, Java calls back to *Mathematica*. In contrast, consider the case where some Java code executes in response to a button click. When the button click event fires, *Mathematica* is probably not in the middle of a call to Java.

Special considerations are necessary in the latter case because there are two threads in the Java runtime that are using *MathLink*. The first one is created and used by the internals of *J/Link* to handle standard calls into Java originating in *Mathematica* as described throughout this tutorial. The second one is the Java user interface thread (sometimes called the AWT thread), which is the one on which your event handler code will be called. You need to make sure that your use of the link back to the kernel on the user interface thread does not interfere with *J/Link*'s internal thread.

The following code shows an idealized version of the `actionPerformed()` method in the `MathActionListener` class. The actual code in `MathActionListener` is different, because this work is farmed out to the parent class, `MathListener`, but this example shows the correct flow of operations. This is the code that is executed when the associated object's action occurs (like a button click).

```
public void actionPerformed(ActionEvent e) {
    KernelLink ml = StdLink.getLink();
    StdLink.requestTransaction();
    synchronized (ml) {
        try {
            // Send the code to perform the user's requested operation.
            ml.putFunction("EvaluatePacket", 1);
            ... code to put rest of expression to evaluate goes here ...
            ml.endPacket();
            ml.discardAnswer();
        } catch (MathLinkException exc) {
            ...
        }
    }
}
```

The first rule to note in this code is that the complete transaction with *Mathematica*, which includes sending the code to evaluate and completely reading the result, is wrapped in a `synchronized(ml)` block. This is how you ensure that the user interface thread has exclusive access to the link for the entire transaction. The second rule is that the `synchronized(ml)` statement must be preceded by a call to `StdLink.requestTransaction()`. This call will block until the kernel is at a point where it is ready to accommodate evaluations originating in Java. The call must occur before the `synchronized(ml)` block begins, and once you call it you must make sure that you send something to *Mathematica*. In other words, when `requestTransaction()` returns, the kernel will be blocking in an attempt to read from the Java link. The kernel will be stuck in this state until you send it something, so you must protect against a Java exception being thrown after you call `requestTransaction()` but before you send anything. Typically you will do this simply by calling `requestTransaction()` immediately before the `synchronized(ml)` block begins and you start sending something.

It was just said that StdLink.requestTransaction() will block until the kernel is ready to accept evaluations originating in Java. To be specific, it will block until one of the following conditions occurs:

- *Mathematica* executes DoModal
- *Mathematica* executes ServiceJava
- Kernel sharing has been turned on via ShareKernel or ShareFrontEnd, and the kernel is not busy with another computation
- *Mathematica* is already in the middle of a call to Java
- Java is not being used from *Mathematica* (InstallJava has not been called)

These conditions should make sense given the discussion about creating user interface elements in the section "Creating Windows and Other User Interface Elements". DoModal, ShareKernel, and ServiceJava are the three ways in which you direct the kernel's attention to the Java link so that it can detect incoming request for computations.

If you make the common mistake of inadvertently triggering a call to *Mathematica* from Java before you have called DoModal or ShareKernel, the Java user interface thread will hang. This can be easily remedied by calling DoModal, ShareKernel, or ServiceJava afterwards (ServiceJava may need to be called more than once, if more than one event callback is queued up).

If the rule about when it is necessary to use StdLink.requestTransaction() and synchronized(ml) is confusing, you will be happy to learn that it is fine to use these constructs in any code that calls *Mathematica*. In code that does not need them, they are pointless, but harmless, and will not cause the calling thread to block. If you are writing a Java method that needs to call *Mathematica* and there is any chance that it might be called from the user interface thread, add the StdLink.requestTransaction() and synchronized(ml).

Debugging Your Java Classes

You can use your favorite debugger to debug Java code that is called from *Mathematica*. The only issue is that you typically have to launch a Java program inside the debugger to do this. The Java program that you need to launch is the one that is normally launched for you when you call `InstallJava`. The class that contains *J/Link*'s `main()` method is `com.wolfram.jlink.Install`. Thus, the command line to start *J/Link* that is executed internally by `InstallJava` is typically

```
java -classpath /path/to/JLink.jar com.wolfram.jlink.Install
```

There may be additions or modifications to this depending on the options to `InstallJava`, and also some extra *MathLink*-specific arguments are tacked on at the end. To use a debugger, you just have to launch Java with the appropriate command-line arguments that allow you to establish the link to *Mathematica* manually.

If you use a development environment that has an integrated debugger, then the debugger probably has a setting for the main class to use (the class whose `main()` method will be invoked) and a setting for command-line arguments. For example, in WebGain Visual Café, you can set these values in the **Project** panel of the **Project/Options** dialog. Set the main class to be `com.wolfram.jlink.Install`, and the arguments to be something like this:

```
(On Windows:)
-linkmode listen -linkname foo

(On Unix/Linux:)
-linkmode listen -linkprotocol tcp -linkname 1234
```

Then start the debugging session. You should see the *J/Link* copyright notice printed and then Java will wait for *Mathematica* to connect. To do this, go to your *Mathematica* session, make sure the JLink.m package has been read in, and execute:

```
(* On Windows: *)
ReinstallJava[LinkConnect["foo"]]

(* On Unix: *)
ReinstallJava[LinkConnect["1234", LinkProtocol -> "TCP"]]
```

This works because `ReinstallJava` can take a `LinkObject` as its argument, in which case it will not try to launch Java itself. This allows you to manually establish the *MathLink* connection between Java and *Mathematica*, then feed that link to `ReinstallJava` and let it do the rest of the work of preparing the *Mathematica* and Java sides for interacting with each other.

If you like to use a command-line debugger like jdb, you can do the following:

```
C:\>jdb
Initializing jdb...
> run com.wolfram.jlink.Install -linkmode listen -linkname foo
running ...
main[1] J/Link (tm)
Copyright (C) 1999-2000, Wolfram Research, Inc. All Rights Reserved.
www.wolfram.com
Version 1.1

Current thread "main" died. Execution continuing...
>
```

The message about the main thread dying is normal. Now jdb is ready for commands. First, though, you have to execute in your *Mathematica* session the `LinkConnect` and `ReinstallJava` lines shown earlier. This example was for Windows, so Unix users will have to adjust the `run` line to reflect the proper arguments:

```
> run com.wolfram.jlink.Install -linkmode listen -linkprotocol tcp
-linkname 1234
```

Deploying Applications that use *J/Link*

This section discusses some issues relevant to developers who are creating add-ons for *Mathematica* that use *J/Link*.

J/Link uses its own custom class loader that allows it to find classes in a set of locations beyond the startup class path. As described in "Dynamically Modifying the Class Path", users can grow this set of extra locations to search for classes by calling the `AddToClassPath` function. One of the motivations for having a custom class loader was to make it easy for application developers to distribute applications that have parts of their implementation in Java. If you structure your application directory properly, your users will be able to install it simply by copying it into any standard location for *Mathematica* applications. *J/Link* will be able to find your Java classes immediately, without users having to perform any classpath-related operations or even restart Java.

If your *Mathematica* application uses *J/Link* and includes its own Java components, you should create a Java subdirectory in your application directory. You can place any jar files that your application needs into this Java subdirectory. If you have loose class files (not bundled into a jar file), they should go into an appropriately nested subdirectory of the Java directory. "Appropriately nested" means that if your class is in the Java package `com.somecompany.math`, then its class file goes into the com/somecompany/math subdirectory of the Java directory. If the class is not in any package, it can go directly into the Java directory. *J/Link* can also find native libraries and resources your application needs. Native libraries must be in a subdirectory of your Java/Libraries directory that is named after the `$SystemID` of the platform on which it is installed. Here is an example directory structure for an application that uses *J/Link*:

```
MyApp/
    ... other files and directories used by the application ...
    Java/
        MyAppClasses.jar
        MyImage.gif
        Libraries/
            Windows/
                MyNativeLibrary.dll
            PowerMac/
                MyNativeLibrary
            Darwin/
                libMyNativeLibrary.jnilib
            Linux/
                libMyNativeLibrary.so
            ... and so on for other Unix platforms
```

Your application directory must be placed into one of the standard locations for *Mathematica* applications. These locations are listed as follows. In this notation, `$InstallationDirectory`/AddOns/Applications means "The AddOns/Applications subdirectory of the directory whose value is given by the *Mathematica* variable `$InstallationDirectory`."

`$UserAddOnsDirectory`/Applications (Mathematica 4.2 and later only)

`$AddOnsDirectory`/Applications (Mathematica 4.2 and later only)

`$InstallationDirectory`/AddOns/Applications

`$InstallationDirectory`/AddOns/ExtraPackages

Coding Tips

Here are a few tips on producing high-quality applications. These suggestions are guided by mistakes that developers frequently make.

Call `InstallJava` in the body of a function or functions, not when your package is read in. It is best to avoid side effects during the reading of a package. Users expect reading in a package to be fast and to do nothing but load definitions. If you launch Java at this time, and it fails, it could cause a mysterious hang in the loading process. It is better to call `InstallJava` in the code of one or more of your functions. You probably do not need to call `InstallJava` in every single function that uses Java. Most applications have a few "major" functions that users are likely to use almost exclusively, or at least at the start of their session. If your application does not have this property, then provide an initialization function that your users must call first, and call `InstallJava` inside it.

Call `InstallJava` with no arguments. You cannot know what options your users need for Java on their systems, so do not override what they may have set up. It is the user's responsibility to make sure that they call `SetOptions` to customize the options for `InstallJava` as necessary. Typically this would be done in their `init.m` file.

Make sure you use `JavaBlock` and/or `ReleaseJavaObject` to avoid leaking object references. You cannot know how others will use your code, so you need to be careful to avoid cluttering up their sessions with a potentially large number of useless objects. Sometimes you need to create an object that persists beyond the lifetime of a single *Mathematica* function, like a viewer window. In such cases, use a `MathFrame` or `MathJFrame` as your top-level window and use its `onClose()` method to specify *Mathematica* code that releases all outstanding objects and unregisters kernel or front end sharing you may have used. If this is not possible, provide a cleanup function that users can call manually. Use `LoadedJavaObjects` to look at the list of objects referenced in *Mathematica* before and after your functions run; it should not grow in length.

If you use `ShareKernel` or `ShareFrontEnd`, make sure you save the return values from these functions and pass them as arguments to `UnshareKernel` and `UnshareFrontEnd`. Do not call `UnshareFrontEnd` or `UnshareKernel` with no arguments, as this will shut down sharing even if other applications are using it.

Do not assume that the Java runtime will not be restarted during the lifetime of your application. Although users are strongly discouraged to call `UninstallJava` or `ReinstallJava`, it happens. It is unavoidable that some applications will fail if the Java runtime is shut down at an inopportune time (e.g., when they have a Java window displayed), but there are steps you can take to increase the robustness of your application in the face of Java shut-downs and restarts. One step was already given as the first tip listed—call `InstallJava` at the start of your "major" functions. Another step is to avoid caching `JavaClass` or `JavaObject` expressions unnecessarily, as these will become invalid if Java restarts. An example of this is calling `InstallJava` and then `LoadJavaClass` and `JavaNew` several times when your package file is read in, and storing the results in private variables for the lifetime of your package. This is problematic if Java is restarted. Never store `JavaClass` expressions—call `LoadJavaClass` whenever there is any doubt about whether a class has been loaded into the current Java runtime. Calling `LoadJavaClass` is very inexpensive if the class has already been loaded. If you have a `JavaObject` that is very expensive to create and therefore you feel it necessary to cache it over a long period of time in a user's session, consider using the following idiom to test whether it is still valid whenever it is used. The `JavaObjectQ` test will fail if Java has been shut down or restarted since the object was last created, so you can then restart Java and create and store a new instance of the object.

```
SomeFunction[] :=
    Module[{...},
        If[!JavaObjectQ[$myCachedExpensiveJavaObject],
            InstallJava[];
            $myCachedExpensiveJavaObject = JavaNew[...];
        ];
        ... use $myCachedExpensiveJavaObject ...
    ]
```

Do not call `UninstallJava` or `ReinstallJava` in your application. You need to coexist politely with other applications that may be using Java. Do not assume that when your package is done with Java, the user is done with it as well. Only users should ever call `UninstallJava`, and they should probably never call it either. There is no cost to leaving Java running. Likewise, users will rarely call `ReinstallJava` unless they are doing active Java development and need to reload modified versions of their classes.

Example Programs

Introduction

This section will work through some example programs. These examples are intended to demonstrate a wide variety of techniques and subtleties. Discussions include some nuances in the implementations and touch on most of the major issues in *J/Link* programming.

This will take a relatively rigorous approach, and in particular it will be careful to avoid leaking references. As discussed in the section "JavaBlock", `JavaBlock` and `ReleaseJavaObject` are the tools in this fight, but if you find yourself becoming the least bit confused about the subject, just ignore it completely. For many casual, personal uses of *J/Link*, you can forget about memory management issues, and just let Java objects pile up.

J/Link includes a number of notebooks with sample programs, including most of the programs developed in this section. These notebooks can be found in the `<Mathematica dir>`/SystemFiles/Links/JLink/Examples/Part1 directory.

A Beep Function

Here is a very simple example. *Mathematica* does not have a `Beep` function to provide simple alerts. But Java has a `beep()` method and, by virtue of that, *Mathematica* has one too.

```
Beep[] :=
 (
  LoadJavaClass["java.awt.Toolkit"];
  Toolkit`getDefaultToolkit[]@beep[]
 )
```

You will notice a short delay the first time `Beep[]` is executed. This is due to the `LoadJavaClass` call, which only takes measurable time the first time it is called for any given class.

```
Beep[]
```

This is a perfectly good beep function, and many users will not need to go beyond this. If you are writing code for others to use, however, you will probably want to embellish this code a little bit. Here is a more professional version of the same function.

```
BetterBeep[]:=
    JavaBlock[
        InstallJava[];
        LoadJavaClass["java.awt.Toolkit"];
        Toolkit`getDefaultToolkit[]@beep[];
    ]
```

Note that the first thing you do is call `InstallJava`. It is a good habit to call `InstallJava` in functions that use *J/Link*, at least if you are writing code for others to use. If `InstallJava` has already been called, subsequent calls will do nothing and return very quickly. The whole program is wrapped in `JavaBlock`. As discussed in the section "JavaBlock", `JavaBlock` automates the process of releasing references to objects returned to *Mathematica*. The `getDefaultToolkit()` method returns a `Toolkit` object, so you want to release the `JavaObject` that gets created in *Mathematica*. The `getDefaultToolkit()` method returns a reference to the same `Toolkit` object every time it is called, so even if you do not call `JavaBlock`, you will only "leak" one object in an entire session. You could also write `Beep` using an explicit call to `ReleaseJavaObject`.

```
(* Alternative version *)
BetterBeep2[]:=
    Module[{toolkit},
        InstallJava[];
        LoadJavaClass["java.awt.Toolkit"];
        toolkit = Toolkit`getDefaultToolkit[];
        toolkit@beep[];
        ReleaseJavaObject[toolkit]
    ]
```

The advantage to using `JavaBlock` is that you do not have to think about what, if any, methods might return objects, and you do not have to assign them to variables.

Formatting Dates

Here is an example of a computation performed in Java. Java provides a number of powerful date- and calendar-oriented classes. Say you want to create a nicely formatted string showing the time and date. In this first step you create a new Java `Date` object representing the current date and time.

```
date = JavaNew["java.util.Date"]
```

```
«JavaObject[java.util.Date] »
```

Next you load the DateFormat class and create a formatter capable of formatting dates.

```
LoadJavaClass["java.text.DateFormat"];
dateFormatter = DateFormat`getInstance[]
```

```
«JavaObject[java.text.SimpleDateFormat] »
```

Now you call the format() method, passing the Date object as its argument.

```
dateFormatter@format[date]
```

```
10/9/00 4:56 AM
```

There are many different ways in which dates and times can be formatted, including respecting a user's locale. Java also has a useful number-formatting class, an example of which was given in "An Optimization Example".

A Progress Bar

A simple example of a popup user interface for a *Mathematica* program is a progress bar. This is an example of a "non-interactive" user interface, as defined in "Interactive and Non-Interactive Interfaces", because it does not need to call back to *Mathematica* or return a result to *Mathematica*. The implementation uses the Swing user interface classes, because Swing has a built-in class for progress bars. (You cannot run this example unless you have Swing installed. It comes as a standard part of Java 1.2 or later, but you can get it separately for Java 1.1.x. Most Java development tools that are still at Version 1.1.x come with Swing.) The complete code for this example is also provided in the file ProgressBar.nb in the JLink/Examples/Part1 directory.

The code is commented to point out the general structure. There are several classes and methods used in this code that may be unfamiliar to you. Just keep in mind that this is completely standard Java code translated into *Mathematica* using the *J/Link* conventions. It is line-for-line identical to a Java program that does the same thing.

This code is presented as a complete program, but this does not suggest that it should be developed that way. The interactive nature of *J/Link* lets you tinker with Java objects a line at a time, experimenting until you get things just how you want them. Of course, this is how *Mathematica* programs are typically written, and *J/Link* lets you do the same with Java objects and methods.

You can create a function ShowProgressBar that prepares and displays a progress bar dialog. The bar will be used to show percentage completion of a computation. You can supply the initial percent completed or use the default value of zero. ShowProgressBar returns the JProgressBar object because the bar needs to be updated later by calling setValue(). Note that because you return the bar object from the JavaBlock, it is not released like all other new Java objects created within this JavaBlock. This is a new behavior of JavaBlock in *J/Link* 2.0. If what is returned from a JavaBlock is precisely a single Java object (and not, for example, a list of objects), then this object is not released. JavaBlock is discussed in the section "JavaBlock".

```
ShowProgressBar[title_String:"Computation Progress",
                  caption_String:"Percent complete:",
                  percent_Integer:0
                ] :=
    JavaBlock[
        Module[{frame, panel, label, bar},
            InstallJava[];
            bar = JavaNew["javax.swing.JProgressBar"];
            frame = JavaNew["javax.swing.JFrame", title];
            frame@setSize[300, 110];
            frame@setResizable[False];
            frame@setLocation[400, 400];
            panel = JavaNew["javax.swing.JPanel"];
            panel@setLayout[Null];
            frame@getContentPane[]@add[panel];
            label = JavaNew["javax.swing.JLabel", caption];
            label@setBounds[20, 10, 260, 20];
            panel@add[label];
            bar@setBounds[20, 40, 260, 30];
            bar@setMinimum[0];
            bar@setMaximum[100];
            bar@setValue[percent];
            panel@add[bar];
            JavaShow[frame];
            bar
        ]
    ]
```

You also need a function to close the progress dialog and clean up after it. Only two things need to be done. First, the dispose() method must be called on the top-level frame window that contains the bar. Second, if you want to avoid leaking object references, you need to call ReleaseJavaObject on the bar object because it is the only object reference that escaped the JavaBlock in ShowProgressBar. You need to call dispose() on the JFrame object you created in ShowProgressBar, but you did not save a reference to it. The SwingUtilities class has a handy method windowForComponent() that will retrieve this frame, given the bar object.

```
DestroyProgressBar[bar_?JavaObjectQ] :=
    JavaBlock[
        LoadJavaClass["javax.swing.SwingUtilities"];
        SwingUtilities`windowForComponent[bar]@dispose[];
        ReleaseJavaObject[bar]
    ]
```

The bar dialog has a close box in it, so a user can dismiss it prematurely if desired. This would take care of disposing the dialog, but you would still need to release the bar object. `DestroyProgressBar` (and the bar's `setValue()` method) is safe to call whether or not the user closed the dialog.

Here is how you would use the progress bar in a computation. The call to `ShowProgressBar` displays the bar dialog and returns a reference to the bar object. Then, while the computation is running, you periodically call the `setValue()` method to update the bar's appearance. When the computation is done, you call `DestroyProgressBar`.

```
bar = ShowProgressBar[];
n = 0;
While[n <= 5,
    bar@setValue[n/5 * 100];
    Pause[1]; (* This simulates the time-consuming computation. *)
    n++
];
DestroyProgressBar[bar];
```

An easy way to test whether your code leaks object references is to call `LoadedJavaObjects[]` before and after the computation. If the list of objects gets longer, then you have forgotten to use `ReleaseJavaObject` or improperly used `JavaBlock`.

It can take several seconds to load all the Swing classes used in this example. This means that the first time `ShowProgressBar` is called, there will be a significant delay. You could avoid this delay by using `LoadJavaClass` ahead of time to explicitly load the classes that appear in `JavaNew` statements.

The dialog appears onscreen with its upper left at the coordinates (400, 400). It is left as an exercise to the reader to make it centered on the screen. (Hint: the `java.awt.Toolkit` class has a `getScreenSize()` method).

Finally, because the progress bar uses the Swing classes, you can play with the look-and-feel options that Swing provides. Specifically, you can change the theme at runtime. The progress bar window is not very complicated, so it changes very little in going from one look-and-feel theme to another, but this demonstrates how to do it. The effect is much more dramatic for more complex windows.

First, create a new progress bar window.

```
bar = ShowProgressBar[];
```

Now load some classes from which you need to call static methods.

```
LoadJavaClass["javax.swing.UIManager"];
LoadJavaClass["javax.swing.SwingUtilities"];
```

The default look and feel is the "metal" theme. You can change it to the native style look for your platform as follows (it helps to be able to see the window when doing this).

```
JavaBlock[
    UIManager`setLookAndFeel[UIManager`getSystemLookAndFeelClassName[]];
    frame = SwingUtilities`windowForComponent[bar];
    SwingUtilities`updateComponentTreeUI[frame]
]
```

Clean up.

```
DestroyProgressBar[bar]
```

A Simple Modal Input Dialog

You saw one example of a simple modal dialog in "Modal Windows". Presented here is another one—a basic dialog that prompts the user to enter an angle, with a choice of whether it is being specified in degrees or radians. This will demonstrate a dialog that returns a value to a running *Mathematica* program when it is dismissed, much like *Mathematica*'s built-in `Input` function, which requests a string from the user before returning. Dialogs like this one are not "modal" in the traditional sense that they must be closed before other Java windows can be used, but rather they are modal with respect to the kernel, which is kept busy until they are dismissed (that is, until `DoModal[]` returns). The section "Creating Windows and Other User Interface Elements" discusses modal and modeless Java windows in detail.

The code is rather straightforward and warrants little in the way of commentary. In creating the window and the controls within it, it exactly mirrors the Java code you would use if you were writing the program in Java. One technique it demonstrates is determining whether the **OK** or **Cancel** button was clicked to dismiss the dialog. This is done by having the `MathActionListener` objects assigned to the two buttons return different things in addition to calling `EndModal[]`. Recall that `DoModal[]` returns whatever the code that calls `EndModal[]` returns, so here you have the **OK** button execute `(EndModal[]; True)&`, a pure function that ignores its arguments, calls `EndModal[]`, and returns `True`, whereas the **Cancel** button executes `(EndModal[]; False)&`. Thus, `DoModal[]` returns `True` if the **OK** button was clicked, or `False` if the **Cancel** button was clicked. It will return `Null` if the window's close box was clicked (this behavior comes from the `MathFrame` itself).

It may take several seconds to display the dialog the first time `GetAngle[]` is called. This is due to the one-time cost of loading the several large AWT classes required. Subsequent invocations of `GetAngle[]` will be much quicker.

The complete code for this example is also provided in the file ModalInputDialog.nb in the JLink/Examples/Part1 directory.

```
GetAngle[] :=
    JavaBlock[
        Module[{frm, inputField, cbGroup, degBox, radBox,
                label, okButton, cancelButton, wasOKButton, angle},

            InstallJava[]; (* In case the user has not called it already. *)

            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            label = JavaNew["java.awt.Label", "Enter an angle:"];
            inputField = JavaNew["java.awt.TextField"];
            cbGroup = JavaNew["java.awt.CheckboxGroup"];
            degBox = JavaNew["java.awt.Checkbox", "degrees", cbGroup, True];
            radBox = JavaNew["java.awt.Checkbox", "radians", cbGroup, False];
            okButton = JavaNew["java.awt.Button", "OK"];
            cancelButton = JavaNew["java.awt.Button", "Cancel"];

            frm@setLayout[Null];
            frm@add[label];
            frm@add[inputField];
            frm@add[degBox];
            frm@add[radBox];
            frm@add[okButton];
            frm@add[cancelButton];

            frm@setBounds[200, 200, 200, 160];
            label@setBounds[20, 30, 150, 20];
            inputField@setBounds[20, 70, 60, 28];
            degBox@setBounds[100, 60, 80, 20];
            radBox@setBounds[100, 80, 80, 20];
            okButton@setBounds[40, 120, 50, 20];
            cancelButton@setBounds[100, 120, 50, 20];
            frm@setResizable[False];

            okButton@addActionListener[
                JavaNew["com.wolfram.jlink.MathActionListener",
                        "(EndModal[]; True)&"]
            ];
            cancelButton@addActionListener[
                JavaNew["com.wolfram.jlink.MathActionListener",
                        "(EndModal[]; False)&"]
            ];

            (* Now make the window visible and bring it to the foreground. *)
            JavaShow[frm];

            frm@setModal[];
            wasOKButton = DoModal[];
            (* Even though the window may have been closed, it is perfectly
               OK to extract values from the controls in the window.
            *)
            If[TrueQ[wasOKButton],
                angle = ToExpression[inputField@getText[]];
                If[angle =!= Null && degBox@getState[], angle *= Pi/180],
            (* else *)
                (* We will get here if the Cancel button was clicked
                   (wasOKButton will be False), or the dialog was closed
```

```
                        by clicking in its close box (wasOKButton will be Null).
                    *)
                    angle = $Failed
                ];
                (* If the cancel or OK buttons were clicked, frm is still
                   visible, so we dispose it here.
                *)
                frm@dispose[];
                angle
            ]
        ]
```

Now invoke it.

```
GetAngle[]
```

π

A File Chooser Dialog Box

A useful feature for *Mathematica* programs is to be able to produce a file chooser dialog, such as the typical **Open** or **Save** dialog boxes. You could use such a dialog box to prompt a user for an input file or a file into which to write data. This is easily accomplished in a cross-platform way with Java, specifically with the `JFileChooser` class in the standard Swing library. The code for such a dialog box is provided in the file FileChooserDialog.nb in the JLink/Examples/Part1 directory.

Mathematica 4.0 introduced a new "experimental" function called `FileBrowse[]` that displays a file browser in the front end. Although this function is usable, it has several shortcomings compared to the Java technique presented next. One of the limitations is that it requires that the front end be in use. Another is that it is not customizable, so you always get a **Save file as:** dialog box and the concomitant behavior, which is not appropriate for an **Open**-type dialog box. The `JFileChooser` class used here allows very sophisticated customization, including setting the initial directory, masking out files based on their names or properties, controlling the title and text on the various buttons, supplying functions to validate the choice before the dialog box is allowed to be dismissed, allowing for multiple file selection, and allowing directories to be selected instead of files.

Although this example is a short program, the code has some unfortunate complexity (meaning "ugliness") in it related to making this special type of dialog window come to the foreground on all platforms. For this reason, the code is not presented here. Instead, some topics in the program code will be mentioned; you can read the full code and its associated comments in the example file if you are interested in the implementation details.

The FileChooserDialog function takes three string arguments. The first is the title of the dialog box (for example, **Select a data file to import**), the second is the text to appear on what is essentially the **OK** button (typically this will be **Open** or **Save**), and the third is the directory in which to start. You can also supply no arguments and get a default Open dialog box that starts in the kernel's current directory.

Although this is a "modal" dialog box, there is no need to use DoModal, because the showDialog() method will not return until the user dismisses the dialog box. Recall that DoModal is a way to force *Mathematica* to stall until the dialog box or other window is dismissed. Here, you get this behavior for free from showDialog(). The other thing that DoModal does is put the kernel into a loop where it is ready to receive input from Java, so you can script some of the functionality of the dialog via callbacks to *Mathematica*. The file chooser dialog box does not need to use *Mathematica* in any way until it returns the selected file, so you have no need for this other aspect that DoModal provides.

A second point of interest is in the name of the constant that showDialog() returns to indicate that the user clicked the **Save** or **Open** button instead of the **Cancel** button. The name of this constant in Java is JFileChooser.APPROVE_OPTION. Java names map to *Mathematica* symbols, so they must be translated if they contain characters that are not legal in *Mathematica* symbols, such as the underscore. Underscores are converted to a "U" when they appear in symbols, so the *Mathematica* name of this constant is JFileChooser`APPROVEUOPTION. See "Underscores in Java Names" for more information.

Sharing the Front End: Palette-Type Buttons

As discussed in the section "Creating Windows and Other User Interface Elements", one of the goals of *J/Link* is to allow Java user interface elements to be as close as possible to first-class members of the notebook front end environment in the way notebook and palette windows are. One of the ways this is accomplished is with the ShareKernel function, which allows Java windows to share the kernel's attention with notebook windows. Such Java windows are referred to as "modeless," not in the traditional sense of allowing other Java windows to remain active, but modeless with respect to the kernel, meaning that the kernel is not kept busy while they are open.

Beyond the ability to have Java windows share the kernel with the front end, it would be nice to allow actions in Java to cause effects in notebook windows, such as printing something, displaying a graph, or any of the notebook-manipulation commands like `NotebookApply`, `NotebookPrint`, `SelectionEvaluate`, `SelectionMove`, and so on. A good example of this is palette buttons. A palette button can cause the current selection to be replaced by something else and the resulting expression to be evaluated in place.

The `ShareFrontEnd` function lets actions in Java modeless windows trigger events in a notebook window just like can be done from palette buttons or *Mathematica* code you evaluate manually in a notebook. Remember that you get automatically the ability to interact with the front end when you use a *modal* dialog (i.e., when `DoModal` is running). When Java is being run in a modal way, the kernel's `$ParentLink` always points at the front end, so all side effect outputs get sent to the front end automatically. A modal window would not be acceptable for the palette example here because the palette needs to be an unobtrusive enhancement to the *Mathematica* environment—it cannot lock up the kernel while it is alive. `ShareKernel` allows Java windows to call *Mathematica* without tying up the kernel, and `ShareFrontEnd` is an extension to `ShareKernel` (it calls `ShareKernel` internally) that allows such "modeless" Java windows to interact with the front end. `ShareFrontEnd` is discussed in more detail in "Sharing the Front End".

In the `PrintButton` example that follows, a simple palette-type button is developed in Java that prints its label at the current cursor position in the active notebook. Because of current limitations with `ShareFrontEnd`, this example will not work with a remote kernel; the same machine must be running the kernel and the front end.

```
PrintButton[label_String] :=
    JavaBlock[
        Module[{frm, button, listener, tok},
            InstallJava[];
            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            button = JavaNew["java.awt.Button"];
            frm@add[button];
            frm@pack[];
            button@setLabel[label];
            listener = JavaNew["com.wolfram.jlink.MathActionListener",
                               "printButtonFunc"];
            button@addActionListener[listener];
            tok = ShareFrontEnd[];
            frm@onClose["UnshareFrontEnd[" <> ToString[tok] <> "]"];
            JavaShow[frm]
        ]
    ]

printButtonFunc[event_, _] :=
    JavaBlock[
        NotebookApply[SelectedNotebook[], event@getSource[]@getLabel[]];
        (* We need to explicitly release the event object, since it was
           sent to Mathematica before the JavaBlock was entered. *)
        ReleaseJavaObject[event]
    ]
```

Now invoke the `PrintButton` function to create and display the palette. Click the button to see the button's label (`foo` in this example) inserted at the current cursor location. When you are done, click the window's close box.

```
PrintButton["foo"]
```

The code is mostly straightforward. As usual, you use the `MathFrame` class for the frame window because it closes and disposes of itself when its close box is clicked. You create a `MathActionListener` that calls `buttonFunc` and you assign it to the button. From the table in the section Handling Events with *Mathematica* Code: The "MathListener" Classes, you know that `buttonFunc` will be called with two arguments, the first of which is the `ActionEvent` object. From this object you can obtain the button that was clicked and then its label, which you insert at the current cursor location using the standard `NotebookApply` function. One subtlety is that you need to specify `SelectedNotebook[]` as the target for notebook operations like `NotebookApply`, `NotebookWrite`, `NotebookPrint`, and so on, which take a notebook as an argument. Because of implementation details of `ShareFrontEnd`, the notebook given by `EvaluationNotebook[]` is not the correct target (after all, there is no evaluation currently in progress in the front end when the button is clicked).

The important thing to note in `PrintButton` is the use of `ShareFrontEnd` and `UnshareFrontEnd`. As discussed earlier, `ShareFrontEnd` puts Java into a state where it forwards everything other than the result of a computation to the front end, and puts the front end into a state where it is able to receive it. This is why the `Print` output triggered by clicking the Java button, which would normally be sent to Java (and just discarded there), appears in the front end. Front end sharing (and also kernel sharing) should be turned off when they are no longer needed, but if you are writing code for others to use you cannot just blindly shut sharing down—the user could have other Java windows open that need sharing. To handle this issue, `ShareFrontEnd` (and `ShareKernel`) works on a register/unregister principle. Every time you call `ShareFrontEnd`, it returns a token that represents a request for front end sharing. If front end sharing is not on, it will be turned on. When a program no longer needs front end sharing, it should call `UnshareFrontEnd`, passing the token from `ShareFrontEnd` as the argument. Only when all requests for sharing have been unregistered in this way will sharing actually be turned off.

The `onClose()` method of the `MathFrame` class lets you specify *Mathematica* code to be executed when the frame is closed. This code is executed after all event listeners have been notified, so it is a safe place to turn off sharing. In the `onClose()` code, you call `UnshareFrontEnd` with the token returned by `ShareFrontEnd`. Using the `onClose()` method in this way allows us to avoid writing a cleanup function that users would have to call manually after they were finished with the palette. It is not a problem to leave front end sharing turned on, but it is desirable to have your program alter the user's session as little as possible.

Now expand this example to include more buttons that perform different operations. The complete code for this example is provided in the file Palette.nb in the JLink/Examples/Part1 directory.

The first thing you do is separate the code that manages the frame containing the buttons from the code that produces a button. In this way you will have a reusable palette frame that can hold any number of different buttons. The `ShowPalette` function here takes a list of buttons, arranges them vertically in a frame window, calls `ShareFrontEnd`, and displays the frame in front of the user's notebook window.

```
ShowPalette[buttons:{__?JavaObjectQ}] :=
    JavaBlock[
        Module[{frm, tok},
            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            frm@setLayout[JavaNew["java.awt.GridLayout", 0, 1]];
            frm@add[#]& /@ buttons;
            ReleaseJavaObject[buttons];
            frm@pack[];
            tok = ShareFrontEnd[];
            frm@onClose["UnshareFrontEnd[" <> ToString[tok] <> "]"];
            JavaShow[frm];
        ]
    ]
```

Note that you do not return anything from the `ShowPalette` function—specifically, you do not return the frame object itself. This is because you do not need to refer to the frame ever again. It is destroyed automatically when its close box is clicked (remember, this is a feature of the `MathFrame` class). Because you do not need to keep references to any of the Java objects you create, the entire body of `ShowPalette` can be wrapped in `JavaBlock`.

Now create a reusable `PaletteButton` function that creates a button. You have to pass in only two things: the label text you want on the button and the function (as a string) you want to have invoked when the button is clicked. This is sufficient to allow completely arbitrary button behavior, as the entire functionality of the button is tied up in the button function you pass in as the second argument.

```
PaletteButton[label_String, buttonFunc_String] :=
    JavaBlock[
        Module[{button, listener},
            button = JavaNew["java.awt.Button"];
            button@setLabel[label];
            listener = JavaNew["com.wolfram.jlink.MathActionListener", buttonFunc]
            button@addActionListener[listener];
            button
        ]
    ]
```

You will use the `PaletteButton` function to create four buttons. The first is just the print button just defined, the behavior of which is specified by `printButtonFunc`.

```
btn1 = PaletteButton["foo", "printButtonFunc"];
```

The second will duplicate the functionality of the buttons in the standard **AlgebraicManipulation** front end palette. These buttons wrap a function (e.g., `Expand`) around the current selection and evaluate the resulting expression in place. Here is how you create the button and define the button function for that operation.

```
btn2 = PaletteButton["Expand[■]", "applyButtonFunc"];

applyButtonFunc[event_, _] :=
    JavaBlock[
        With[{nb = SelectedNotebook[]},
            NotebookApply[nb, event@getSource[]@getLabel[], All];
            ReleaseJavaObject[event];
            SelectionEvaluate[nb]
        ];
    ]
```

The third button will create a plot. All you have to do is call a plotting function—the work of directing the graphics output to a new cell in the frontmost notebook is handled internally by *J/Lin*k as a result of having front end sharing turned on via `ShareFrontEnd`.

```
btn3 = PaletteButton["Create Plot", "plotButtonFunc"];

plotButtonFunc[event_, _] :=
    (
        Plot[x, {x, 0, 1}];
        ReleaseJavaObject[event];
    )
```

The final button demonstrates another method for causing text to be inserted at the current cursor location. The first example of this, `printButtonFunc`, uses `NotebookApply`. You can also just call `Print`—as with graphics, `Print` output is automatically routed to the frontmost notebook window by *J/Link* when front end sharing is on. This quick-and-easy `Print` method works fine for many situations when you want something to appear in a notebook window, but using `NotebookApply` is a more rigorous technique. You will see some differences in the effects of these two buttons if you put the insertion point into a `StandardForm` cell and try them.

```
btn4 = PaletteButton["foo", "printButtonFunc2"];

printButtonFunc2[event_, _] :=
    JavaBlock[
        Print[event@getSource[]@getLabel[]];
        ReleaseJavaObject[event];
    ]
```

Now you are finally ready to create the palette and show it.

```
ShowPalette[{btn1, btn2, btn3, btn4}]
```

In closing, it must be noted that although this example has demonstrated some useful techniques, it is not a particularly valuable way to use `ShareFrontEnd`. In creating a simple palette of buttons, you have done nothing that the front end cannot do all by itself. The real uses you will find for `ShareFrontEnd` will presumably involve aspects that cannot be duplicated within the front end, such as more sophisticated dialog boxes or other user interface elements.

Real-Time Algebra: A Mini-Application

This example will put together everything you have learned about modal and modeless Java user interfaces. You will implement the same "mini-application" (essentially just a dialog box) in both modal and modeless flavors. The application is inspired by the classic *MathLink* example program RealTimeAlgebra, originally written for the NeXT computer by Theodore Gray and then done in HyperCard by Doug Stein and John Bonadies. The original RealTimeAlgebra provides an input window into which the user types an expression that depends on certain parameters, an output window that displays the result of the computation, and some sliders that are used to vary the values of the parameters. The output window updates as the sliders are moved, hence the name RealTimeAlgebra. Our implementation of RealTimeAlgebra will be very simplistic, with only a single slider to modify the value of one parameter.

The complete code for this example is provided in the file RealTimeAlgebra.nb in the JLink/Examples/Part1 directory.

Here is the function that creates and displays the window.

```
CreateWindow[] :=
    Module[{frame, slider, listener},
        InstallJava[];
        (* inText and outText are globals, because we need to refer to
           them by name in the scrollFunc. This also means we must
           create them outside the JavaBlock below.
        *)
        inText = JavaNew["java.awt.TextArea", "Expand[(x+1)^a]", 8, 40];
        outText = JavaNew["java.awt.TextArea", 8, 40];
        (* This frame could be created inside the JavaBlock, because it is returned
           from the JavaBlock and therefore will not be released, but it makes
           our intentions more clear to create it outside.
        *)
        frame = JavaNew["com.wolfram.jlink.MathFrame", "RealTimeAlgebra"];
        JavaBlock[
            frame@setLayout[JavaNew["java.awt.BorderLayout"]];
            (* Note that we can refer to the Scrollbar`HORIZONTAL constant within the JavaNew
               command that first loads the Scrollbar class. Its value will not need to be
               resolved until that class has been loaded and all necessary definitions created.
            *)
            slider = JavaNew["java.awt.Scrollbar", Scrollbar`HORIZONTAL, 0, 1, 0, 20];
            frame@add[slider, ReturnAsJavaObject[BorderLayout`NORTH]];
            frame@add[outText, ReturnAsJavaObject[BorderLayout`CENTER]];
            frame@add[inText, ReturnAsJavaObject[BorderLayout`SOUTH]];
            frame@pack[];
            (* Use a fixed-width font for the output window to preserve
               formatting of multi-line expressions. *)
            outText@setFont[JavaNew["java.awt.Font", "Courier", Font`PLAIN, 12]];
            listener = JavaNew["com.wolfram.jlink.MathAdjustmentListener"];
            listener@setHandler["adjustmentValueChanged", "sliderFunc"];
            slider@addAdjustmentListener[listener];
            frame@setLocation[200, 200];
            JavaShow[frame];
        ];
        frame
    ]

(* This is what will be called in response to moving the slider position: *)

sliderFunc[evt_, type_, scrollPos_] :=
    outText@setText[
        Block[{a = scrollPos}, ToString[ToExpression[inText@getText[]]]]
    ]
```

The `sliderFunc` function is called by the `MathAdjustmentListener` whenever the slider's position changes. It gets the text in the `inputText` box, evaluates it in an environment where `a` has the value of the slider position (the range for this is 0..20, as established in the `JavaNew` call that creates the slider), and puts the resulting string into the `outText` box. It then calls `ReleaseJavaObject` to release the first argument, which is the `AdjustmentEvent` object itself. This is the only object passed in as an argument (the other two arguments are integers). If you are wondering how you determine the argument sequence for `sliderFunc`, you get it from the `MathListener` table in the section Handling Events with *Mathematica* Code: The "MathListener" Classes. Note that you need to refer by name to the input and output text boxes in `sliderFunc`, so you cannot make their names local variables in the `Module` of `CreateWindow`, and of course they cannot be created inside that function's `JavaBlock`.

There is one interesting thing in the code that deserves a remark. Look at the lines where you add the three components to the frame. What is the `ReturnAsJavaObject` doing there? The method being called here is in the `Frame` class, and has the following signature:

```
void add(Component comp, Object constraints);
```

The second argument, `constraints`, is typed only as `Object`. The value you pass in depends on the layout manager in use, but typically it is a string, as is the case here (`BorderLayout`NORTH`, for example, is just the string "NORTH"). The problem is that *J/Link* creates a definition for this signature of `add` that expects a `JavaObject` for the second argument, and *Mathematica* strings do not satisfy `JavaObjectQ`, although they are converted to Java string objects when sent. This means that you can only pass strings to methods that expect an argument of type `String`. In the rare cases where a Java method is typed to take an `Object` and you want to pass a string from *Mathematica*, you must first create a Java `String` object with the value you want, and pass that object instead of the raw *Mathematica* string. You have encountered this issue several times before, and you have used `MakeJavaObject` as the trick to get the raw string turned into a reference to a Java `String` object. `MakeJavaObject[BorderLayout`NORTH]` would work fine here, but it is instructive to use a different technique (it also saves a call into Java). `BorderLayout`NORTH` calls into Java to get the value of the `BorderLayout.NORTH` static field, but in the process of returning this string object to *Mathematica*, it gets converted to a raw *Mathematica* string. You need the object reference, not the raw string, so you wrap the access in `ReturnAsJavaObject`, which causes the string, which is normally returned by value, to be returned in the form of a reference.

Getting back to the **RealTimeAlgebra** dialog box, you are now ready to run it as a modal window. You write a special modal version that uses `CreateWindow` internally.

```
RealTimeAlgebraModal[] :=
    JavaBlock[
        (* In the modal case, we can wrap the whole thing in JavaBlock
           and be sure that all the objects will get released, including
           the inText and outText ones needed during event handling.
        *)
        Module[{frm},
            frm = CreateWindow[];
            frm@setModal[];
            DoModal[];
        ]
    ]
```

Note that the whole function is wrapped in JavaBlock. This is an easy way to make sure that all object references created in *Mathematica* while the dialog is running are treated as temporary and released when DoModal finishes. This saves you having to properly use JavaBlock and ReleaseJavaObject in all the handler functions used for your MathListener objects (you will notice that these calls are absent from the sliderFunc function).

Now run the dialog. The RealTimeAlgebraModal function will not return until you close the **RealTimeAlgebra** window, which is what you mean when you call this a "modal" interface.

```
RealTimeAlgebraModal[]
```

It may take several seconds before the window appears the first time. As always, this is the one-time cost of loading all the necessary classes. Play around by dragging the slider, and try changing the text in the input box, for example, to N[Pi, 2 *a*].

Recall that while *Mathematica* is evaluating DoModal[], any Print output, messages, graphics, or any other output or commands other than the result of computations triggered from Java will be sent to the front end. To see this in action, try putting Print[*a*] in the input text box (you will want to arrange windows on your screen so that you can see the notebook window while you are dragging the slider). Next, try Plot[Sin[*a x*], {*x*, 0, 4 Pi}].

Quit RealTimeAlgebra by clicking the window's close box. In addition to closing and disposing of the window, this causes EndModal[] to be executed in *Mathematica*, which then causes DoModal to return. The disposing of the window is due to using the MathFrame class for the window, and executing EndModal[] is the result of calling the setModal() method of MathFrame, as discussed in "Modal Windows".

Now implement RealTimeAlgebra as a modeless window. The CreateWindow function can be used unmodified. The only difference is how you make *Mathematica* able to service the computations triggered by dragging the slider. For a modal window, you use DoModal to force *Mathematica* to pay attention exclusively to the Java link. The drawback to this is that you cannot use the kernel from the notebook front end until DoModal ends. To allow the notebook front end and Java to share the kernel's attention, you use ShareKernel.

```
RealTimeAlgebraModeless[] :=
    Module[{frm, token},
        frm = CreateWindow[];

token = ShareKernel[];

(* We use the MathFrame onClose method to specify code to
           be executed when the frame is closed.The use here is
           typical--we clean up the object references that need to
           persist throughout the lifetime of the window (otherwise
           we would leak these references),and we call UnshareKernel
           to unregister this application's request for kernel sharing.
        *)
        frm@onClose[
            "ReleaseJavaObject[inText, outText]; UnshareKernel[" <> ToString[token] <> "];"
        ];

ReleaseJavaObject[frm]
    ]
```

Now run it.

```
RealTimeAlgebraModeless[]
```

`RealTimeAlgebraModeless` returns immediately after the window is displayed, leaving the front end and the **RealTimeAlgebra** window able to use the kernel for computations.

You still need a little bit of polish on the modeless version, however. First, to avoid leaking object references, you must change `sliderFunc`. With the modal version, you did not bother to use `JavaBlock` or `ReleaseJavaObject` in `sliderFunc` because you had `DoModal` wrapped in `JavaBlock`. Every call to `sliderFunc`, or any other `MathListener` handler function, occurs entirely within the scope of `DoModal`, so you can handle all object releasing at this level. With a modeless interface, you no longer have a single function call that spans the lifetime of the window. Thus, you put memory-management functions in our handler functions. Here is the new `sliderFunc`.

```
sliderFunc[evt_, type_, scrollPos_] :=
    JavaBlock[
        outText@setText[

    Block[{a = scrollPos}, ToString[To Expression[inText@getText[]]]]
        ];
        ReleaseJavaObject[evt]
    ]
```

The `JavaBlock` here is unnecessary because the code it wraps creates no new object references. Out of habit, though, you wrap these handlers in `JavaBlock`. You need to explicitly call `ReleaseJavaObject` on `evt`, which is the `AdjustmentEvent` object, because its reference is created in *Mathematica* before `sliderFunc` is entered, so it will not be released by the `JavaBlock`. The `type` and `scrollPos` arguments are integers, not objects.

Try setting the input text to `Print[a]`. Notice that nothing appears in the front end when you move the slider, in contrast to the modal case. With a modeless interface, the Java link is the kernel's `$ParentLink` during the times when the kernel is servicing a request initiated from the Java side. Thus, the output from `Print` and graphics goes to Java, not the notebook front end. (The Java side ignores this output, in case you are wondering.) To get this output sent to the front end instead, use `ShareFrontEnd`.

```
ShareFrontEnd[];
```

Now if you set the input text to, say, `Print[a]` or `Plot[a x, {x, 0, a}]`, you will see the text and graphics appearing in the front end.

When you are finished, quit `RealTimeAlgebra` by clicking its close box. Then turn off front end sharing that was turned on in the previous input.

```
UnshareFrontEnd[]
```

A modal interface is simpler than a modeless one in terms of how it uses *Mathematica*, and is therefore the preferred method unless you specifically need the modeless attribute. `ShareKernel` and `ShareFrontEnd` are complex functions that put the kernel into an unusual state. They work fine, but do not use them unnecessarily.

GraphicsDlg: Graphics and Typeset Output in a Window

It is useful to be able to display *Mathematica* graphics and typeset expressions in your Java user interface, and this is easy to do using *J/Link*'s `MathCanvas` class. This example demonstrates a simple dialog box that allows the user to type in a *Mathematica* expression and see the output in the form of a picture. If the expression is a plotting or other graphics function, the resulting image is displayed. If the expression is not a graphic, then it is typeset in `TraditionalForm` and displayed as a picture. The example is first presented in modal form and then in modeless form using `ShareKernel` and `ShareFrontEnd`.

This example also demonstrates a trivial example of using Java code that was created by a drag-and-drop GUI builder of the type present in most Java development environments. For layout of simple windows, it is easy enough to do everything from *Mathematica*. This method was chosen for all the examples in this tutorial, writing no Java code and instead scripting the creation and layout of controls in windows with *Mathematica* calls into Java. This has the advantage of not requiring any Java classes to be written and compiled. For more complex windows, however, you will probably find it much easier to create the controls, arrange them in position, set their properties in a GUI builder, and let it generate Java code for you. You might also want to write some additional Java code by hand.

If you choose this route, the question becomes, "How do I connect the Java code thus generated with *Mathematica*?" Any public fields or methods can be called directly from *Mathematica*, but your GUI builder may not have made public all the ones you need to use. You could make these fields and methods public or add some new public methods that expose them. The latter approach is probably preferable since it does not involve modifying the code that the GUI builder wrote, which could confuse the builder or cause it to overwrite your changes in future modifications.

The complete code for this example is provided in the JLink/Examples/Part1/GraphicsDlg directory. Some of the code is in Java.

This example uses the GUI builder in the WebGain Visual Café Java development environment. The builder was used to create a frame window with three controls. The frame window was made to be a subclass of `MathFrame` because you want to inherit the `setModal()` method. In the top left is an AWT `TextArea` that serves as the input box for the expression. To its right is an **Evaluate** button. Occupying the rest of the window is a `MathCanvas`.

Up to this point, no code has been written by hand at all—everything has been done automatically as components were dropped into the frame and their properties set. All that is left to do is to wire up the button so that when it is clicked the input text is taken and supplied as to the `MathCanvas` via its `setMathCommand()` method. You could write that code in Java, using Visual Café's Interaction Wizard to wire up this event (similar facilities exist in other Java GUI builders). You would have to write some Java code by hand, as the code's logic is more complex than can be handled by graphical tools for creating event handlers.

Rather than doing that, move to *Mathematica* to script the rest of the behavior because it is easier and more flexible. You will need to access the `TextArea`, `Button`, and `MathCanvas` objects from *Mathematica*, but the GUI builder made these nonpublic fields of the frame class. Thus, you need to add three public methods that return these objects to the frame class.

```
public Button getEvalButton()       {return evalButton;}
public TextArea getInputTextArea()  {return inputTextArea;}
public MathCanvas getMathCanvas()   {return mathCanvas;}
```

That is all you need to do to the Java code created by the GUI builder.

The GUI builder created a subclass of `MathFrame` that is named `GraphicsDlg`. It also gave it a `main()` method that does nothing but create an instance of the frame and make it visible. You will not bother with the `main()` method, choosing instead to do those two steps manually, since you need a reference to the frame.

Needed before the code is run is a demonstration of one more feature of *J/Link*—the ability to add directories to the class search path dynamically. You need to load the Java classes for this example, but they are not on the Java class path. With *J/Link*, you can add the directory in which the classes reside to the search path by calling `AddToClassPath`. This will work exactly as written in *Mathematica* 4.2 and later. You will need to modify the path if you have an earlier version of *Mathematica*.

```
classDir = ToFileName[{$TopDirectory, "SystemFiles", "Links", "JLink",
                       "Examples", "Part1", "GraphicsDlg"}];
InstallJava[];
AddToClassPath[classDir];
```

Here is the first implementation of the *Mathematica* code to create and run the graphics dialog. This runs the dialog in a modal loop.

```
DoGraphicsDialogModal[] :=
    JavaBlock[
        Module[{frm, btn, listener},
            InstallJava[];
            (* We named the MathFrame subclass GUI builder created "MyFrame". *)
            frm = JavaNew["GraphicsDlg"];
            (* Here we call one of the accessor methods we had to add
               by hand to the GraphicsDlg class.
            *)
            btn = frm@getEvalButton[];
            listener = JavaNew["com.wolfram.jlink.MathActionListener"];
            listener@setHandler["actionPerformed", "btnFunc"];
            btn@addActionListener[listener];
            JavaShow[frm];
            frm@setModal[];
            DoModal[]
        ]
    ]

btnFunc[event_, _] :=
    JavaBlock[
        Module[{frm, expr, textArea, inputText, mathCanvas},
            frm = event@getSource[]@getParent[];
            (* Here we call two of the accessor methods we had to add
               by hand to the GraphicsDlg class.
            *)
            textArea = frm@getInputTextArea[];
            mathCanvas = frm@getMathCanvas[];

            inputText = textArea@getText[];
            (* We have to evaluate the expression ahead of time to determine
               whether it is a graphics object or not, so we can decide
               whether it display it as a plot or as a typeset result.
            *)
            expr = Block[{$DisplayFunction = Identity}, ToExpression[inputText]];
            If[MatchQ[expr, _Graphics | _Graphics3D | _SurfaceGraphics |
                            DensityGraphics | _ContourGraphics],
                mathCanvas@setImageType[MathCanvas`GRAPHICS],
            (* else *)
                mathCanvas@setImageType[MathCanvas`TYPESET];
                mathCanvas@setUsesTraditionalForm[True]
            ];
            mathCanvas@setMathCommand[ToString[expr, InputForm]];
            ReleaseJavaObject[event]
        ]
    ]
```

As mentioned in the section "Creating Windows and Other User Interface Elements" only the notebook front end can perform the feat of taking a typeset (i.e., "box") expression and creating a graphical representation of it. Thus, the `MathCanvas` can render typeset expressions provided that it has a front end available to farm out the chore of creating the appropriate representation. The front end is used to run this example, but it is really because you are

running the Java dialog "modally" that everything works the way it does. All the while the dialog is up, the front end is waiting for a result from a computation (`DoModal[]`), and therefore it is receptive to requests from the kernel for various services. As far as the front end is concerned, the code for `DoModal` invoked the request for typesetting, even though it was actually triggered by clicking a Java button.

Now run the dialog.

```
DoGraphicsDialogModal[]
```

What if you are not happy with the restriction of running the dialog modally? Now you want to have the dialog remain open and active while not interfering with normal use of the kernel from the front end. As discussed in "Modal Windows" and "Real-Time Algebra: A Mini-Application", you get a lot of useful behavior regarding the front end for free when you run your Java user interface modally. One of these features is that the front end is kept receptive to the various sorts of requests the kernel can send to it (such as for typesetting services). You know you can run a Java user interface in a "modeless" way by using `ShareKernel`, but then you give up the ability to have the kernel use the front end during computations initiated by actions in Java. Luckily, the `ShareFrontEnd` function exists to restore these features for modeless windows.

Re-implement the graphics dialog in modeless form.

```
DoGraphicsDialogModeless[] :=
    JavaBlock[
        Module[{frm, btn, listener, tok},
            InstallJava[];
            frm = JavaNew["GraphicsDlg"];
            btn = frm@getEvalButton[];
            listener = JavaNew["com.wolfram.jlink.MathActionListener"];
            listener@setHandler["actionPerformed", "btnFunc"];
            btn@addActionListener[listener];
            tok = ShareFrontEnd[];
            frm@onClose["UnshareFrontEnd[" <> ToString[tok] <> "]"];
            JavaShow[frm]
        ]
     ]
```

The code shown is exactly the same as `DoGraphicsDialogModal` except for the last few lines. You call `ShareFrontEnd` here instead of `setModal` and `DoModal`. That is the only difference—the rest of the code (including `btnFunc`) is exactly the same. Notice also that you use the `onClose()` method of `MathCanvas` to execute code that unregisters the request for front end sharing when the window is closed.

Run the modeless version. Note how you can continue to perform computations in the front end while the window is active.

```
DoGraphicsDialogModeless[]
```

This new version functions exactly like the modeless version except that it does not leave the front end hanging in the middle of a computation. It is interesting to contrast what happens if you turn off front end sharing (but you need to leave kernel sharing on or the Java dialog will break completely). You can do this by replacing `ShareFrontEnd` and `UnshareFrontEnd` in `DoGraphicsDialogModeless` with `ShareKernel` and `UnshareKernel`. Now if you use the dialog you will find that it fails to render typeset expressions, producing just a blank window, but it still renders graphics normally (unless they have some typeset elements in them, such as a plot label). All the functionality is kept intact except for the ability of the kernel to make use of the front end for typesetting services.

BouncingBalls: Drawing in a Window

This example demonstrates drawing in Java windows using the Java graphics API directly from *Mathematica*. It also demonstrates the use of the `ServiceJava` function to periodically allow event handler callbacks into *Mathematica* from Java. The issues surrounding `ServiceJava` and how it compares to `DoModal` and `ShareKernel` are discussed in greater detail in "Manual" Interfaces: The ServiceJava Function.

The full code is a little too long to include here in its entirety, but it is available in the sample file BouncingBalls.nb in the JLink/Examples/Part1 directory. Here is an excerpt that demonstrates the use of `ServiceJava`.

```
...
mwl = JavaNew["com.wolfram.jlink.MathWindowListener"];
mwl@setHandler["windowClosing", "(keepOn = False)&"];
mathCanvas@addWindowListener[mwl];
keepOn = True;
While[keepOn,
    g@setColor[bkgndColor];
    g@fillRect[0, 0, 300, 300];
    drawBall[g, #]& /@ balls;
    mathCanvas@setImage[offscreen];
    balls = recomputePosition /@ balls;
    ServiceJava[]
];
...
```

A MathWindowListener is used to set keepOn = False when the window is closed, which will cause the loop to terminate. While the window is up, mouse clicks will cause new balls to be created, appended to the balls list, and set in motion. This is done with a MathMouseListener (not shown in the code). Thus, *Mathematica* needs to be able to handle calls originating from user actions in Java. As discussed in the section "Creating Windows and Other User Interface Elements", there are three ways to enable *Mathematica* to do this: DoModal (modal interfaces), ShareKernel or ShareFrontEnd (modeless interfaces), and ServiceJava (manual interfaces). A modal loop via DoModal would not be appropriate here because the kernel needs to be computing something at the same time it is servicing calls from Java (it is computing the new positions of the balls and drawing them). ShareKernel would not help because that is a way to give Java access to the kernel *between* computations triggered from the front end, not *during* such computations.

You need to periodically point the kernel's attention at Java to service requests if any are pending, then let the kernel get back to its other work. The function that does this is ServiceJava, and the code above is typical in that it has a loop that calls ServiceJava every time through. The calls from Java that ServiceJava will handle are the ones from mouse clicks to create new balls and when the window is closed.

Spirograph

This example is just a little fun to create an interesting, nontrivial application—an implementation of a simple Spirograph-type drawing program. It is run as a modal window, and it demonstrates drawing into a Java window from *Mathematica,* along with a number of MathListener objects for various event callbacks. It uses the Java Graphics2D API, so it will not run on systems that have only a Java 1.1.x runtime.

The code for this example can be found in the file Spirograph.nb in the JLink/Examples/Part1 directory.

One of the things you will notice is that on a reasonably fast machine, the speed is perfectly acceptable. There is nothing to suggest that the entire functionality of the application is scripted from *Mathematica*. It is very responsive despite the fact that a large number of callbacks to *Mathematica* are triggered. For example, the cursor is changed as you float the mouse over various regions of the window (it changes to a resize cursor in some places), so there is a constant flow of callbacks to *Mathematica* as you move the mouse. This example demonstrates the feasibility of writing a sophisticated application entirely in *Mathematica*.

This application was written in *Mathematica*, but it could also have been written entirely in Java, or a combination of Java and *Mathematica*. An advantage of doing it in *Mathematica* is that you generally can be much more productive. Spirograph would have taken at least twice as long to write in Java. It is invaluable to be able to write and test the program a line at a time, and to debug and modify it while it is running. Even if you intend to eventually port the code to pure Java, it can be very useful to begin writing it in *Mathematica*, just to take advantage of the scripting mode of development.

Modal programs like this are best developed using `ShareFrontEnd`, then made modal only when they are complete. Making it modeless while it is being developed is necessary to be able to build and debug it interactively, because while it is running you can continue to use the front end to modify the code, make new definitions, add debugging statements, and so on. Using `ShareFrontEnd` instead of `ShareKernel` for modeless operation lets *Mathematica* error and warning messages generated by event callbacks, and `Print` statement inserted for debugging, show up in the notebook window. Only when everything is working as desired do you add the `DoModal[]` call to turn it into a modal window.

A Piano Keyboard

With the inclusion of the Java Sound API in Java 1.3 and later, it becomes possible to write Java programs that do sophisticated things with sound, such as playing MIDI instruments. The Piano.nb example in the JLink/Examples/Part1 directory displays a keyboard and lets you play it by clicking the mouse. A popup menu at the top lists the available MIDI instruments. This example was created precisely because it is so far outside the limitations of traditional *Mathemat ica* programming. Using *J/Link*, you can actually write a short and completely portable program, entirely in the *Mathematica* language, that displays a MIDI keyboard and lets you play it! With just a little more work, the code could be modified to record a sequence played and then return it to *Mathematica*, where you could manipulate it by transposing, altering the tempo, and so on.

J/Link Basics

Calling Java from Mathematica

Preamble

J/Link provides *Mathematica* users with the ability to interact with arbitrary Java classes directly from *Mathematica*. You can create objects and call methods directly in the *Mathematica* language. You do not need to write any Java code, or prepare in any way the Java classes you want to use. You also do not need to know anything about *MathLink*. In effect, all of Java becomes a transparent extension to *Mathematica*, almost as if every existing and future Java class were written in the *Mathematica* language itself.

This facility is called "installable Java" because it generalizes the ability that *Mathematica* has always had to plug in extensions written in other languages through the `Install` function. You will see later how *J/Link* vastly simplifies this procedure for Java compared to languages like C or C++. In fact, *J/Link* makes the procedure go away completely, which is why Java becomes a transparent extension to *Mathematica*.

Although Java is often referred to as an interpreted language, this is really a misnomer. To use Java you must write a complete program, compile it, and then execute it (some environments exist that let you interactively execute lines of Java code, but these are special tools, and similar tools exist for traditional languages like C). *Mathematica* users have the luxury of working in a true interpreted, interactive environment that lets them experiment with functions and build and test programs a line at a time. *J/Link* brings this same productive environment to Java programmers. You could say that *Mathematica* becomes a scripting language for Java.

To *Mathematica* users, then, the "installable Java" feature of *J/Link* opens up the expanding universe of Java classes as an extension to *Mathematica*; for Java users, it allows the extraordinarily powerful and versatile *Mathematica* environment to be used as a shell for interactively developing, experimenting with, and testing Java classes.

Loading the J/Link Package

The first step is to load the *J/Link* package file.

```
Needs["JLink`"]
```

Launching the Java Runtime

InstallJava

The next step is to launch the Java runtime and "install" it into *Mathematica*. The function for this is `InstallJava`.

`InstallJava[]`	launch the Java runtime and prepare it for use from *Mathematica*
`ReinstallJava[]`	quit and restart the Java runtime if it is already running
`JavaLink[]`	give the `LinkObject` that is being used to communicate with the Java runtime

Launching the Java runtime.

```
In[1]:= InstallJava[]

Out[1]= LinkObject[d:\jdk122\bin\java, 5, 2]
```

`InstallJava` can be called more than once in a session. On every call after the first, it does nothing. Thus, it is safe to call `InstallJava` in any program you write, without considering whether the user has already called it.

`InstallJava` creates a command line that is used to launch the Java runtime (typically called "java") and specify some initial arguments for it. In rare cases you will need to control what is on this command line, so `InstallJava` takes a number of options for this purpose. Most users will not need to use these options, and in fact you should avoid them. Programmers should not assume that they have the ability to control the launch of the Java runtime, as it might already be running. If for some reason you absolutely must apply options to control the launch of the Java runtime, use `ReinstallJava` instead of `InstallJava`.

`ClassPath->None`	use the default class path of your Java runtime
`ClassPath->"dirs"`	use the specified directories and jar files
`CommandLine->"cmd"`	use the specified command line to launch the Java runtime, instead of "java"

Options for `InstallJava`.

Controlling the Command Used to Launch Java

An important option to InstallJava and ReinstallJava is CommandLine. This specifies the first part of the command line used to launch Java. One use for this option is if you have more than one Java runtime installed on your system, and you want to invoke a specific one:

```
ReinstallJava[CommandLine → "d:\\full\\path\\to\\java.exe"]
```

By default, InstallJava will launch the Java runtime that is bundled with *Mathematica* 4.2 and later. If you have an earlier version of *Mathematica,* the default command line that will be used is java on most systems. If the java executable is not on your system path, you can use InstallJava to point at it. Another use for this option is to specify arguments to Java that are not covered by other options. Here is an example that specifies verbose garbage collection and defines a property named foo to have the value bar.

```
ReinstallJava[CommandLine → "/path/to/java -verbosegc -Dfoo=bar"]
```

Overriding the Class Path

The class path is the set of directories in which the Java runtime looks for classes. When you launch a Java program from your system's command line, the class path used by Java includes some default locations and any locations specified in the CLASSPATH environment variable, if it exists. If you use the –classpath command-line option to specify a set of locations, however, then the CLASSPATH environment variable is ignored. The ClassPath option to InstallJava and ReinstallJava works the same way. If you leave it at the default value, Automatic, then *J/Link* will include the contents of the CLASSPATH environment variable in its class search path. If you set it to None or a string, then the contents of CLASSPATH are not used. If you set it to be a string, use the same syntax that you would use for setting the CLASSPATH environment variable, which is different for Windows and Unix:

```
ReinstallJava[ClassPath → "c:\\my\\java\\dir;d:\\MyJavaStuff.jar"] (* Windows *)

ReinstallJava[ClassPath → "/my/java/dir:/home/me/MyJavaStuff.jar"]
(* Unix/Linux *)
```

J/Link has its own mechanism for controlling the class search path that is very flexible. Not only does *J/Link* automatically search for classes in *Mathematica* application directories, it also lets you dynamically add new search locations while the Java runtime is running. This means that using the ClassPath option to configure the class path when Java first launches is not very important. One setting for the ClassPath option that is sometimes useful is None, to prevent

J/Link from finding any classes from the contents of CLASSPATH. You might want to do this if you had an experimental version of some class in a development directory and you wanted to make sure that *J/Link* used that version in preference to an older one that was present on your CLASSPATH. "The Java Class Path" presents a complete treatment of the subject of how *J/Link* searches for classes, and how to add locations to this search path.

Loading Classes

LoadJavaClass

LoadJavaClass["classname"]	load the specified class into Java and *Mathematica*
LoadClass["classname"]	deprecated name from earlier versions of *J/Link*; use LoadJavaClass instead

Loading classes.

To use a Java class in *Mathematica*, it must first be loaded into the Java runtime and certain definitions must be set up in *Mathematica*. This is accomplished with the LoadJavaClass function. LoadJavaClass takes a string specifying the fully qualified name of the class (i.e., the full hierarchical name with all the periods):

```
urlClass = LoadJavaClass["java.net.URL"]
JavaClass[java.net.URL]
```

The return value is an expression with head JavaClass. This JavaClass expression can be used in many places in *J/Link*, so you might want to assign it to a variable as done here. Virtually everywhere in *J/Link* where a class needs to be specified as an argument, you can use either a JavaClass expression, the fully qualified class name as a string, or an object of the class. Note that you cannot create a valid JavaClass expression by simply typing it in—it must be returned by LoadJavaClass.

When a class has been loaded, you can call static methods in the class, create objects of the class, and invoke methods and access fields of these objects. You can use any *public* constructors, methods, or fields of a class.

`StaticsVisible->True`	make static methods and fields accessible by just their names, not in a special context
`AllowShortContext->False`	make static methods and fields accessible only in their fully qualified class context
`UseTypeChecking->False`	suppress the type checking that is normally inserted in definitions for calls into Java

Options for `LoadJavaClass`.

"The Java Class Path" discusses the details of how and where *J/Link* finds classes. *J/Link* will be able to find classes on the class path, in the special Java extensions directory, and in a set of extra directories that users can control even while *J/Link* is running.

When to Call LoadJavaClass

It is often the case that you do not need to explicitly load a class with `LoadJavaClass`. As described later, when you create a Java object with `JavaNew`, you can supply the class name as a string. If the class has not already been loaded, `LoadJavaClass` will be called internally by `JavaNew`. In fact, anytime a Java object is returned to *Mathematica* its class is loaded automatically if necessary. This would seem to imply that there is little reason to use `LoadJavaClass`. There are a number of reasons why you would want or need to use `LoadJavaClass` explicitly:

- You need to call a static method of a class and you will not create, or have not yet created, an object of that class. A class must be loaded before any of its static methods can be called.
- You need to use one of the options to `LoadJavaClass`. When `LoadJavaClass` is called internally by `JavaNew`, it is called with the default option settings.
- You want to see errors associated with loading a class reported at a well-defined time.
- You want to control where your users experience the initial delay associated with loading a class. Loading a class can take several seconds if it or one of its parent classes is very large (although it rarely takes that long). You might want to avoid a mysterious delay in a function that users expect to be very quick.
- You want to hang on to the `JavaClass` expression returned by `LoadJavaClass` to use it in other functions. Although all functions that take a `JavaClass` can also take a class name string, you might prefer to use a named `JavaClass` variable for readability purposes. It is also slightly faster than using a string, but this will not be perceptible unless you are using it many times in a loop.
- You feel that it makes your code more self-documenting.

The operation of loading a class in *J/Link* is only done once in a *J/Link* session (a session is the period between `InstallJava` and `UninstallJava`). You can call `LoadJavaClass` on a given class as many times as you want, and every call after the first one immediately returns the `JavaClass` expression without doing any work. This is important, as it means that you never have to worry whether a class has been loaded already—if you are not sure, call `LoadJavaClass`.

Developers writing code for a wide audience should always call `LoadJavaClass` on any classes they need in every function that needs them. It is not suitable to call `LoadJavaClass` in the body of your package code when it is read in, as the user may quit and restart the Java runtime (i.e., `UninstallJava` and `InstallJava`) after your package was read. To be safe, every user-level function that uses *J/Link* should call `InstallJava` and `LoadJavaClass` (if `LoadJavaClass` is necessary; see the following). Both calls execute very quickly if they are not needed.

As mentioned already, loading a class can take several seconds in some cases. When a class is loaded, all of its superclasses are loaded in succession, walking up the inheritance hierarchy. Because a given class is only actually loaded once, if you load another class that shares some of the same superclasses as a previously loaded class, these superclasses will not have to be loaded again. This means that loading the second class will be much quicker than the first if any of the shared superclasses were large. An example of this is loading classes in the java.awt package. The class `java.awt.Component` is very large, so the first time you load a class that inherits from it, say `java.awt.Button`, there will be a noticeable delay. Subsequent loading of other classes derived from `Component` will be much quicker.

Contexts and Visibility of Static Members

`LoadJavaClass` has two options that let you control the naming and visibility of static methods and fields. To understand these options, you need to understand the problems they help to solve. This explanation gets a bit ahead since how to call Java methods has not been discussed. When a class is loaded, definitions are created in *Mathematica* that allow you to call methods and access fields of objects of that class. Static members are treated quite differently from nonstatic ones. None of these issues arise for nonstatic members, so only static members are discussed in this section. Say you have a class named `com.foobar.MyClass` that contains a static method named `foo`. When you load this class, a definition must be set up for `foo` so that it can be called by name, something like `foo[`*`args`*`]`. The question becomes: In what context do you want the symbol `foo` defined, and do you want this context to be visible (i.e., on `$ContextPath`)?

J/Link always creates a definition for `foo` in a context that mirrors its fully qualified classname: `com`foobar`MyClass`foo`. This is done to avoid conflicting with symbols named `foo` that might be present in other contexts. However, you might find it clumsy to have to call `foo` by typing the full context name every time, as in `com`foobar`MyClass`foo[`*args*`]`. The option `AllowShortContext -> True` (this is the default setting) causes *J/Link* to also make definitions for `foo` accessible in a shortened context, one that consists of just the class name without the hierarchical package name prefix. In the example, this means that you could call `foo` as simply `MyClass`foo[`*args*`]`. If you need to avoid use of the short context because there is already a context of the same name in your *Mathematica* session, you can use `AllowShortContext -> False`. This forces all names to be put only in the "deep" context. Note that even with `AllowShortContext -> True`, names for statics are also put into the deep context, so you can always use the deep context to refer to a symbol if you desire.

`AllowShortContext`, then, lets you control the context where the symbol names are defined. The other option, `StaticsVisible`, controls whether this context is made visible (put on `$ContextPath`) or not. The default is `StaticsVisible -> False`, so you have to use a context name when referring to a symbol, as in `MyClass`foo[`*args*`]`. With `StaticsVisible -> True`, `MyClass`` will be put on `$ContextPath`, so you could just write `foo[`*args*`]`. Having the default be `True` would be a bit dangerous—every time you load a class a potentially large number of names would suddenly be created and made visible in your *Mathematica* session, opening up the possibility for all sorts of "shadowing" problems if symbols of the same names were already present. This problem is particularly acute with Java, because method and field names in Java typically begin with a lowercase letter, which is also the convention for user-defined symbols in *Mathematica*. Some Java classes define static methods and fields with names like `x`, `y`, `width`, and so on, so shadowing errors are very likely to occur (see "Contexts" for a discussion of contexts and shadowing problems).

For these reasons `StaticsVisible -> True` is recommended only for classes that you have written, or ones whose contents you are familiar with. In such cases, it can save you some typing, make your code more readable, and prevent the all-too-easy bug of forgetting to type the package prefix. A classic example would be implementing the venerable "addtwo" *MathLink* example program. In Java, it might look like this:

```
public class AddTwo {
    public static int addtwo(int i, int j) {return i + j;}
}
```

With the default `StaticsVisible -> False`, you would have to call addtwo as `AddTwo`addtwo[3, 4]`. Setting `StaticsVisible -> True` lets you write the more obvious `addtwo[3, 4]`.

Be reminded that these options are only for *static* methods and fields. As discussed later, non-statics are handled in a way that makes context and visibility issues go away completely.

Inner Classes

Inner classes are public classes defined inside another public class. For example, the class `javax.swing.Box` has an inner class named `Filler`. When you refer to the `Filler` class in a Java program, you typically use the outer class name, followed by a period, then the inner class name:

```
Box.Filler f = new Box.Filler(…);
```

You can use inner classes with *J/Link*, but you need to use the true internal name of the class, which has a `$`, not a period, separating the outer and inner class names:

```
filler = JavaNew["java.swing.Box$Filler", …]
```

If you look at the class files produced by the Java compiler, you will see these `$`-separated class names for inner classes.

Conversion of Types Between Java and Mathematica

Before you encounter the operations of creating Java objects and calling methods, you should examine the mapping of types between *Mathematica* and Java. When a Java method returns a result to *Mathematica*, the result is automatically converted into a *Mathematica* expression. For example, Java integer types (e.g., byte, short, int, and so on), are converted into *Mathematica* integers, and Java real number types (float, double) are converted into *Mathematica* reals. The following table shows the complete set of conversions. These conversions work both ways—for example, when a *Mathematica* integer is sent to a Java method that requires a byte value, the integer is automatically converted to a Java byte.

Java type	*Mathematica type*
`byte`, `char`, `short`, `int`, `long`	`Integer`
`Byte`, `Character`, `Short`, `Integer`, `Long`, `BigInteger`	`Integer`
`float`, `double`	`Real`
`Float`, `Double`, `BigDecimal`	`Real`
`boolean`	`True` or `False`
`String`	`String`
`array`	`List`
controlled by user (see "Complex Numbers")	`Complex`
`Object`	`JavaObject`
`Expr`	any expression
`null`	`Null`

Corresponding types in Java and *Mathematica*.

Java arrays are mapped to *Mathematica* lists of the appropriate depth. Thus, when you call a method that takes a `double[]`, you might pass it `{1.0, 2.0, N[Pi], 1.23}`. Similarly, a method that returns a two-deep array of integers (i.e., `int[][]`) might return to *Mathematica* the expression `{{1, 2, 3}, {5, 3, 1}}`.

In most cases, *J/Link* will let you supply a *Mathematica* integer to a method that is typed to take a real type (`float` or `double`). Similarly, a method that takes a `double[]` could be passed a list of mixed integers and reals. The only times when you cannot do this are the rare cases where a method has two signatures that differ only in a real versus integer type at the same argument slot. For example, consider a class with these methods:

```
public void foo(byte b, Object obj);
public void foo(float f, Object obj);
public void bar(float f, Object obj);
```

J/Link would create two *Mathematica* definitions for the method `foo`—one that required an integer for the first argument and invoked the first signature, and one that required a real number for the first argument and invoked the second signature. The definition created for the method `bar` would accept an integer or a real for the first argument. In other words, *J/Link* will automatically convert integers to reals, except in cases where such conversion makes it ambiguous as to which signature of a given method to invoke. This is not strictly true, though, as *J/Link* does not try as hard as it possibly could to determine whether real versus integer ambigu-

ous as to which signature of a given method to invoke. This is not strictly true, though, as *J/Link* does not try as hard as it possibly could to determine whether real versus integer ambiguity is a problem at every argument position. The presence of ambiguity at one position will cause *J/Link* to give up and require exact type matching at all argument positions. This is starting to sound confusing, but you will find that in most cases *J/Link* allows you to pass integers or lists with integers to methods that take reals or arrays of reals, respectively, as arguments. In cases where it does not, the call will fail with an error message, and you will have to use *Mathematica*'s `N` function to convert all integers to reals explicitly.

Creating Objects

To instantiate Java objects, use the `JavaNew` function. The first argument to `JavaNew` is the object's class, specified either as a `JavaClass` expression returned from `LoadJavaClass` or as a string giving the fully qualified class name (i.e., having the full package prefix with all the periods). If you wish to supply any arguments to the object's constructor, they follow as a sequence after the class.

`JavaNew[`*cls*`,`*arg1*`,…]`	construct a new object of the specified class and return it to *Mathematica*
`JavaNew["`*classname*`",`*arg1*`,…]`	construct a new object of the specified class and return it to *Mathematica*

Constructing Java objects.

For example, this will create a new `Frame`.

```
frm = JavaNew["java.awt.Frame"]
```

```
«JavaObject[java.awt.Frame] »
```

The return value from `JavaNew` is a strange expression that looks like it has the head `JavaObject`, except that it is enclosed in angle brackets. The angle brackets are used to indicate that the form in which the expression is displayed is quite different from its internal representation. These expressions will be referred to as `JavaObject` expressions. `JavaObject` expressions are displayed in a way that shows their class name, but you should consider them opaque, meaning that you cannot pick them apart or peer into their insides. You can only use them in *J/Link* functions that take `JavaObject` expressions. For example, if *obj* is a `JavaObject`, you cannot use `First[`*obj*`]` to get its class name. Instead, there is a *J/Link* function, `ClassName[`*obj*`]`, for this purpose.

JavaNew invokes a Java constructor appropriate for the types of the arguments being passed in, and then returns to *Mathematica* what is, in effect, a reference to the object. That is how you should think of JavaObject expressions—as references to Java objects very much like object references in the Java language itself. What is returned to *Mathematica* is not large no matter what type of object you are constructing. In particular, the object's data (that is, its fields) are not sent back to *Mathematica*. The actual object remains on the Java side, and *Mathematica* gets a reference to it.

The Frame class has a second constructor, which takes a title in the form of a string. Here is how you would call that constructor.

```
frm = JavaNew["java.awt.Frame", "My Example Frame"]
```

```
«JavaObject[java.awt.Frame] »
```

Note that simply constructing a Frame does not cause it to appear. That requires a separate step (calling the frame's show or setVisible methods will work, but as you will see later, *J/Link* provides a special function, JavaShow, to make Java windows appear and come to the foreground).

The previous examples specified the class by giving its name as a string. You can also use a JavaClass expression, which is a special expression returned by LoadJavaClass that identifies a class in a particularly efficient manner. When you specify the class name as a string, the class is loaded if it has not already been.

```
frameClass = LoadJavaClass["java.awt.Frame"];
frm = JavaNew[frameClass, "My Example Frame"];
```

JavaNew is not the only way to get a reference to a Java object in *Mathematica*. Many methods and fields return objects, and when you call such a method, a JavaObject expression is created. Such objects can be used in the same way as ones you explicitly construct with JavaNew.

At this point, you may be wondering about things like reference counts and how objects returned to *Mathematica* get cleaned up. These issues are discussed in "Object References in *Mathematica*".

J/Link has two other functions for creating Java objects, called MakeJavaObject and MakeJavaExpr. These specialized functions are described in the section "MakeJavaObject and MakeJavaExpr".

Calling Methods and Accessing Fields

Syntax

The *Mathematica* syntax for calling Java methods and accessing fields is very similar to Java syntax. The following box compares the *Mathematica* and Java ways of calling constructors, methods, fields, static methods, and static fields. You can see that *Mathematica* programs that use Java are written in almost exactly the same way as Java programs, except *Mathematica* uses `[]` instead of `()` for arguments, and *Mathematica* uses `@` instead of Java's `.` (dot) as the "member access" operator.

An exception is that for static methods, *Mathematica* uses the context mark `` ` `` in place of Java's dot. This parallels Java usage also, as Java's use of the dot in this circumstance is really as a scope resolution operator (like `::` in C++). Although *Mathematica* does not use this terminology, its scope resolution operator is the context mark. Java's hierarchical package names map directly to *Mathematica*'s hierarchical contexts.

	constructors
Java:	`MyClass obj=new MyClass (args);`
Mathematica:	`obj=JavaNew["MyClass",args];`
	methods
Java:	`obj.methodName (args);`
Mathematica:	`obj@methodName[args]`
	fields
Java:	`obj.fieldName=1;` `value=obj.fieldName;`
Mathematica:	`obj@fieldName=1;` `value=obj@fieldName;`
	static methods
Java:	`MyClass.staticMethod (args);`
Mathematica:	`MyClass`staticMethod[args];`
	static fields
Java:	`MyClass.staticField=1;` `value=MyClass.staticField;`
Mathematica:	`MyClass`staticField=1;` `value=MyClass`staticField;`

Java and *Mathematica* syntax comparison.

You may already be familiar with @ as a *Mathematica* operator for applying a function to an argument: f@x is equivalent to the more commonly used f[x]. *J/Link* does not usurp @ for some special operation—it is really just normal function application slightly disguised. This means that you do not have to use @ at all. The following are equivalent ways of invoking a method:

```
(* These are equivalent *)
obj@method[args];
obj[method[args]];
```

The first form preserves the natural mapping of Java's syntax to *Mathematica*'s, and it will be used exclusively in this tutorial.

When you call methods or fields and get results back, *J/Link* automatically converts arguments and results to and from their *Mathematica* representations according to the table in "Conversion of Types between Java and *Mathematica*".

Method calls can be chained in *Mathematica* just like in Java. For example, if meth1 returns a Java object, you could write in Java obj.meth1().meth2(). In *Mathematica*, this becomes obj@meth1[]@meth2[]. Note that there is an apparent problem here: *Mathematica*'s @ operator groups to the right, whereas Java's dot groups to the left. In other words, obj.meth1().meth2() in Java is really (obj.meth1()).meth2() whereas obj@meth1[]@meth2[] in *Mathematica* would normally be obj@(meth1[]@meth2[]). I say "normally" because *J/Link* automatically causes chained calls to group to the left like Java. It does this by defining rules for JavaObject expressions, not by altering the properties of the @ operator, so the global behavior of @ is not affected. This chaining behavior only applies to method calls, not fields. You cannot do this:

```
(* These are incorrect. You cannot chain calls after a field access. *)
x = obj@field@method[args];
x = obj@field1@field2;
```

You would have to split these up into two lines. For example, the second line above would become:

```
temp = obj@field1;
x = temp@field2;
```

In Java, like other object-oriented languages, method and field names are scoped by the object on which they are called. In other words, when you write obj.meth(), Java knows that you are calling the method named meth that resides in obj's class, even though there may be other methods named meth in other classes. *J/Link* preserves this scoping for *Mathematica* symbols

so that there is never a conflict with existing symbols of the same name. When you write obj@meth[], there is no conflict with any other symbols named meth in the system—the symbol meth used by *Mathematica* in the evaluation of this call is the one set up by *J/Link* for this class. Here is an example using a field. First, you create a Point object.

```
pt = JavaNew["java.awt.Point"]
```

```
«JavaObject[java.awt.Point] »
```

The Point class has fields named x and y, which hold its coordinates. A user's session is also likely to have symbols named x or y in it, however. You set up a definition for x that will tell you when it is evaluated.

```
x := Print["gotcha"]
```

Now set a value for the field named x (this would be written as pt.x = 42 in Java).

```
pt@x = 42;
```

You will notice that "gotcha" was not printed. There is no conflict between the symbol x in the Global` context that has the Print definition and the symbol x that is used during the evaluation of this line of code. *J/Link* protects the names of methods and fields on the right-hand side of @ so that they do not conflict with, or rely on, any definitions that might exist for these symbols in visible contexts. Here is a method example that demonstrates this issue differently.

```
frm = JavaNew["java.awt.Frame"];
frm@show[]
```

Even though a new symbol show is being created here, the show that is used by *J/Link* is the one that resides down in the java`awt`Frame context, which has the necessary definitions set up for it.

In summary, for nonstatic methods and fields, you never have to worry about name conflicts and shadowing, no matter what context you are in or what the $ContextPath is at the moment. This is not true for static members, however. Static methods and fields are called by their full name, without an object reference, so there is no object out front to scope the name. Here is a simple example of a static method call that invokes the Java garbage collector. You need to call LoadJavaClass before you call a static method to make sure the class has been loaded.

```
LoadJavaClass["java.lang.Runtime"];
Runtime`gc[];
```

The name scoping issue is not usually a problem with statics, because they are defined in their own contexts (`Runtime`` in this example). These contexts are usually not on `$ContextPath`, so you do not have to worry that there is a symbol of the same name in the `Global`` context or in a package that has been read. There is more discussion of this issue in the section on `LoadJavaClass`, because `LoadJavaClass` takes options that determine the contexts in which static methods are defined and whether or not they are put on `$ContextPath`. If there is already a context named `Runtime`` in your session, and it has its own symbol `gc`, you can always avoid a conflict by using the fully hierarchical context name that corresponds to the full class name for a static member.

```
java`lang`Runtime`gc[];
```

Finally, just as in Java, you can call a static method on an object if you like. In this case, since there is an object out front, you get the name scoping. Here you call a static method of the `Runtime` class that returns the current `Runtime` object (you cannot create a `Runtime` object with `JavaNew`, as `Runtime` has no constructors). You then invoke the (static) method `gc` on the object, and you can use `gc` without any context prefix.

```
runtime = Runtime`getRuntime[];
runtime@gc[];
```

Underscores in Java Names

Java names can have characters in them that are not legal in *Mathematica* symbols. The only common one is the underscore. *J/Link* maps underscores in class, method, and field names to "U". Note that this mapping is only used where it is necessary—when names are used in symbolic form, not as strings. For example, assume you have a class named `com.acme.My_Class`. When you refer to this class name as a string, you use the underscore.

```
LoadJavaClass["com.acme.My_Class"];
JavaNew["com.acme.My_Class"];
```

But when you call a static method in such a class, the hierarchical context name is symbolic, so you must convert the underscore to `U`.

```
com`acme`MyUClass`staticMethod[];
MyUClass`staticMethod[];
```

The same rule applies to method and field names. Many Java field names have underscores in them, for example `java.awt.Frame.TOP_ALIGNMENT`. To refer to this method in code, use the `U`.

```
LoadJavaClass["java.awt.Frame"];
Frame`TOPUALIGNMENT
0.
```

In cases where you supply a string, leave the underscore.

```
Fields["java.awt.Frame", "*_ALIGNMENT"]
```

```
static final float BOTTOM_ALIGNMENT
static final float CENTER_ALIGNMENT
static final float LEFT_ALIGNMENT
static final float RIGHT_ALIGNMENT
static final float TOP_ALIGNMENT
```

Getting Information about Classes and Objects

J/Link has some useful functions that show you the constructors, methods, and fields available for a given class or object.

`Constructors[`*cls*`]`	return a table of the public constructors and their arguments
`Constructors[`*obj*`]`	constructors for this object's class
`Methods[`*cls*`]`	return a table of the public methods and their arguments
`Methods[`*cls*`,"`*pat*`"]`	show only methods whose names match the string pattern *pat*
`Methods[`*obj*`]`	show methods for this object's class
`Fields[`*cls*`]`	return a table of the public fields
`Fields[`*cls*`,"`*pat*`"]`	show only fields whose names match the string pattern *pat*
`Fields[`*obj*`]`	show fields for this object's class
`ClassName[`*cls*`]`	return, as a string, the name of the class represented by *cls*
`ClassName[`*obj*`]`	return, as a string, the name of this object's class
`GetClass[`*obj*`]`	return the `JavaClass` representing this object's class
`ParentClass[`*obj*`]`	return the `JavaClass` representing this object's parent class
`InstanceOf[`*obj*`,`*cls*`]`	return `True` if this object is an instance of *cls*, `False` otherwise
`JavaObjectQ[`*expr*`]`	return `True` if *expr* is a valid reference to a Java object, `False` otherwise

Getting information about classes and objects.

You can give an object or a class to Constructors, Methods, and Fields. The class can be specified either by its full name as a string, or as a JavaClass expression:

```
urlClass = LoadJavaClass["java.net.URL"];
urlObject = JavaNew["java.net.URL", "http://www.wolfram.com"];
(* The next three lines are equivalent *)
Methods[urlClass]
Methods[urlObject]
Methods["java.net.URL"]
```

The declarations returned by these functions have been simplified by removing the Java keywords public, final (removed only for methods, not fields), synchronized, native, volatile, and transient. The declarations will always be public, and the other modifiers are probably not relevant for use via *J/Link*.

Methods and Fields take one option, Inherited, which specifies whether to include members inherited from superclasses and interfaces or show only members declared in the class itself. The default is Inherited -> True.

Inherited->False	show only members that are declared in the class itself, not inherited from superclasses or interfaces

Option for Methods and Fields.

There are additional functions that give information about objects and classes. These functions are ClassName, GetClass, ParentClass, InstanceOf, and JavaObjectQ. They are self-explanatory, for the most part. The InstanceOf function mimics the Java language's instanceof operator. JavaObjectQ is useful for writing patterns that match only valid Java objects:

```
Stringify[obj_?JavaObjectQ] := obj[toString[]]
```

JavaObjectQ returns True if and only if its argument is a valid reference to a Java object or if it is the symbol Null, which maps to Java's null object.

Quitting or Restarting Java

When you are finished with using Java in a *Mathematica* session, you can quit the Java runtime by calling UninstallJava[].

UninstallJava[]	quit the Java runtime
ReinstallJava[]	restart the Java runtime

Quitting the Java runtime.

In addition to quitting Java, `UninstallJava` clears out the many symbols and definitions created in *Mathematica* when you load classes. All outstanding `JavaObject` expressions will become invalid when Java is quit. They will no longer satisfy `JavaObjectQ`, and they will show up as raw symbols like `JLink`Objects`JavaObject12345678` instead of `<< JavaObject[classname] >>`.

Most users will have no reason to call `UninstallJava`. You should think of the Java runtime as an integral part of the *Mathematica* system—start it up, and then just leave it running. All code that uses *J/Link* shares the same Java runtime, and there may be packages that you are using that make use of Java without you even knowing it. Shutting down Java might compromise their functionality. Developers writing packages should *never* call `UninstallJava` in their packages. You cannot assume that when your application is done with *J/Link*, your users are done with it as well.

About the only common reason to need to stop and restart Java is when you are actively developing Java classes that you want to call from *Mathematica*. Once a class is loaded into the Java runtime, it cannot be unloaded. If you want to modify and recompile your class, you need to restart Java to reload the modified version. Even in this circumstance, though, you will not be calling `UninstallJava`. Instead, you will call `ReinstallJava`, which simply calls `UninstallJava` followed by `InstallJava` again.

Version Information

J/Link provides three symbols that supply version information. These symbols provide the same type of information as their counterparts in *Mathematica* itself, except that they are in the `JLink`Information`` context, which is not on `$ContextPath`, so you must specify them by their full names.

`JLink`Information`$Version`	a string giving full version information
`JLink`Information`$VersionNumber`	a real number giving the current version number
`JLink`Information`$ReleaseNumber`	an integer giving the release number (the last digit in a full x.x.x version specification)
`ShowJavaConsole[]`	the console window will show version information for the Java runtime and the *J/Link* Java component

J/Link version information.

```
JLink`Information`$Version
J/Link Version 4.0.1

JLink`Information`$VersionNumber
4.

JLink`Information`$ReleaseNumber
1
```

The ShowJavaConsole[] function, described in "The Java Console Window", will also display some useful version information. It shows the version of the Java runtime being used and the version of the portion of *J/Link* that is written in Java. The version of the *J/Link* Java component should match the version of the *J/Link Mathematica* component.

Controlling the Class Path: How J/Link Finds Classes

The Java Class Path

The class path tells the Java runtime, compiler, and other tools where to find third-party and user-defined classes—classes that are not Java "extensions" or part of the Java platform itself. The class path has always been a source of confusion among Java users and programmers.

Java can find classes that are part of the standard Java platform (so-called "bootstrap" classes), classes that use the so-called "extensions" mechanism, and classes on the class path, which is controlled by the CLASSPATH environment variable or by command-line options when Java is launched. *J/Link* can load and use any classes that the Java runtime can find through these normal mechanisms. In addition, *J/Link* can find classes, resources, and native libraries that are in a set of extra locations, beyond what is specified on the class path at startup. This set of extra locations can be added to while Java is running.

J/Link provides two ways to alter the search path Java uses to find classes. The first way is via the ClassPath option to ReinstallJava. The second way, which is superior to modifying the class path at startup, is to add new directories and jar files to the special set of extra locations that *J/Link* searches. These two methods will be described in the next two subsections.

Overriding the Startup Class Path

For a class to be accessible via the standard Java class path, one of the following must apply:

- It is inside a .zip or .jar file that is itself named on the class path.
- It is a loose class file that is in an appropriately nested directory beneath a directory that is on the class path.

"Appropriately nested" means that the class file must be in a directory whose hierarchy mirrors the full package name of the class. For example, assume that the directory c:\MyClasses is on the class path. If you have a class that is not in a package (there is no package statement at the beginning of the code), its class file should be put directly into c:\MyClasses. If you have a class that is in the package `com.acme.stuff`, its class file would need to be in the directory c:\MyClasses\com\acme\stuff. Note that jar and zip files must be explicitly named on the class path—you cannot just toss them into a directory that is itself named on the class path. Directory issues are not relevant for jar and zip files, meaning that regardless of how hierarchically organized the classes inside a jar file are, you simply name the jar file itself on the class path and all the classes inside it can be found.

If you want to specify paths for classes that are not part of the standard Java platform or extensions, you can use the `ClassPath` option to `ReinstallJava`. The value that you supply for the `ClassPath` option is a string that names the desired directories and zip or jar files. This string is platform-dependent; the paths are specified in the native style for your platform, and the separator character is a colon on Unix and a semicolon on Windows. Here are typical specifications:

```
ReinstallJava[ClassPath → "c:\\MyJavaDir\\MyPackage.jar;c:\\MyJavaDir"]
(* Windows *)

ReinstallJava[ClassPath → "~/MyJavaDir/MyPackage.jar:~/MyJavaDir"]
(* Unix *)
```

The default setting for `ClassPath` is `Automatic`, which means to use the value of the `CLASSPATH` environment variable. If you set `ClassPath` to something else, then J/Link will ignore the `CLASSPATH` environment variable—it will not be able to find those classes. In other words, if you use a `ClassPath` specification, you lose the `CLASSPATH` environment variable. This is similar to the behavior of the `-classpath` command-line option to the Java runtime and compiler, if you are familiar with those tools.

It is recommended that users avoid the ClassPath option. If you need the dynamic control that the ClassPath option provides, you should use the more powerful and convenient AddToClassPath mechanism, described in the next section. The most common reason for using the ClassPath option is if you want to specifically prevent the contents of the CLASSPATH environment variable from being used. To do this, set ClassPath -> None.

Dynamically Modifying the Class Path

One thing that is inconvenient about the standard Java class path is that it cannot be changed after the Java runtime has been launched. *J/Link* has its own class loader that searches in a set of special locations beyond the standard Java class path. This gives *J/Link* an extremely powerful and flexible means of finding classes. To add locations to this extra set, use the AddToClassPath function.

AddToClassPath["*location*",...]	add the specified directories or jar files to *J/Link*'s class search path

Adding classes to the search path.

After Java has been started, you can call AddToClassPath whenever you wish, and it will take effect immediately. One convenient feature of this extra class search path is that if you add a directory, then any jar or zip files in that directory will be searched. This means that you do not have to name jar files individually, as you need to do with the standard Java class path. For loose class files, the nesting rules are the same as for the class path, meaning that if a class is in the package com.acme.stuff, and you called AddToClassPath["d:\\myClasses"], then you would need to put the class file into d:\MyClasses\com\acme\stuff.

Changes to the search path that you make with AddToClassPath only apply to the current Java session. If you quit and restart java, you will need to call AddToClassPath again.

In addition to the locations you add yourself with AddToClassPath, *J/Link* automatically includes any Java subdirectories of any directories in the standard *Mathematica* application locations ($UserBaseDirectory/AddOns/Applications, $BaseDirectory/AddOns/Applications, < *Mathematica dir* >/AddOns/Applications, and < *Mathematica dir* >/AddOns/ExtraPackages). This feature is designed to provide extremely easy deployment for developers who create applications for *Mathematica* that use Java and *J/Link* for part of their implementation. This is described in "Deploying Applications that use *J/Link*" in more detail, but even casual Java pro-

grammers who are writing classes to use with *J/Link* can take advantage of it. Just create a subdirectory of AddOns/Applications, say MyStuff, create a Java subdirectory within it, and toss class or jar files into it. *J/Link* will be able to find and use them. Of course, loose class files have to be placed into an appropriately nested subdirectory of the Java directory, corresponding to their package names (if any), as described.

The `AddToClassPath` function was introduced in *J/Link* 2.0. Previous versions of *J/Link* had a variable called `$ExtraClassPath` that specified a list of extra locations. You could add to this list like this:

```
AppendTo[$ExtraClassPath, "d:\\MyClasses"];
```

`$ExtraClassPath` was deprecated in *J/Link* 2.0, but it still works. One advantage of `$ExtraClassPath` over using `AddToClassPath` is that changes made to `$ExtraClassPath` persist across a restart of the Java runtime.

Examining the Class Path

The `JavaClassPath` function returns the set of directories and jar files in which *J/Link* will search for classes. This includes all locations added with `AddToClassPath` or `$ExtraClassPath`, as well as Java subdirectories of application directories in any of the standard *Mathematica* application locations. It does not display the jar files that make up the standard Java platform itself, or jar files in the Java extensions directory. Those classes can always be found by Java programs.

`JavaClassPath[]`	gives the complete set of directories and jar files in which *J/Link* will search for classes

Inspecting the class search path.

Using *J/Link*'s Class Loader Directly

As stated earlier, *J/Link* uses its own class loader to allow it to find classes and other resources in a dynamic set of locations beyond the startup class path. Essentially all the classes that you load using *J/Link* that are not part of the Java platform itself will be loaded by this class loader. One consequence of this is that calling Java's `Class.forName()` method from *Mathematica* will often not work.

```
LoadJavaClass["java.lang.Class"];
cls = Class`forName["some.class.that.only.JLink.can.find"]
```

```
Java::excptn : A Java exception occurred: java.lang.ClassNotFoundException:
    some.class.that.only.JLink.can.find
      at java.net.URLClassLoader$1.run(Unknown Source)
      at java.security.AccessController.doPrivileged(Native Method)
      at java.net.URLClassLoader.findClass(Unknown Source)
      at java.lang.ClassLoader.loadClass(Unknown Source)
      at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
      at java.lang.ClassLoader.loadClass(Unknown Source)
      at java.lang.ClassLoader.loadClassInternal(Unknown Source)
      at java.lang.Class.forName0(Native Method)
      at java.lang.Class.forName(Unknown Source)
      at
    sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
      at
    sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source).
$Failed
```

The problem is that `Class.forName()` finds classes using a default class loader, not the *J/Link* class loader, and this default class loader does not know about the special directories in which *J/Link* looks for classes (in fact, it does not even know about the startup class path, because of details of how *J/Link* launches Java). If you are translating Java code into *Mathematica*, or if you just want to get a `Class` object for a given class, watch out for this problem. The fix is to force *J/Link*'s class loader to be used. One way to do this is to use the three-argument form of `Class.forName()`, which allows you to specify the class loader to be used:

```
LoadJavaClass["com.wolfram.jlink.JLinkClassLoader"];
cls = Class`forName["some.class.that.only.JLink.can.find",
  True, JLinkClassLoader`getInstance[]]
```

An easier way is to use the static `classFromName` method of `JLinkClassLoader`:

```
cls = JLinkClassLoader`classFromName["some.class.that.only.JLink.can.find"]
```

You should think of this `classFromName()` method as being the replacement for `Class.forName()`. When you find yourself wanting to obtain a `Class` object from a class name given as a string, remember to use `JLinkClassLoader.classFromName()`.

`Class.forName()` is not very commonly found in Java code. One reason it is used is when an object needs to be created, but its class was not known at compile time. For example, the class name might come from a preferences file or be determined programmatically in some other way. Often, the very next line creates an instance of the class, like this:

```
// Java code
Class cls = Class.forName("SomeClassThatImplementsInterfaceX");
X obj = (X) cls.newInstance();
```

If you are translating code like this into a *Mathematica* program, this operation can be performed simply by calling `JavaNew`:

```
obj = JavaNew["SomeClassThatImplementsInterfaceX"]
```

The point here is that for a very common usage of `Class.forName()`, you do not have to translate it line-by-line into *Mathematica*—you can duplicate the functionality by calling `JavaNew`.

Performance Issues

Overhead of Calls to Java

The speed of Java programs is highly dependent on the Java runtime. On certain types of programs, for example, ones that spend most of their time in a tight number-crunching loop, the speed of Java can approach that of compiled, optimized C.

Java is a good choice for computationally intensive programs. Your mileage may vary, but do not rule out Java for any type of program before you have done some simple speed testing. For less demanding programs, where every ounce of speed is not necessary, the simplicity of using *J/Link* instead of programming traditional *MathLink* "installable" programs with C makes Java an obvious choice.

The speed issues with *J/Link* are not, for the most part, the speed of Java execution. Rather, the bottleneck is the rate at which you can perform calls into Java, which is itself limited mainly by the speed of *MathLink* and the processing that must be done in *Mathematica* for each call into Java. The maximum rate of calls into Java is highly dependent on which operating system and which Java runtime you use. A fast Windows machine can perform more than 5000 Java method calls per second, and considerably more if they are static methods, which require less preprocessing in *Mathematica*. On some operating systems the results will be less. You should keep in mind that there is a more or less fixed cost of a call into Java regardless of what the call

does, and on slow machines this cost could be as much as .001 seconds. Many Java methods will execute in considerably less time than this, so the total time for the call is often dominated by the fixed turnaround time of a *J/Link* call, not the speed of Java itself.

For most uses, the overhead of a call into Java is not a concern, but if you have a loop that calls into Java 500,000 times, you will have a problem (unless your program takes so long that the *J/Link* cost is negligible, in which case you have an even bigger problem!). If your *Mathematica* program is structured in a way that requires a great many calls into Java, you may need to refactor it to do more on the Java side and thus reduce the number of times you need to cross the Java-*Mathematica* boundary. This will probably involve writing some Java code, which unfortunately defeats the *J/Link* premise of being able to use *Mathematica* to script the functionality of an arbitrary Java program. There are uses of Java that just cannot be feasibly scripted in this way, and for these you will need to write more of the functionality in Java and less in *Mathematica*.

Speeding Up Sending Large Arrays

You can send and receive arrays of most "primitive" Java types (e.g., `byte`, `short`, `int`, `float`, `double`) nearly as fast as in a C-language program. The set of types that can be passed quickly corresponds to the set of types for which the *MathLink* C API has single functions to put arrays. The Java types `long` (these are 64 bits), `boolean`, and `String` do not have fast *MathLink* functions, and so sending or receiving these types is much slower. Try to avoid using extremely large arrays of these types (say, more than 100,000 elements) if possible.

A setting that has a big effect on the speed of moving multidimensional arrays is the one used to control whether "ragged" arrays are allowed. As discussed in "Ragged Arrays", the default behavior of *J/Link* is to require that all arrays be fully rectangular. But Java does not require that arrays conform to this restriction, and if you want to send or receive ragged arrays, you can call `AllowRaggedArrays[True]` in your *Mathematica* session. This causes *J/Link* to switch to a much slower method for reading and writing arrays. Avoid using this setting unless you need it, and switch it off as soon as you no longer require it.

When you load a class with a method that takes, say, an `int[][]`, the definition in *Mathematica* that *J/Link* creates for calling this method uses a pattern test that requires its argument to be a two-dimensional array of integers. If the array is quite large, say on the order of 500 by 500, this test can take a significant amount of time, probably similar to the time it takes to

actually transfer the array to Java. If you want to avoid the time taken by this testing of array arguments, you can set the variable $RelaxedTypeChecking to True. If you do this, you are on your own to ensure that the arrays you send are of the right type and dimensionality. If you pass a bad array, you will get a *MathLink* error, but this will not cause any problems for *J/Link* (other than that the call will return $Failed).

You probably do not want to leave $RelaxedTypeChecking set to True for a long time, and if you are writing code for others to use you certainly do not want to alter its value in their session. $RelaxedTypeChecking is intended to be used in a Block construct, where it is given the value of True for a short period:

```
Block[{$RelaxedTypeChecking = True}, obj[meth[someLargeArray]]]
```

$RelaxedTypeChecking only has an effect for arrays, which are the only types for which the pattern test that *J/Link* creates is expensive relative to the actual call into Java.

Another optimization to speed up *J/Link* programs is to use ReturnAsJavaObject to avoid unnecessary passing of large arrays or strings back and forth between *Mathematica* and Java. ReturnAsJavaObject is discussed in the section "ReturnAsJavaObject".

An Optimization Example

Next examine a simple example of steps you might take to improve the speed of a *J/Link* program. Java has a powerful DecimalFormat class you can use to format *Mathematica* numbers in a desired way for output to a file. Here you create a DecimalFormat object that will format numbers to exactly four decimal places.

```
fmt = JavaNew["java.text.DecimalFormat", "#.0000"];
```

To use the fmt object, you call its format() method, supplying the number you want formatted.

```
fmt@format[12.34]
12.3400
```

This returns a string with the requested format. Now suppose you want to use this ability to format a list of 20000 numbers before writing them to a file.

```
data = Table[Random[], {40 000}];

Map[fmt@format[#] &, data];
```

The Map call, which invokes the format method 40000 times, takes 46 seconds on a certain PC (this is wall clock time, not the result of the Timing function, which is not accurate for *MathLink* programs on most systems). Clearly this is not acceptable. As a first step, you try using MethodFunction because you are calling the same method many times.

```
methodFunc = MethodFunction[fmt, format];
```

Note that you use fmt as the first argument to MethodFunction. The first argument merely specifies the class; as with virtually all functions in *J/Link* that take a class specification, you can use an object of the class if you desire. The MethodFunction that is created can be used on any object of the DecimalFormat class, not just the fmt object.

```
Map[methodFunc[fmt, #] &, data];
```

Using methodFunc, this now takes 36 seconds. There is a slight speed improvement, much less than in earlier versions of *J/Link*. This means you are getting about 1100 calls per second, and it is still not fast enough to be useful. The only thing to do is to write your own Java method that takes an *array* of numbers, formats them all, and returns an array of strings. This will reduce the number of calls from *Mathematica* into Java 40000 down to one.

Here is the code for the trivial Java class necessary. Note that there is nothing about this code that suggests it will be called from *Mathematica* via *J/Link*. This is exactly the same code you would write if you wanted to use this functionality within Java.

```
public class FormatArray {
    public static String[] format(java.text.DecimalFormat fmt,double[] d) {
        String[] result=new String[d.length];
        for (int i = 0; i < d.length; i++)
            result[i] = fmt.format(d[i]);
        return result;
    }
}
```

This new version takes less than 2 seconds.

```
LoadJavaClass["FormatArray"];
FormatArray`format[fmt, data];
```

Reference Counts and Memory Management

Object References in *Mathematica*

The earlier treatment of JavaObject expressions avoided discussing deeper issues such as reference counts and uniqueness. Every time a Java object reference is returned to *Mathematica*, either as a result of a method or field or an explicit call to JavaNew, *J/Link* looks to see if a reference to this object has been sent previously in this session. If not, it creates a JavaObject expression in *Mathematica* and sets up a number of definitions for it. This is a comparatively time-consuming process. If this object has already been sent to *Mathematica*, in most cases *J/Link* simply creates a JavaObject expression that is identical to the one created previously. This is a much faster operation.

There are some exceptions to this last rule, meaning that sometimes when an object is returned to *Mathematica* a new and different JavaObject expression is created for it, even though this same object has previously been sent to *Mathematica*. Specifically, any time an object's hashCode() value has changed since the last time it was seen in *Mathematica*, the JavaObject expression created will be different. You do not really need to be concerned with the details of this, except to remember that SameQ is not a valid way to compare JavaObject expressions to decide whether they refer to the same object. You must use the SameObjectQ function.

SameObjectQ[*obj1*,*obj2*]	return True if the JavaObject expressions *obj1* and *obj2* refer to the same Java object, False otherwise

Comparing JavaObject expressions.

Here is an example.

```
pt = JavaNew["java.awt.Point", 1, 1]
```

```
«JavaObject[java.awt.Point]»
```

The variable pt refers to a Java Point object. Now put it into a container so you can get it back out later.

```
vec = JavaNew["java.util.Vector"];
vec@add[pt];
```

Now change the value of one of its fields. For a Point object, changing the value of one of its fields changes its hashCode() value.

```
pt@x = 2;
```

Now you compare the JavaObject expression given by pt and the JavaObject expression created when you ask for the first element of the Vector to be returned to *Mathematica*. Even though these are both references to the same Java object, the JavaObject expressions are different.

```
pt === vec@elementAt[0]
False
```

Because you cannot use SameQ (===) to decide whether two object references in *Mathematica* refer to the same Java object, *J/Link* provides a function, SameObjectQ, for this purpose.

```
SameObjectQ[pt, vec@elementAt[0]]
True
```

You may be wondering why the SameObjectQ function is necessary. Why not just call an object's equals() method? It certainly gives the correct result for this example.

```
pt@equals[vec@elementAt[0]]
True
```

The problem with this technique is that equals() does not always compare object references. Any class is free to override equals() to provide any desired behavior for comparing two objects of that class. Some classes make equals() compare the "contents" of the objects, such as the String class, which uses it for string comparison. Java provides two distinct equality operations, the == operator and the equals() method. The == operator always compares references, returning true if and only if the references point to the same object, but equals() is often overridden for some other type of comparison. Because there is no method call in Java that mimics the behavior of the language's == operator as applied to object references, *J/Link* needs a SameObjectQ function that provides that behavior for *Mathematica* programmers.

In an unusual case where you need to compare object references for equality a very large number of times, the comparative slowness of SameObjectQ compared to SameQ could become an issue. The only thing that could cause two JavaObject expressions that refer to the exact same Java object to be not SameQ is if the hashCode() value of the object changed between the times that the two JavaObject expressions were created. If you know this has not happened, then you can safely use SameQ as the test whether they refer to the same object.

ReleaseJavaObject

The Java language has a built-in facility called "garbage collection" for freeing up memory occupied by objects that are no longer in use by a program. Objects become eligible for garbage collection when no references to them exist anywhere, except perhaps in other objects that are also unreferenced. When an object is returned to *Mathematica*, either as a result of a call to `JavaNew` or as the return value of a method call or field access, the *J/Lin*k code holds a special reference to the object on the Java side to ensure that it cannot be garbage-collected while it is in use by *Mathematica*. If you know that you no longer need to use a given Java object in your *Mathematica* session, you can explicitly tell *J/Link* to release its reference. The function that does this is `ReleaseJavaObject`. In addition to releasing the *Mathematica*-specific reference in Java, `ReleaseJavaObject` clears out internal definitions for the object that were created in *Mathematica*. Any subsequent attempt to use this object in *Mathematica* will fail.

```
frm = JavaNew["java.awt.Frame"]
«JavaObject[java.awt.Frame] »
```

Now tell Java that you no longer need to use this object from *Mathematica*.

```
ReleaseJavaObject[frm]
```

It is now an error to refer to `frm`.

`ReleaseJavaObject[`*obj*`]`	let Java know that you are done using *obj* in *Mathematica*
`ReleaseObject[`*obj*`]`	deprecated; replaced by `ReleaseJavaObject` in *J/Link* 2.0
`JavaBlock[`*expr*`]`	all novel Java objects returned to *Mathematica* during the evaluation of *expr* will be released when *expr* finishes
`BeginJavaBlock[]`	all novel Java objects returned to *Mathematica* between now and the matching `EndJavaBlock[]` will be released
`EndJavaBlock[]`	release all novel objects seen since the matching `BeginJavaBlock[]`
`LoadedJavaObjects[]`	return a list of all objects that are in use in *Mathematica*
`LoadedJavaClasses[]`	return a list of all classes loaded into *Mathematica*

J/Link memory management functions.

Calling `ReleaseJavaObject` will not necessarily cause the object to be garbage-collected. It is quite possible that other references to it exist in Java. `ReleaseJavaObject` does not tell Java to throw the object away, only that it does not need to be kept around solely for *Mathematica*'s sake.

An important fact about the references that *J/Link* maintains for objects sent to *Mathematica* is that only one reference is kept for each object, no matter how many times it is returned to *Mathematica*. It is your responsibility to make sure that after you call `ReleaseJavaObject`, you never attempt to use that object through any reference that might exist to it in your *Mathematica* session.

```
frm = JavaNew["java.awt.Frame"];
b1 = JavaNew["java.awt.Button"];
```

The `add()` method of the `Frame` class returns the object added, so `b2` refers to the same object as `b1`:

```
b2 = frm@add[b1];
```

If you call `ReleaseJavaObject[b1]`, it is not the *Mathematica* symbol `b1` that is affected, but the Java object that `b1` refers to. Therefore, using `b2` is also an error (or any other way to refer to this same `Button` object, such as `%`).

Calling `ReleaseJavaObject` is often not necessary in casual use. If you are not making heavy use of Java in your session then you will usually not need to be concerned about keeping track of what objects may or may not be needed anymore—you can just let them pile up. There are special times, though, when memory use in Java will be important, and you may need the extra control that `ReleaseJavaObject` provides.

JavaBlock

`ReleaseJavaObject` is provided mainly for developers who are writing code for others to use. Because you can never predict how your code will be used, developers should always be sure that their code cleans up any unnecessary references it creates. Probably the most useful function for this is `JavaBlock`.

`JavaBlock` automates the process of releasing objects encountered during the evaluation of an expression. Often, a *Mathematica* program will need to create some Java objects with `JavaNew`, operate with them, perhaps causing other objects to be returned to *Mathematica* as the results of method calls, and finally return some result such as a number or string. Every Java object encountered by *Mathematica* during this operation is needed only during the lifetime of the program, much like the local variables provided in *Mathematica* by `Block` and `Module`, and in C, C++, Java, and many other languages by block scoping constructs (e.g., {}). `JavaBlock` allows you to mark a block of code as having the property that any new objects returned to *Mathematica* during the evaluation are to be treated as temporary, and released when `JavaBlock` finishes.

It is important to note that the preceding sentence said "new objects". `JavaBlock` will not cause every object encountered during the evaluation to be released, only those that are being encountered for the first time. Objects that have already been seen by *Mathematica* will not be affected. This means that you do not have to worry that `JavaBlock` will aggressively release an object that is not truly temporary to that evaluation.

It is not enough simply to call `ReleaseJavaObject` on every object you create with `JavaNew`, because many Java method calls return objects. You may not be interested in these return values, or you may never assign them to a named variable because they may be chained together with other calls (as in `obj@returnsObject[]@foo[]`), but you still need to release them. Using `JavaBlock` is an easy way to be sure that all novel objects are released when a block of code finishes.

`JavaBlock[`*expr*`]` returns whatever *expr* returns.

Many *J/Link Mathematica* programs will have the following structure:

```
MyFunc[args__] :=
     JavaBlock[
          Module[{locals},
               ...
           ]
     ]
```

It is very common to write a function that creates and manipulates a number of `JavaObject` expressions, and then returns one of them, the rest being temporary. To facilitate this, if the return value of a `JavaBlock` is a single `JavaObject`, it will not be released.

```
MyOtherFunc[args__] :=
    JavaBlock[
        Module[{obj},
            ...
            obj = JavaNew["java.awt.Frame"];
            ...
            Return[obj]
   (* OK: obj will not be released when JavaBlock finishes. *)
        ]
    ]
```

New in *J/Link* 2.1 is the `KeepJavaObject` function, which allows you to specify an object or sequence of objects that should not be released when the `JavaBlock` ends. Calling `KeepJavaObject` on a single object or sequence of objects means they will not be released when the first enclosing `JavaBlock` ends. If there is an outer enclosing `JavaBlock`, the objects will be freed when *it* ends, however, so if you want the objects to escape a nested set of `JavaBlock` expressions, you must call `KeepJavaObject` at each level. Alternatively, you can call

KeepJavaObject[*obj*, Manual], where the Manual argument tells *J/Link* that the object should not be released by any enclosing JavaBlock expressions. The only way such object will be released is if you manually call ReleaseJavaObject on it. Here is an example that uses KeepJavaObject to allow you to return a list of two objects without them being released:

```
MyOtherFunc[args__] :=
    Module[{obj1, obj2, obj3},
        JavaBlock[
            obj1 = JavaNew["java.awt.Frame"];
            obj2 = JavaNew["java.awt.Button"];
            obj3 = JavaNew["SomeTemporaryObject"];
            ...
            KeepJavaObject[obj1, obj2];
            {obj1, obj2}
        ]
     ]
```

BeginJavaBlock and EndJavaBlock can be used to provide the same functionality as JavaBlock across more than one evaluation. EndJavaBlock releases all novel Java objects returned to *Mathematica* since the previous matching BeginJavaBlock. These functions are mainly of use during development, when you might want to set a mark in your session, do some work, and then release all novel objects returned to *Mathematica* since that point. BeginJavaBlock and EndJavaBlock can be nested. Every BeginJavaBlock should have a matching EndJavaBlock, although it is not a serious error to forget to call EndJavaBlock, even if you have nested levels of them—you will only fail to release some objects.

LoadedJavaObjects and LoadedJavaClasses

LoadedJavaObjects[] returns a list of all Java objects that are currently referenced in *Mathematica*. This includes all objects explicitly created with JavaNew and all those that were returned to *Mathematica* as the result of a Java method call or field access. It does not include objects that have been released with ReleaseJavaObject or through JavaBlock. LoadedJavaObjects is intended mainly for debugging. It is very useful to call it before and after some function you are working on. If the list grows, your function leaks references, and you need to examine its use of JavaBlock and/or ReleaseJavaObject.

LoadedJavaClasses[] returns a list of JavaClass expressions representing all classes loaded into *Mathematica*. Like LoadedJavaObjects, LoadedJavaClasses is intended mainly for debugging. Note that you do not have to determine if a class has already been loaded before you call LoadJavaClass. If the class has been loaded, LoadJavaClass does nothing but return the appropriate JavaClass expression.

Exceptions

How Exceptions Are Handled

J/Link handles Java exceptions automatically. If an uncaught exception is thrown during any call into Java, you will get a message in *Mathematica*. Here is an example that tries to format a real number as an integer.

```
LoadClass["java.lang.Integer"];
Integer`parseInt["1234.5"]
```

```
Java::excptn :
    A Java exception occurred : java.lang.ArrayIndexOutOfBoundsException.
$Failed
```

If the exception is thrown before the method returns a result to *Mathematica*, as in the example, the result of the call will be `$Failed`. As discussed later in "Manually Returning a Result to *Mathematica*", it is possible to write your own methods that manually send a result to *Mathematica* before they return. In such cases, if an exception is thrown after the result is sent to *Mathematica* but before the method returns, you will get a warning message reporting the exception, but the result of the call will be unaffected.

If the Java code was compiled with debugging information included, the *Mathematica* message you get as a result of an exception will show the full stack trace to the point where the exception occurred, with the exact line numbers in each file.

The JavaThrow Function

In some cases, you may want to cause an exception to be thrown in Java. This can be done with the `JavaThrow` function. `JavaThrow` is new in *J/Link* 2.0 and should be considered experimental. Its behavior might change in future versions.

`JavaThrow[`*exceptionObj*`]`	throw the given exception object in Java

Throwing Java exceptions from *Mathematica.*

You will only want to use `JavaThrow` in *Mathematica* code that is itself called from Java. It is quite common for *J/Link* programs written in *Mathematica* to involve both calls from *Mathematica* into Java and calls from Java back to *Mathematica*. Such "callbacks" to *Mathematica* are used extensively in *Mathematica* programs that create Java user interfaces, as described in

detail later in the section "Creating Windows and Other User Interface Elements". For example, you can associate a *Mathematica* function to be called when the user clicks a Java button. This *Mathematica* function is called directly from Java, and you might want it to behave just like a Java method, including having the ability to throw Java exceptions.

An example of throwing an exception in a callback from a user interface action like clicking a button is not very realistic because there is typically nothing in Java to catch such exceptions; thus they are essentially ignored. A more meaningful example would be a program that involved a mix of Java and *Mathematica* code where, for flexibility and ease of development reasons, you have a *Mathematica* function being called to implement the "guts" of a Java method that can throw an exception. As a concrete example, say you are doing XML processing with Java and *Mathematica* using the SAX (Simple API for XML) API. SAX processing is based on a set of handler methods that are called as certain events occur during parsing of the XML document. Each such method can throw a `SAXException` to indicate an error and halt the parsing. You want to implement these handler methods in *Mathematica* code, and thus you want a way to throw a `SAXException` from *Mathematica*. Here is a hypothetical example of one such handler method, the `startDocument()` method, which is invoked by the SAX engine when document processing starts:

```
startDocument[] :=
 If[! readyToAcceptParsingEvents, JavaThrow[JavaNew["org.xml.sax.SAXException",
    "Mathematica code has not been initialized"]]]
```

After a call to `JavaThrow`, the rest of the *Mathematica* function executes normally, but there is no result returned to Java.

Returning Objects "by Value" and "by Reference"

References and Values

J/Link provides a mapping between certain *Mathematica* expressions and their Java counterparts. What this means is that these *Mathematica* expressions are automatically converted to and from their Java counterparts as they are passed between *Mathematica* and Java. For example, Java integer types (`long`, `short`, `int`, and so on) are converted to *Mathematica* integers and Java real types (`float` and `double`) are converted to *Mathematica* real numbers. Another mapping is that Java objects are converted to `JavaObject` expressions in *Mathematica*. These `JavaObject` expressions are *references* to Java objects—they have no meaning in *Mathematica* except as they are manipulated by *J/Link*. However, some Java objects are things that have meaningful values in *Mathematica*, and these objects are by default converted to values. Examples of such objects are strings and arrays.

You could say, then, that Java objects are by default returned to *Mathematica* "by reference", except for a few special cases. These special cases are strings, arrays, complex numbers (discussed later), `BigDecimal` and `BigInteger` (discussed later), and the "wrapper" classes (e.g., `java.lang.Integer`). You could say that these exceptional cases are returned "by value". The table in "Conversion of Types between Java and *Mathematica*" shows how these special Java object types are mapped into *Mathematica* values.

In summary, every Java object that has a meaningful value representation in *Mathematica* is converted into this value, simply because that is the most useful behavior. There are times, however, when you might want to override this default behavior. Probably the most common reason for doing this is to avoid unnecessary traffic of large expressions over *MathLink*.

`ReturnAsJavaObject[`*expr*`]`	a Java object returned by *expr* will be in the form of a reference
`ByRef[`*expr*`]`	deprecated; replaced by `ReturnAsJavaObject` in *J/Link* 2.0
`JavaObjectToExpression[`*obj*`]`	give the value of the Java object *obj* as a *Mathematica* expression
`Val[`*obj*`]`	deprecated; replaced by `JavaObjectToExpression` in *J/Link* 2.0

"By reference" and "by value" control.

ReturnAsJavaObject

Consider the case where you have a static method in class `MyClass` called `arrayAbs()` that takes an array of doubles and returns a new array where each element is the absolute value of the corresponding element in the argument array. The declaration of this method thus looks like `double[] arrayAbs(double[] a)`. This is how you would call such a method from *Mathematica*.

```
LoadJavaClass["MyClass", StaticsVisible → True];
arrayAbs[{1., -2., 3., 4.}]
```

```
{1., 2., 3., 4.}
```

The example showed how you probably want the method to work: you pass a *Mathematica* list and get back a list. Now assume you have another method named `arraySqrt()` that acts like `arrayAbs()` except that it performs the `sqrt()` function instead of `abs()`.

```
arraySqrt[arrayAbs[{1., -2., 3., 4.}]]
```

```
{1., 1.41421, 1.73205, 2.}
```

In this computation, the original list is sent over *MathLink* to Java and a Java array is created with these values. That array is passed as an argument to `arrayAbs()`, which itself creates and returns another array. This array is then sent back to *Mathematica* via *MathLink* to create a list, which is then promptly sent back to Java as the argument for `arraySqrt()`. You can see that it was a waste of time to send the array data back to *Mathematica*—you had a perfectly good array (the one returned by the `arrayAbs()` method) living on the Java side, ready to be passed to `arraySqrt()`, but instead you sent its contents back to *Mathematica* only to have it immediately come back to Java again as a new array with the same values! For this example, the cost is negligible, but what if the array has 200,000 elements?

What is needed is a way to let the array data remain in Java and return only a reference to the array, not the actual data itself. This can be accomplished with the `ReturnAsJavaObject` function.

```
ReturnAsJavaObject[arrayAbs[{1., -2., 3., 4.}]]
«JavaObject[[D] »
```

Note that the class name of the `JavaObject` is `"[D"`, which, although a bit cryptic, is the actual Java class name of a one-dimensional array of doubles. Here is how the computation looks using `ReturnAsJavaObject`:

```
arraySqrt[ReturnAsJavaObject[arrayAbs[{1., -2., 3., 4.}]]]
{1., 1.41421, 1.73205, 2.}
```

Earlier you saw `arraySqrt()` being called with an argument that was a *Mathematica* list of reals. Here it is being called with a reference to a Java object that is a one-dimensional array of doubles. All methods and fields that take an array can be called from *Mathematica* with either a *Mathematica* list or a reference to a Java array of the appropriate type.

Strings are the other type for which `ReturnAsJavaObject` is useful. Like arrays, strings have the two properties that (1) they are represented in Java as objects but also have a meaningful *Mathematica* value, and (2) they can be large, so it is useful to be able to avoid passing their data back and forth unnecessarily. As an example, say your class `MyClass` has a static method that appends to a string a digit taken from an external device that you are controlling from Java. It takes a string and returns a new one, so its signature is `static String appendDigit (String s)`. You have a *Mathematica* variable named `veryLongString` that holds a long string, and you want to append to this string 100 times. This code will cause the string contents to make 100 round trips between *Mathematica* and Java.

```
Do[veryLongString = appendString[veryLongString], {100}];
```

Using ReturnAsJavaObject lets the strings remain on the Java side, and thus it generates virtually no *MathLink* traffic.

```
Do[veryLongString = ReturnAsJavaObject[appendString[veryLongString]], {100}];
```

This example is somewhat contrived, since repeatedly appending to a growing string is not a very efficient style of programming, but it illustrates the issues.

When the Do loop is executed, veryLongString gets assigned values that are not *Mathematica* strings, but JavaObject expressions that refer to strings residing in Java. That means that appendString () gets called with a *Mathematica* string the very first iteration, but with a JavaObject expression thereafter. As is the case with arrays, any Java method or field that takes a string can be called in *Mathematica* either with a string or a JavaObject expression that refers to one. The veryLongString variable started out holding a string, but at the end of the loop it holds a JavaObject expression.

```
veryLongString
«JavaObject[java.lang.String] »
```

At some point, you probably want an actual *Mathematica* string, not this string object reference. How do you get the value back? You will visit this example again later when the JavaObjectToExpression function is introduced.

In summary, the ReturnAsJavaObject function causes methods and fields that return objects that would normally be converted into *Mathematica* values to return references instead. It is an optimization to avoid unnecessarily passing large amounts of data between *Mathematica* and Java, and as such it will be useful primarily for very large arrays and strings. As with all optimizations, you should not concern yourself with ReturnAsJavaObject unless you have some code that is running at an unacceptable speed, or you know ahead of time that the code you are writing will benefit measurably from it. Objects of most Java classes have no meaningful "by value" representation in *Mathematica*, and they are always returned "by reference". ReturnAsJavaObject will have no effect in these cases.

Finally, note that ReturnAsJavaObject has no effect on methods in which the Java programmer manually sends the result back to *Mathematica* (this topic is discussed later in this User Guide). Manually returning a result bypasses the normal result-handling routines in *J/Link*, so there is no chance for the ReturnAsJavaObject request to be accommodated.

JavaObjectToExpression

In the previous section, you saw how the `ReturnAsJavaObject` function can be used to cause objects normally returned to *Mathematica* by value to be returned by reference. It is necessary to have a function that does the reverse—takes a reference and converts it to its value representation. That function is `JavaObjectToExpression`.

Returning to the earlier `appendString` example, you used `ReturnAsJavaObject` to avoid costly passing of string data back and forth over *MathLink*. The result of this was that the `veryLongString` variable now held a `JavaObject` expression, not a literal *Mathematica* string. `JavaObjectToExpression` can be used to get the value of this string object as a *Mathematica* string.

```
JavaObjectToExpression[veryLongString]
```

```
037118086362644534489492294928989287822791948284089742269122236592851667829700627394053209887
   2893368
```

The majority of Java objects have no meaningful value representation in *Mathematica*. These objects can only be represented in *Mathematica* as `JavaObject` expressions, and using `JavaObjectToExpression` on them has no effect.

The `ReturnAsJavaObject` function is not the only way to get a `JavaObject` expression for an object that is normally returned to *Mathematica* as a value. The `JavaNew` function always returns a reference.

```
JavaNew["java.lang.String", "a string"]
```

```
«JavaObject[java.lang.String] »
```

```
JavaObjectToExpression[%]
```

```
a string
```

The next section introduces the `MakeJavaObject` function, which is easier than using `JavaNew` to construct Java objects out of *Mathematica* strings and arrays.

MakeJavaObject and MakeJavaExpr

Preamble

In addition to `JavaNew`, which calls a class constructor, *J/Link* provides two convenience functions for creating Java objects from *Mathematica* expressions. These functions are

MakeJavaObject and MakeJavaExpr. Do not get them confused, despite their similar names. MakeJavaObject is a commonly used function for constructing objects of some special types. MakeJavaExpr is an advanced function that creates an object of *J/Link*'s Expr class representing an arbitrary *Mathematica* expression.

MakeJavaObject

MakeJavaObject[*val*]	construct an object of the appropriate type to represent the *Mathematica* expression *val* (numbers, strings, lists, and so on)

MakeJavaObject.

When you call a Java method from *Mathematica* that takes, say, a Java String object, you can call it with a *Mathematica* string. The internals of *J/Link* will construct a Java string that has the same characters as the *Mathematica* string, and pass that string to the Java method. Sometimes, however, you want to pass a string to a method that is typed to take Object. You cannot call such a method from *Mathematica* with a string as the argument because although *J/Link* recognizes that a *Mathematica* string corresponds to a Java string, it does not recognize that a *Mathematica* string corresponds to a Java Object. It does this deliberately, for the sake of imposing as much type safety as possible on calls into Java. For this example, assume that the class MyClass has a method with the following signature:

```
void foo(Object obj);
```

Assume also that theObj is an object of this class, created with JavaNew. Try to call foo with a literal string.

```
theObj@foo["this is a string"]
```

```
Java::argxs :
    The method foo was called with an incorrect number or type of arguments.
$Failed
```

It fails for the reason given above. To call a Java method that is typed to take an Object with a string, you must first explicitly create a Java string object with the appropriate value. You can do this using JavaNew.

```
javaStr = JavaNew["java.lang.String", "this is a string"]
```

```
«JavaObject[java.lang.String] »
```

Now it works, because the argument is a JavaObject expression.

```
theObj@foo[javaStr]
```

To avoid having to call JavaNew to create a Java string object, *J/Link* provides the MakeJavaObject function.

```
javaStr = MakeJavaObject["this is a string"];
```

In the case of a string, MakeJavaObject just calls JavaNew for you. Of course, it would not be of much use if it could only construct String objects. The same issue arises with other Java objects that are direct representations of *Mathematica* values. This includes the "wrapper" classes like java.lang.Integer, java.lang.Boolean, and so on, and the array classes. If you want to call a Java method that takes a java.lang.Integer as an argument, you can call it from *Mathematica* with a raw integer. But if you want to pass an integer to a method that is typed to take an Object, you must explicitly create an object of type java.lang.Integer—*J/Link* will not construct one automatically from an integer argument. It is simpler to call MakeJavaObject than JavaNew for this.

```
MakeJavaObject[42]
```

```
«JavaObject[java.lang.Integer] »
```

When given an integer argument, MakeJavaObject always constructs a java.lang.Integer, never a java.lang.Short, java.lang.Long, or other "integer" Java wrapper object. Similarly, if you call MakeJavaObject with a real number, it creates a java.lang.Double, never a java.lang.Float. If you require an object of one of these other types, you will have to call JavaNew explicitly.

MakeJavaObject also works for Boolean values.

```
MakeJavaObject[True]
```

```
«JavaObject[java.lang.Boolean] »
```

If MakeJavaObject were only a shortcut for calling JavaNew, it would not be all that useful. It becomes indispensable, however, for creating objects of an array class. Recall that in Java, arrays are objects and they belong to a class. These classes have cryptic names, but if you know them you can create array objects with JavaNew. When creating array objects, the second argument to JavaNew is a list giving the length in each dimension. Here you create a 2×3 array of ints.

```
intArray2D = JavaNew["[[I", {2, 3}]
«JavaObject[[[I]
```

JavaNew lets us create array objects, but it does not let us supply initial values for the elements of the array. MakeJavaObject, on the other hand, takes a *Mathematica* list and converts it into a Java array object with the same values.

```
intArray2D = MakeJavaObject[{{1, 2, 3}, {4, 5, 6}}]
«JavaObject[[[I] »
```

Thus, MakeJavaObject is particularly useful for creating array objects, because it lets you supply the initial values for the array elements, and it frees you from having to learn and remember the names of the Java array classes ([[I for a two-dimensional array of ints, [D for a one-dimensional array of doubles, and so on). MakeJavaObject can create arrays up to three dimensions deep of integers, doubles, strings, Booleans, and objects.

The JavaObjectToExpression function is discussed in the section "JavaObjectToExpression", and you can think of MakeJavaObject as being the inverse of JavaObjectToExpression. MakeJavaObject takes a *Mathematica* expression that has a corresponding Java object that can represent its value, and creates that object. It literally "makes it into a Java object". The JavaObjectToExpression function goes the other way—it takes a Java object that has a meaningful *Mathematica* representation and converts it into that expression. It will always be the case that, for any x that MakeJavaObject can operate on,

```
JavaObjectToExpression[MakeJavaObject[x]] === x
```

Remember that MakeJavaObject is not a commonly used function. You do not need to explicitly construct Java objects from *Mathematica* strings, arrays, and so on, just to pass them to Java methods—*J/Link* does this automatically for you. But even though *J/Link* will create objects automatically from certain arguments in most circumstances, it will not do so when an argument is typed as a generic Object. Then you must create a JavaObject yourself, and MakeJavaObject is the easiest way to do this.

The code for the SetInternetProxy function discussed in the section SetInternetProxy provides a concrete example of using MakeJavaObject. To specify proxy information (for users behind firewalls), you need to set some system properties using the Properties class. This class is a subclass of Hashtable, so it has a method with the signature

```
Object put(Object key, Object value);
```

You should always specify keys and values for `Properties` in the form of strings. Thus, you might try this from *Mathematica*.

```
LoadJavaClass["java.lang.System"];
System`getProperties[]@put["proxySet", "true"]
```

```
Java::argx :
  Method named put defined in class java.util.Properties was called with
    an incorrect number or type of arguments. The
    arguments, shown here in a list, were {proxySet, true}.
```

```
$Failed
```

For this to work, you need to use `MakeJavaObject` to create Java `String` objects:

```
System`getProperties[]@put[MakeJavaObject["proxySet"], MakeJavaObject["true"]]
```

MakeJavaExpr

To understand the `MakeJavaExpr` function, you need to understand the motivation for *J/Link*'s `Expr` class, which is discussed in detail in "Motivation for the Expr Class". Basically, an `Expr` is a Java object that can represent an arbitrary *Mathematica* expression. Its main use is as a convenience for Java programmers who want to examine and operate on *Mathematica* expressions in Java. Sometimes it is useful to have a way of creating `Expr` objects in the *Mathematica* language instead of from Java. `MakeJavaExpr` is the function that fills this need.

`MakeJavaExpr[`*expr*`]`	construct an object of *J/Link*'s `Expr` class that represents the *Mathematica* expression

`MakeJavaExpr`.

Note that if you are calling a Java method that is typed to take an `Expr`, then you do not have to call `MakeJavaExpr` to construct an `Expr` object. *J/Link* will automatically convert any expression you supply as the argument to an `Expr` object, as it does with other automatic conversions. Like `MakeJavaObject`, `MakeJavaExpr` is used in cases where you are calling a method that takes a generic `Object`, not an `Expr`, and therefore *J/Link* will not perform any automatic conversion for you. In such cases you need to explicitly create an `Expr` object out of some *Mathematica* expression. One reason you might want to do this is to store a *Mathematica* expression in Java for retrieval later.

Here is a simple example of `MakeJavaExpr`. This demonstrates a few methods from the `Expr` class, which has a number of *Mathematica*-like methods for examining, modifying, and extracting portions of expressions. Of course, this is a highly contrived example—if you wanted to know the length of an expression you would just call *Mathematica*'s `Length[]` function. The `Expr` methods demonstrated here are typically called from Java, not *Mathematica*.

```
e = MakeJavaExpr[1 + 2 x + x^2]
```

```
«JavaObject[com.wolfram.jlink.Expr] »
```

```
e@length[]
```

```
3
```

```
e@part[3]
```

$$x^2$$

```
e@insert[x^3, -1]
```

$$1 + 2\,x + x^2 + x^3$$

Note that `Expr` objects, like *Mathematica* expressions, are immutable. The above call to `insert()` did not modify `e`; instead, it returned a new `Expr`.

```
JavaObjectToExpression[e]
```

$$1 + 2\,x + x^2$$

If you are having trouble understanding why you might want to use `MakeJavaExpr` in a *Mathematica* program, do not worry. It is an advanced function that few programmers will have any use for.

Creating Windows and Other User Interface Elements

Preamble

One of the most useful applications for *J/Link* is to write user interface elements for *Mathematica* programs. Examples of such elements would be a progress bar monitoring the completion of a computation, a window that displays an image or animation, a dialog box that prompts the user for input or helps them compose a proper call of an unfamiliar function, or a mini-application that leads the user through the steps of an analysis. These types of user interfaces are distinct from what you might write for a Java program that uses *Mathematica* in the background in that they "pop up" when the user invokes some *Mathematica* code. They do not replace the notebook front end, they just augment it. In this way, they are like an extension of the palettes and other specialty notebook elements you can create in the front end.

Mathematica with *J/Link* is an extremely powerful and productive environment for creating user interfaces. The complexity of user interface code is ideally suited to the interactive line-at-a-time nature of *J/Link* development. You can literally build, modify, and experiment with your user interface *while it is running*.

Anyone considering writing user interfaces for *Mathematica* programs should also look at *GUIKit* . *GUIKit* is built on top of *J/Link*, and provides an extremely high-level means of creating interfaces. Further discussion of *GUIKit* is beyond the scope of this manual, but be aware that *GUIKit* was specifically designed to provide an easier means of creating user interfaces than writing in "raw" *J/Link*, as described here.

Interactive and Non-Interactive Interfaces

To write *Mathematica* programs that create Java windows you need to understand important distinctions between several types of such user interfaces. These distinctions relate to how they interact with the *Mathematica* kernel.

At the highest level of categorization, there is a distinction between "interactive" and "non-interactive" interfaces. The interactiveness under consideration here is with the *Mathematica* kernel, not with the user. What we are calling non-interactive user interfaces have no need to communicate back to *Mathematica*, although they typically are controlled by *Mathematica*. Such interfaces often accept no user input at all—they are created, manipulated, and destroyed by *Mathematica* code. An example of this type is a window that shows a progress bar (a complete progress bar program is presented in "A Progress Bar"). A progress bar does not return a result to *Mathematica* and it does not need to respond to user actions, at least not by interacting with *Mathematica*. In other words, the window may go away when its close box is clicked (a user action), but this is not relevant to *Mathematica* because it does not return a result or trigger a call back into *Mathematica*. A progress bar is completely driven by a *Mathematica* program. The flow of information is in one direction only.

Such user interfaces typically have lifetimes that are encompassed by a single *Mathematica* program, as is the case with a progress bar. This is not required, however. Hosting an applet in its own window, as described in "Hosting Applets", is an example where the window lives on after the code that created it ends execution. The applet window is only dismissed when the user clicks in its close box. Again, though, the important property is that the applet does not need to interact with *Mathematica*.

This type of user interface, which requires no interaction back with *Mathematica*, poses no special issues that need to be discussed in this section. A program that creates, runs, and destroys such an interface is very much like a non-GUI Java computation that is accomplished with a series of calls into Java. It just happens to produce a visual effect. You can examine the progress bar code in "A Progress Bar" if you want to see a fully fleshed out example.

The more common "interactive" type of user interface needs to communicate back to *Mathematica*. This might be to return a result, like a typical modal input dialog, or to initiate a computation as a consequence of the user clicking a button. To understand the special problem this imposes, it is useful to examine some basic considerations about the kernel's "main loop", whereby it acquires input, evaluates it, and sends off any output.

When the *Mathematica* kernel is being used from the front end, it spends most of its life waiting for input to arrive on the *MathLink* that it uses to communicate with the front end. This *MathLink* is given by `$ParentLink`, and it is `$ParentLink` that has the kernel's "attention". When input arrives on `$ParentLink`, it is evaluated, any results are sent back on the link, and the kernel goes back to waiting for more input on `$ParentLink`. When *J/Link* is being used, the kernel has another *MathLink* open, the one that connects to the Java runtime. When you execute some code that calls into Java, the kernel sends something to Java and then blocks waiting for the return value from Java. During this period when the kernel is waiting for a return value from Java, the Java link has the kernel's attention. It is only during this period of time that the kernel is paying attention to the Java link. A more general way of saying this is that the kernel is only listening for input arriving from Java when it has been specifically instructed to do so. The rest of the time it is listening only to `$ParentLink`, which is typically the notebook front end.

Consider what happens when the user clicks on a button in your Java window and that button tries to execute some code that calls into *Mathematica*. The Java side sends something to *Mathematica* and then waits for the result, but the kernel will never get the request because it is not paying attention to the Java link. It is necessary to use some means to tell the kernel to look for input arriving on the Java link. *J/Link* provides three different ways to manage the kernel's attention to the Java link, and thereby control its readiness to accept requests for evaluations initiated by the Java side.

These three ways can be called "modal", "modeless", and "manual". In modal interaction, characterized by the use of the `DoModal` *Mathematica* function, the kernel is pointed at the Java link until the Java side releases it. The kernel is a complete slave to the Java side, and is unavailable for any other computations. In modeless interaction, characterized by the use of the `ShareKernel` *Mathematica* function, the kernel is kept in a state where it is receptive to evaluation requests arriving from either the notebook front end or Java, evenly sharing its attention between these two programs. Lastly, there is a manual mode, characterized by the use of the `ServiceJava` *Mathematica* function, which in some ways is intermediate between modal and modeless operation. Here, you manually instruct the kernel to allow single requests from Java while in the middle of running a larger program. The next few sections are devoted to further exploration of these types of user interfaces.

Before continuing, it is important to remember that all these issues about how to prepare the kernel for computations arriving from Java are only relevant for computations *initiated* in Java, typically by user actions like clicking a button. Calls from Java to *Mathematica* that are part of a back-and-forth series of calls that involve a call from *Mathematica* into Java are not a problem. Any time *Mathematica* has called into Java, *Mathematica* is actively listening for results arriving from Java. This may sound confusing, but that is mostly because it is only in a much later section that discusses writing your own Java methods to be called from *Mathematica*; such methods can call back to *Mathematica* for computations before they return their result (typical examples are to print something in the notebook window or display a message). These are true *callbacks* into *Mathematica*, and *Mathematica* is always ready to handle them. In contrast, calls to *Mathematica* that occur as the result of a user action in the Java side are, in effect, a surprise to *Mathematica*, and it is not normally in a state where it is ready to accept them.

Modal versus Modeless Operation

A common type of user interface element is like a modal dialog: once it is displayed, the *Mathematica* program hangs waiting for the user to dismiss the window. Typically, this is because the window returns a result to *Mathematica*, so it is not meaningful for *Mathematica* to continue until the window is closed. An example of such a window is a simple input window that asks the user for some value, which it returns to *Mathematica* when the **OK** button is clicked.

It is important to understand our slightly generalized use of the term "modal" to describe these windows. They may not be modal in the traditional sense that they must be dismissed before

anything else can be done in the user interface. Rather, they are modal with respect to the *Mathematica* kernel—the kernel cannot do anything else until they are closed. A Java window that you create might not be modal with respect to other Java windows on the screen (i.e., a dialog might not have the `isModal` property set), but it ties up the kernel's attention until it is dismissed.

Another type of user interface element is like a modeless dialog: after it is displayed, the *Mathematica* program that created it will finish, leaving the window visible and usable while the user continues working in the notebook front end. This sounds a lot like the first type of user interface element described earlier, but these windows are distinguished by the fact that they can initiate interactions with *Mathematica* while they are visible. An example would be a window that lets users load packages into *Mathematica* by selecting them from a scrolling list. You write a *J/Link* program that creates this window, displays it, and returns. The window is left open and usable until the user clicks in its close box. In the meantime, the user is free to continue working in the front end, going back to use this Java window whenever it is convenient.

Such a window is almost like another type of notebook or palette window in the front end. You can have any number of front end or Java windows open at once, and active, meaning that they can be used to initiate computations in *Mathematica*. They are each their own little interface onto the same kernel. What is different about the Java window is that it is much more general than a notebook window, and, importantly, it lives in a different application layer than the front end. This last fact makes the Java window in effect a second front end, rather than an extension of the notebook front end. To accommodate such a second front end, the kernel must be kept in a special state that allows it to handle requests for evaluations arriving from either the notebook front end or Java.

Before presenting examples of how to implement modal and modeless windows, it is necessary to jump ahead a little bit and explain the main mechanism by which Java user interface elements can communicate with *Mathematica*.

Handling Events with *Mathematica* Code: The "MathListener" Classes

User interface elements typically have active components like buttons, scrollbars, menus, and text fields that need to trigger some action when they are clicked. In the Java event model, components fire events in response to user actions, and other components indicate their interest in these events by registering as event listeners. In practice, though, components do not

usually act as event listeners directly. Instead, the programmer writes an adapter class that implements the desired event-listener interface and calls certain methods in the component in response to various events. This avoids having to subclass the responding component just to have it act as an event listener. The only specialty code goes into the adapter class, allowing the components that fire and respond to events to be generic.

As an example, say you are writing a standard Java program and you have a button that you want to use to control the appearance of a text area. Clicking the button should toggle between black text on a white background and white text on a black background. Buttons fire `ActionEvent`s when they are clicked, and a class that wants to receive notifications of clicks must implement the `ActionListener` interface, and register with the button by calling its `addActionListener` method. You would write a class, perhaps called `MyActionAdapter`, that implements `ActionListener`. In its `actionPerformed()` method, which is what will be called when the button is clicked, you would call the appropriate methods to set the foreground and background colors of the text area.

If you have ever used a Java GUI builder that lets you create an application by dropping components on a form and then wiring them together via events, the code that is being generated for you consists in large part of adapter classes that manage the logic of calling certain methods in the target objects when events are fired by the source objects.

What all this is leading up to is simply that the wiring of components in a GUI typically involves writing a lot of Java code in the form of classes that implement various event-listener interfaces. *J/Link* programmers want to write GUIs that use the standard Java event model, and they should not have to write Java code to do it. The solution is simple: *J/Link* provides a complete set of classes that implement the standard event-listener interfaces and whose actions are to call back into *Mathematica* to execute user-defined code. This brings all the event-handling logic down into *Mathematica*, where it can be scripted like every other part of the program.

Not only does this solution preserve the "pure *Mathematica*" property of even complex Java GUIs, it is vastly more flexible than writing a traditional application in Java. When you write in Java, or use a fancy drag-and-drop GUI builder, you hard-code the event logic. You have to decide at compile time what every click, scroll, and keystroke will do. But when you use *J/Link*, you decide how your program is wired together at run time. You can even change the behavior on the fly simply by typing a few lines of code.

J/Link provides implementations of all the standard AWT event-listener classes. These classes are named after the interfaces they implement, with "Math" prepended. Thus, the class that implements `ActionListener` is `MathActionListener`. (Perhaps these classes would be better named `MathXXXAdapter`.) The following table shows a summary of all the `MathListener` classes, the methods they implement, and the arguments they send to your *Mathematica* handler function.

class	*methods*	*arguments to Mathematica handler*
`MathActionListener`	`actionPerformed (ActionEvent e)`	`e, e.getActionCommand () (String)`
`MathAdjustmentListener`	`adjustmentValueChanged ( AdjustmentEvent e)`	`e, e.getAdjustmentType (), (Integer) e.getValue () (Integer)`
`MathComponentListener`	`componentHidden (ComponentEvent e) componentShown (ComponentEvent e) componentResized (ComponentEvent e) componentMoved (ComponentEvent e)`	`e`
`MathContainerListener`	`componentAdded (ContainerEvent e) componentRemoved (ContainerEvent e)`	`e`
`MathFocusListener`	`focusGained (FocusEvent e) focusLost (FocusEvent e)`	`e`
`MathItemListener`	`itemStateChanged (ItemEvent e)`	`e, e.getStateChange () (Integer)`
`MathKeyListener`	`keyPressed (KeyEvent e) keyReleased (KeyEvent e) keyTyped (KeyEvent e)`	`e, e.getKeyChar (),(Integer) e.getKeyCode () (Integer)`

MathMouseListener	mouseClicked (MouseEvent e) mouseEntered (MouseEvent e) mouseExited (MouseEvent e) mousePressed (MouseEvent e) mouseReleased (MouseEvent e)	e, e.getX (), (Integer) e.getY (), (Integer) e.getClickCount () (Integer)
MathMouseMotionListener	mouseMoved (MouseEvent e) mouseDragged (MouseEvent e)	e, e.getX (), (Integer) e.getY (), (Integer) e.getClickCount () (Integer)
MathPropertyChangeListener	propertyChanged (PropertyChangeEvent e)	e
MathTextListener	textValueChanged (TextEvent e)	e
MathVetoableChangeListener	vetoableChange (PropertyChangeEvent e)	e (veto the change by returning False from your handler)
MathWindowListener	windowOpened (WindowEvent e) windowClosed (WindowEvent e) windowClosing (WindowEvent e) windowActivated (WindowEvent e) windowDeactivated (WindowEvent e) windowIconified (WindowEvent e) windowDeiconified (WindowEvent e)	e

Listener classes provided with *J/Link*.

As an example of how to read this table, take the `MathKeyListener` class. `MathKeyListener` implements the `KeyListener` interface, which contains the methods `keyPressed()`, `keyReleased()`, and `keyTyped()`. If you register a `MathKeyListener` object with a component that fires `KeyEvent`s, then these three methods will be called in response to the key events they are named after. When any of these methods are called, they will call into *Mathematica* and exe-

cute a user-defined function, passing it three arguments: the KeyEvent object itself, followed by two integers that are the results of the event object's getKeyChar() and getKeyCode() methods. All the MathListener classes pass your handler function the event object itself, and a few, like this one, pass additional integer arguments that are commonly needed values. This just saves you the overhead of having to call back into Java to get these additional values.

To specify the *Mathematica* function associated with any of the methods of a MathListener object, call the object's setHandler() method. setHandler() takes two strings, the first of which is the name of the event-handler method (e.g., "actionPerformed" or "keyPressed"), and the second of which is the *Mathematica* function that should be called in response. The *Mathematica* function can be a name, as in "myButtonFunction" or a pure function (specified as a string). The reason for supplying the name of the actual Java method in the listener interface is that many of the listeners have multiple methods. setHandler() returns True if the handler was set correctly and False otherwise (for example, if the method you named is not spelled correctly).

obj@setHandler["*methodName*","*funcName*"]	set the *Mathematica* function that will be called when the MathListener object *obj*'s event-handler method methodName() is called.

Assigning the *Mathematica* function that will be called in response to an event notification.

The use of these classes will become clear in the simple examples that follow for modal and modeless windows, and in the more fully worked-out examples in the sections "A Simple Modal Input Dialog" and "A Piano Keyboard".

You are not required to use the *J/Link* MathListener classes for creating calls into *Mathematica* triggered by user actions. They are provided simply as a convenience. You could write your own classes to handle events and put calls into *Mathematica* directly into their code. All the "MathListener" classes in *J/Link* are derived from an abstract base class called, appropriately, MathListener. The code in MathListener takes care of all of the details of interacting with *Mathematica*, and it also provides the setHandler() methods that you use to associate events with *Mathematica* code. Users who want to write their own classes in MathListener style (for example, for one of the Swing-specific event-listener interfaces, which *J/Link* does not provide) are strongly encouraged to make their classes subclasses of MathListener to inherit all this

functionality. You should examine the source code for one of the concrete classes derived from `MathListener` (`MathActionListener` is probably the simplest one) to see how it is written. You can use this as a starting point for your own implementation. If you do not make your class a subclass of `MathListener`, and instead choose instead to write your own event-handler code that calls into *Mathematica*, you *must* read "Writing Your Own Event Handler Code".

Bringing Java Windows to the Foreground

If you are creating a Java window with a *Mathematica* program, you probably want that window to pop up in front of the notebook the user is working in, so that its presence becomes apparent. You might expect that the `toFront()` method of Java's `Window` class is what you would use for this, but this does not work on the Macintosh, and it works slightly differently on different Java runtimes on Windows. As a result of these differences, it is difficult to write a *Mathematica* program that behaves identically on all platforms and all Java virtual machines with respect to making Java windows visible in front of all other windows the user might see.

As a result of these unfortunate differences, *J/Link* provides a *Mathematica* function, `JavaShow`, which performs the proper steps on all configurations. You should use `JavaShow[`*window*`]` in place of `window@setVisible[True]`, `window@show[]`, or `window@toFront[]`. You will see `JavaShow` used in all the example programs. The argument to `JavaShow` must be a Java object that is an instance of a class that can represent a top-level window. Specifically, it must be of class `java.awt.Window` or a subclass. This includes the AWT `Frame` and `Dialog` windows, and also the Swing classes used for top-level windows (`JFrame`, `JWindow`, and `JDialog`).

`JavaShow[`*windowObj*`]`	make the specified Java window visible and bring it in front of all other windows, including notebook windows

Bringing a Java window to the foreground.

Modal Windows

Here is an example of a simple "modal" window. The window contains a button and a text field. The text field starts out displaying the value 1, and each time the button is clicked the value is incremented. The `com.wolfram.jlink.MathFrame` class is used for the enclosing window. `MathFrame` is a simple extension to `java.awt.Frame` that calls `dispose()` on itself when its close box is clicked (the standard `Frame` class does nothing).

```
frm = JavaNew["com.wolfram.jlink.MathFrame"];

button = JavaNew["java.awt.Button"];
textField = JavaNew["java.awt.TextField"];

frm@setLayout[JavaNew["java.awt.GridLayout"]];
frm@add[button];
frm@add[textField];
frm@pack[];
JavaShow[frm];
```

At this point, you should see a small frame window with a button on the left and a text field on the right. Now label the button and set the starting text for the field.

```
button@setLabel["++"];
textField@setText["1"];
```

Now you want to add behavior to the button that causes it to increment the text field value. Buttons fire `ActionEvents`, so you need an instance of `MathActionListener`.

```
buttonListener = JavaNew["com.wolfram.jlink.MathActionListener"];
```

It must be registered with the button by calling `addActionListener`.

```
button@addActionListener[buttonListener];
```

At this point, if you were to click the **++** button, the `actionPerformed()` method of your `MathActionListener` would be called (do not click the button yet!). You know from the `MathListener` table in the previous subsection that the `actionPerformed()` method will call a user-defined *Mathematica* function with two arguments: the `ActionEvent` object itself and the integer value that results from the event's `getActionCommand()` method.

You have not yet set the user-defined code to be called by the `actionPerformed()` method. That is done for all the `MathListener` classes with the `setHandler()` method. This method takes two strings, the first being the name of the method in the event-listener interface, and the second being the function you want called.

```
buttonListener@setHandler["actionPerformed", "buttonFunc"];
```

Now you need to define `buttonFunc`. It must be written to take two arguments, but in this example you are not interested in either argument.

```
buttonFunc[_, _] :=
 Module[{curText, newVal},
       curText = textField@getText[];
       newVal = ToExpression[curText] + 1;
       textField@setText[ToString[newVal]]
 ]
```

You are still not quite ready to try the button. If you click the button now, the Java user interface thread will hang because it will call into *Mathematica* trying to evaluate `buttonFunc` and wait for the result, but the result will never come because the kernel is not waiting for input to arrive on the Java link. What you need is a way to put the kernel into a state where it is continuously reading from the Java link. This is what makes the window "modal"—the kernel cannot do anything else until the window is closed. The function that implements this modal state is `DoModal`.

`DoModal[]`	put the kernel into a state where its attention is solely directed at the Java link
`EndModal[]`	what the Java program must call to make the `DoModal` function return, ending the modal state

Entering and exiting the modal state.

`DoModal` will not return until the Java program calls back into *Mathematica* to evaluate `EndModal[]`. While `DoModal` is executing, the kernel is ready to handle callbacks from Java—for example, from `MathListener` objects. The way to get the Java side to call `EndModal[]` is typically to use a `MathListener`. For example, if your window had **OK** and **Cancel** buttons, these should dismiss the window, so you would create `MathActionListener` objects and register them with these two buttons. These `MathActionListener` objects would be set to call `EndModal[]` in their `actionPerformed()` methods.

`DoModal` returns whatever the block of code that calls `EndModal[]` returns. You would typically use this return value to determine how the window was closed—for example, whether it was the **OK** or **Cancel** button. You could then take appropriate action. See "A Simple Modal Input Dialog" for an example of using the return value of `DoModal`.

In the present example, the only way to close the window is by clicking its close box. Clicking the close box fires a `windowClosing` event, so you use a `MathWindowListener` to receive notifications.

```
windowListener = JavaNew["com.wolfram.jlink.MathWindowListener"];
frm@addWindowListener[windowListener];
```

Now you assign the *Mathematica* function to be called when the close box is clicked. All you need it to do is call `EndModal[]`, so you can specify a pure function that ignores its arguments and does nothing but execute `EndModal[]`.

```
windowListener@setHandler["windowClosing", "EndModal[]&"];
```

The preceding few lines are a fine example of how to use a MathWindowListener to trigger a call to EndModal[] when a window's close box is clicked. You would use something similar to this, except with a MathActionListener, if you wanted to have an explicit **Close** button. In this example, though, there is an easier way. Mentioned earlier is that the MathFrame class is just a normal AWT Frame except that it calls dispose() on itself when its close box is clicked. Actually it has another useful property—it can also execute EndModal[] when its close box is clicked. Thus, if you use MathFrame as the top-level window class for your interfaces, you will not have to manually create a MathWindowListener to terminate the modal loop every time. To enable this behavior of MathFrame, you need to call its setModal method:

```
(***
 This is even easier than using the MathWindowListener above.
 We won't call it here, though, because we have already arranged
 for EndModal to be called, and bad things will happen if we try
 to call it twice.

frm@setModal[]

***)
```

You must not call setModal if you are not using DoModal. This is because after setModal has been called, the MathFrame will try to call into *Mathematica* when it is closed (to execute EndModal), and *Mathematica* needs to be in a state where it is ready for calls originating in Java. The same issue exists for any MathListener you create yourself.

Now that everything is ready, you can enter the modal state and use the window.

```
DoModal[]
```

When you are done playing with the window, click the close box in the frame, which will trigger a callback into *Mathematica* that calls EndModal[]. DoModal then returns, and the kernel is ready to be used from the front end. DoModal[] returns Null if you click the close box of a MathFrame.

Here is how the entire example looks when packaged into a single program. (The code for SimpleModal is also available as SimpleModal.nb in the JLink/Examples/Part1 directory.)

```
SimpleModal[] :=
    JavaBlock[
        Module[{frm, button, textField, windowListener,
                buttonListener, buttonFunc},

            (* Create the GUI components. *)
            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            button = JavaNew["java.awt.Button"];
            textField = JavaNew["java.awt.TextField"];

            (* Configure their properties. *)
            frm@setLayout[JavaNew["java.awt.GridLayout"]];
            frm@add[button];
            frm@add[textField];
            button@setLabel["++"];
            textField@setText["1"];
            frm@pack[];

            (* Create the listener and set its handler function. *)
            buttonListener =
                JavaNew["com.wolfram.jlink.MathActionListener"];
            buttonListener@setHandler["actionPerformed", ToString[buttonFunc]];
            button@addActionListener[buttonListener];

            (* Define buttonFunc. *)
            buttonFunc[_, _] :=
                JavaBlock[
                    Module[{curText, newVal},
                        curText = textField@getText[];
                        newVal = ToExpression[curText] + 1;
                        textField@setText[ToString[newVal]]
                    ]
                ];

            (* Make the window visible and bring it in front of any
               notebook windows. *)
            JavaShow[frm];

            (* Tell the frame to end the modal loop when it is closed. *)
            frm@setModal[];

            (* Enter the modal loop. *)
            DoModal[];
        ]
]
```

Remember that `DoModal` will not return until the Java side calls `EndModal`. You have to be a little careful when you call `DoModal` that you have already established a way for the Java side to trigger a call to `EndModal`. As explained earlier, you will typically have done this by using a `MathFrame` as the frame window and calling its `setModal` method, or by creating and registering a `MathListener` of your own that will call `EndModal` in response to a user action (such as clicking an **OK** or **Cancel** button). Once `DoModal` has begun, the kernel is not responsive to the front end and thus it is too late to set anything up. If you call `DoModal` and realize that for some reason you cannot end it from Java, you can abort it from the front end by selecting **Evaluation ▸ Interrupt Evaluation** in the menu, and then in the resulting dialog, clicking the button labeled **Abort**.

There is one subtlety you might notice in the code for `SimpleModal` that is not directly related to *J/Link*. In the line that calls `buttonListener@setHandler`, you pass the name of the button function not as the literal string `"buttonFunc"`, but as `ToString[buttonFunc]`. This is because `buttonFunc` is a local name in a `Module`, and thus its real name is not `buttonFunc`, but something like `buttonFunc$42`. To make sure you capture its true run-time name, you call `ToString` on the symbolic name. You could avoid this by simply not making the name `buttonFunc` local to the `Module`, but the way you have done it automatically cleans up the definition for `buttonFunc` when the `Module` finishes.

MathFrame and MathJFrame

You encountered the `MathFrame` class in this section, which is a useful top-level window class for *J/Link* programmers because it has three special properties. You have already encountered two of them: it calls `dispose()` on itself when it is closed, and it has the `setModal()` method, which gives it easy support for use with `DoModal`. The third property is that it has an `onClose()` method that you can use to specify *Mathematica* code that will be executed when the window is closed. The `onClose()` method is used in the Palette example in "Sharing the Front End: Palette-Type Buttons". *J/Link* also has a `MathJFrame` class, which is a subclass of the Swing `JFrame` class, and it also has these three special properties. Programmers who want to create interfaces with Swing components instead of AWT ones can use `MathJFrame` as their top-level window class.

Modeless Windows: Sharing the Kernel with Java

The previous subsection demonstrated how to write *J/Link* programs that display Java windows and then how to use the `DoModal` function to cause the kernel to wait until the window is closed. During the time that `DoModal` is running, the kernel is able to receive and process requests for computations that originate from the Java side. The word "modal" is used in this context to refer to the fact that the kernel is busy servicing the Java link, and thus the notebook front end cannot use the kernel until `DoModal` returns.

This arrangement works fine for many types of Java windows, and it is required for those that return a result to *Mathematica*, because the kernel cannot sensibly proceed until the window is dismissed. Unfortunately, it is too restrictive for a large class of user interface elements. Consider trying to duplicate the general concept of a front end palette window in Java. You want to have a window of buttons that, when clicked, cause some computation to occur in *Mathematica*.

Like a front end palette window, you want this window to be created and remain visible and active indefinitely. It would not be of much use if every time you wanted to click one of the buttons you had first to execute `DoModal[]` (and you would also have to arrange for each button to call `EndModal[]` as part of the computation it triggers). You want to be able to go back and forth between notebook windows in the front end and our Java window without needing manually to switch the kernel into and out of some special state each time.

What is needed is a way for the kernel to automatically pay attention to input arriving from the Java link in addition to the notebook front end link. What you really have here is two front ends vying for the kernel's attention. *J/Link* solves this problem by introducing a simple way in which the kernel can be put into a state where it is simultaneously listening for input on any number of links. The function that accomplishes this is `ShareKernel`.

Important Note: In *Mathematica* 5.1 and later, the kernel is always shared with Java. This means that the functions `ShareKernel` and `UnshareKernel` are not necessary and, in fact, do nothing at all. If you are writing program that only need to run in *Mathematica* 5.1 and later, you never need to call `ShareKernel` or `UnshareKernel` (`ShareFrontEnd` and `UnshareFrontEnd` are still useful, however). If your programs need to work on all versions of *Mathematica*, then you will need to use `ShareKernel` and `UnshareKernel` as described next.

`ShareKernel[]`	begin sharing the kernel with Java
`ShareKernel[`*link*`]`	begin sharing the kernel with *link*
`UnshareKernel[`*id*`]`	unregisters the request for sharing (that is, the call to `ShareKernel`) that returned *id*; kernel sharing will not be turned off unless no other requests are outstanding
`UnshareKernel[`*link*`]`	end sharing of the kernel with *link*
`UnshareKernel[]`	end sharing of the kernel with Java
`KernelSharedQ[]`	`True` if the kernel is currently being shared; `False` otherwise
`SharingLinks[]`	a list of the links currently sharing the kernel

Sharing the kernel.

ShareKernel takes a LinkObject as an argument and initiates sharing of the kernel between that link and the current $ParentLink (typically, the notebook front end). If you call ShareKernel with no arguments, it assumes you mean the link to Java. Most users will call it with no arguments.

```
In[]:= ShareKernel[];

In[]:= 2 + 2
Out[]= 4
```

Note that while the kernel is being shared, the input prompt has "(sharing)" prepended to it. The string that is prepended is specified by the SharingPrompt option to ShareKernel.

Sharing is transparent to the user. Other than the changed input prompt, there is nothing to suggest that anything different is going on. Input sent from either the front end or a Java program to the kernel will be evaluated and the result sent back to the program that sent the input. Each link is the kernel's $ParentLink during the time that the kernel is computing input that arrived from that link. In other words, ShareKernel takes care of shuffling the $ParentLink value back and forth between links as input arrives on each.

It is safe to call ShareKernel if the kernel is already being shared. This means that programs you write can call it without your having to worry that a user might already have initiated sharing. When you are finished with the need to share the kernel with Java, you can call UnshareKernel. This restores the kernel to its normal mode of operation, paying attention only to the front end.

```
In[]:= UnshareKernel[]
```

When called with no arguments, UnshareKernel shuts down sharing. This is not a desirable thing in most cases, because it might be that some other Java-based program is running that requires sharing. If you are writing code for others to use, you certainly cannot shut down sharing on your users just because your code is done with it. To solve this problem, ShareKernel returns a token (it is just an integer, but you should not be concerned with its representation) that reflects a request for sharing functionality. In other words, calling ShareKernel registers a request for sharing, turns it on if it is not on already, and returns a token that represents that particular request. When you call UnshareKernel, you pass it the token to "unregister" that particular request for sharing. Only if there are no other outstanding requests will sharing actually be turned off.

A quirk of ShareKernel is that you cannot call ShareKernel and UnshareKernel in the same cell. Doing so will cause the kernel to hang. Of course, there is no reason to ever do this, as kernel sharing is only relevant when it spans multiple evaluations (more precisely, the evaluation of multiple cells). There would be no point to turning sharing on and off within the scope of a single computation.

An example of a nontrivial user interface that uses ShareKernel is presented in "Real-Time Algebra: A Mini-Application".

Sharing the Front End

One goal of *J/Link* was to have Java user interface elements be as close as possible to first-class citizens of the notebook front end environment, in the way that notebooks and palettes are. The ability to share the kernel mimics one important aspect of this citizenship, hiding the fact that the Java runtime is a separate program and the kernel is normally only waiting for input from the front end.

There is one more important thing that palettes can do that would be nice to do from Java, and that is interact with the front end. You can create a palette button that, when clicked, evaluates the code Print["hello"]. You can do this easily with *J/Link* also, but with one big difference: when you click the palette button, hello appears in the active notebook, but when you click the Java button, the "hello" gets sent back to the Java program (which is, after all, the kernel's $ParentLink at that moment). Even if you persuaded the kernel to write the TextPacket that contains "hello" to the front end link instead of the Java link, nothing useful would happen because the front end is not paying attention to the kernel link when the front end is not waiting for the result of a computation. Poking some output at the front end while it is idle simply will not work.

J/Link provides the ShareFrontEnd function as the solution to this problem. ShareFrontEnd[] causes Print output and graphics generated by a Java user-interface element to appear in the front end. It also lets the Java side call *Mathematica* functions that manipulate elements of notebooks and have them work properly in the front end (for example, NotebookRead, NotebookWrite, SelectionEvaluate, and so on). While sharing is on, the front end behaves normally, and you can continue to use it for editing, calculations, or whatever. The sharing is transparent.

`ShareFrontEnd[]`	begin sharing the front end with Java
`UnshareFrontEnd[`*id*`]`	unregisters the request for sharing (that is, the call to `ShareFrontEnd`) that returned *id*; front end sharing will not be turned off unless no other requests are outstanding
`UnshareFrontEnd[]`	end sharing of the front end with Java
`FrontEndSharedQ[]`	`True` if the front end is currently being shared with Java; `False` otherwise

Sharing the notebook front end.

`ShareFrontEnd` currently does not work with a remote kernel; the same machine must be running the kernel and the front end.

`ShareFrontEnd` is as close as you currently can come to having Java user interfaces hosted directly by the notebook front end itself, as if they were special types of notebook windows. This type of tight integration might be possible in the future.

Note that `Print` output, graphics, and messages generated by a *modal* Java window will appear in the front end without needing to call `ShareFrontEnd`. This is because `$ParentLink` remains the front end link during `DoModal` (these "side effect" packets always get sent to `$ParentLink`), and also because the front end is able to handle various packets arriving from the kernel because the front end *is* in the middle of a computation—it is waiting for the result of the code that called `DoModal`. `ShareFrontEnd` is a way to restore a feature that was lost when you gained the ability to create *modeless* interfaces via `ShareKernel`. That is how to think of `ShareFrontEnd`—as a step beyond `ShareKernel` that allows side effect output generated by computations triggered in Java to appear in the notebook front end. `ShareFrontEnd` is particularly useful when developing code that needs to use `ShareKernel`, even if the code does not need the extra functionality of `ShareFrontEnd`. This is because *Mathematica* error messages generated by computations triggered by Java events get lost with `ShareKernel`. The messages will show up in the front end if front end sharing is turned on.

When you are done with the need to share the front end, call `UnshareFrontEnd`. Like the `ShareKernel`/`UnshareKernel` pair of functions, `ShareFrontEnd` returns a token that you should pass to `UnshareFrontEnd` to unregister the request for front end sharing. Only when all calls to `ShareFrontEnd` have been unregistered by calls to `UnshareFrontEnd` will front end sharing be turned off. You can force front end sharing to be shut down immediately by calling

UnshareFrontEnd with no arguments, but although this is convenient when you are developing code of your own, it should never be called in code that is intended for others to use. Just because your code is done with front end sharing does not mean that your users are done with it. Instead, save the token returned from ShareFrontEnd and pass it to UnshareFrontEnd.

ShareFrontEnd requires that the kernel be shared, so it calls ShareKernel internally. Calling UnshareKernel with no arguments forces kernel sharing to stop immediately, and this turns off front end sharing as well. Thus, you can use UnshareKernel[] as a quick shortcut to immediately shut down all sharing.

An example of some simple palette-type buttons that use ShareFrontEnd is presented in "Sharing the Front End: Palette-Type Buttons".

An important use for ShareFrontEnd is to allow a popup Java user interface to display graphics containing typeset expressions. When the kernel is asked to produce a graphic containing typeset expressions, say a plot with PlotLabel -> Sqrt[z], it crunches out PostScript for the plot itself, but when it comes time to produce PostScript for the typeset label, it cannot do this. Instead, it sends a special request back to the front end, asking it for the PostScript representation. Because dealing with typeset expressions is a skill possessed only by the notebook front end, when any other interface is driving the kernel, the interface must be careful to instruct the kernel to not attempt to typeset anything in a graphic (ShareKernel handles this automatically for you). This works fine, but you lose the ability to get pictures of typeset expressions in your Java interface.

ShareFrontEnd does two things to overcome this limitation: it fools the kernel into thinking that the Java runtime is a notebook front end and, therefore, capable of handling the special "convert to PostScript" requests; and it gives Java the ability to make good on this promise by forwarding the requests to the front end. "GraphicsDlg: Graphics and Typeset Output in a Window" describes an example of a Java dialog box that displays typeset expressions using ShareFrontEnd.

Summary of Modal and Modeless Operation

The previous discussion of modal and modeless operation, `ShareKernel`, and `ShareFrontEnd` may have seemed complex. In fact, the principles and uses of these techniques are simple. This will become clear upon seeing some more examples. Many of the example programs in "Example Programs" use `ShareKernel` or `ShareFrontEnd`. The important thing is to understand the capabilities they provide so that you can begin to see how to use them in your own programs.

If you want your user-interface element (typically a window) to tie up the kernel until the user dismisses it, then you will use the `setModal`/`DoModal`/`EndModal` suite. Because the internal workings of the modal state are simpler than the modeless state, you should use this style unless your program needs the features of a modeless window. You will always want to use this type of window if you need to return a result to a running *Mathematica* program, such as if you are creating a dialog box into which the user will enter values and then click **OK**. "A Simple Modal Input Dialog" gives an example of this type of dialog.

If you want your window to remain visible and active while the user returns to work in the front end, you must run your window in a "modeless" fashion. This requires calling `ShareKernel` to put the kernel into a state where it is simultaneously receptive to input arriving from either the notebook front end or Java. At this point the kernel is dividing its attention between two independent and essentially equivalent front ends. One drawback (or feature, depending on your point of view) of this state is that all side effect output like `Print` output, messages, or plots triggered by Java code is sent to Java instead of the front end (and the standard Java `MathListener` classes just throw all this output away). Thus, you could not create a button that prints something in a notebook window when it is clicked, like you can with a palette button in the front end. If you want to give your Java program the ability to interact with the front end the way that notebook and palette windows themselves can, you must instead use `ShareFrontEnd`, which you can think of as an extension to `ShareKernel`.

A very common mistake is to create a Java window, wire up a `MathListener` class that calls back to *Mathematica* on some event, and then trigger the event before you have called `DoModal` or `ShareKernel`. This will cause the Java user interface thread to hang. A symptom that the UI thread is hanging is that the controls in your Java window are visually unresponsive (for example, buttons will not appear to depress when you click them). If you do inadvertently get into this state, you can just call `ShareKernel` to allow the queued-up call(s) from Java to proceed.

"Manual" Interfaces: The ServiceJava Function

In addition to the modal and modeless types of interfaces just discussed, there is another type that in some ways is intermediate. Consider the following scenario. You want to create a *Mathematica* program that puts up a Java window and displays something in it that changes over the course of the program. So far, this sounds like an example of a "non-interactive" interface, which was discussed way back at the beginning of this section, the progress bar example being a classic case. Now, though, you want to add some interactivity to the window, meaning that you want user actions in the window to trigger calls into *Mathematica*. Keeping with the progress bar example, say you want to add an **Abort** button that stops the program. How do you manage to get the kernel's attention directed at the Java side so that Java events can trigger calls to *Mathematica*?

The modal type of interface will not work, because in the modal state the kernel is executing `DoModal`, not your computation—the kernel is doing nothing but paying attention to Java. The modeless type of interface will not work either, because the modeless technique causes the kernel to pay attention to the front end and Java alternately, letting each perform a full computation in turn. There is no sharing within the context of a single computation.

The obvious answer is the there needs to be a function that allows the kernel to service a single computation arriving from Java, if there is one waiting. That function is `ServiceJava`. Calling `ServiceJava` in a program will cause the kernel to accept one request for a computation from the Java side. It performs the computation and then returns control to your program. If there is no request waiting, `ServiceJava` returns immediately.

Here is some pseudocode showing the structure of a program that displays a progress bar with an **Abort** button and periodically calls `ServiceJava` to handle user clicks on that button, stopping the computation if requested.

```
    ... create progress bar ...
    progressBar@addActionListener[
        JavaNew["com.wolfram.jlink.MathActionListener", "(userCancelled =
True)&"]
    ];
    JavaShow[progressBar];
    While[i < 100 && !userCancelled,
        ... compute one iteration ...
        ... update progress bar ...
        ServiceJava[];
        i++
    ];
    ... destroy progress bar ...
```

You might recognize that `ServiceJava` is closely related to `DoModal`, and although this is not the actual implementation, you can think of `DoModal` as being written in terms of `ServiceJava` as follows:

```
(* Not the actual implementation of DoModal, but the principle is correct.
*)
DoModal[] :=
    While[!endModal,
        ServiceJava[]
    ]
```

Seen in this way, `DoModal` is a special case of the use of `ServiceJava`, where *Mathematica* is doing nothing but servicing requests from Java. Sometimes you need something else to be going on in *Mathematica*, but still need to be able to handle requests arriving from Java. That is when you call `ServiceJava` yourself. Like `DoModal`, there is no shifting of `$ParentLink` when `ServiceJava` is called. Thus, side-effect output like graphics, messages, and `Print` output triggered by Java computations appear in the notebook, just as if they were hard-coded into the *Mathematica* program that called `ServiceJava`.

The BouncingBalls example program presented in "BouncingBalls: Drawing in a Window" uses `ServiceJava`.

Using a GUI Builder

The preceding discussion on modal and modeless interfaces featured examples that were created entirely with *Mathematica* code. For complex user interfaces, you might find it more convenient to lay out your windows and wire up events with a drag-and-drop GUI builder like the

ones present in most commercial Java development environments. You are free to write as much or as little of the code for your interface in native Java. If you want events in your GUI to trigger calls into *Mathematica*, then you can use any of the `MathListener` classes from Java code just as they are used from *Mathematica* code. Alternatively, you could write your own Java code that calls into *Mathematica* at appropriate times. See the section "Writing Your Own Installable Java Classes" for information about how to write Java code that calls back into *Mathematica*. "GraphicsDlg: Graphics and Typeset Output in a Window" gives a simple example of a dialog box that was created with a GUI builder and is then invoked and controlled by *Mathematica* code.

Drawing and Displaying Mathematica Images in Java Windows

The MathCanvas and MathGraphicsJPanel classes

J/Link makes it easy to draw into Java windows from *Mathematica*, and also display *Mathematica* graphics and typeset expressions. The `MathCanvas` and `MathGraphicsJPanel` classes are provided for this purpose. You can use these classes in pure Java programs that use the *Mathematica* kernel, as described in "Writing Java Programs that use *Mathematica*", but it is also handy for Java windows that are created and scripted from *Mathematica*. Note that the `MathGraphicsJPanel` class is new in *J/Link* 2.0.

`MathCanvas` is a subclass of the AWT `Canvas` class, and `MathGraphicsJPanel` is a subclass of the Swing `JPanel` class. In terms of their special added *Mathematica* graphics capabilities, they are identical. These classes provide two ways to supply the image to be displayed. The first way is by providing a fragment of *Mathematica* code whose output will be displayed. The output can either be a graphics object, or a nongraphics expression that will be typeset. This makes it trivial to display *Mathematica* graphics or typeset expressions in a Java window. The second way to control the display is to provide a Java `Image` object that will be painted. This `Image` will typically be created by *Mathematica* code, such as code that creates a bitmap out of raw *Mathematica* data, or code that draws something using calls to Java's graphics routines.

Because `MathCanvas` and `MathGraphicsJPanel` are Java classes and can be used from Java programs as well as *Mathematica* programs, there is full JavaDoc format documentation for them in the JLink/Documentation/JavaDoc directory. You can browse that documentation for more details.

Showing *Mathematica* Graphics and Typeset Expressions

Here is a simple example of displaying a window that shows a *Mathematica* plot. This example uses `MathCanvas`, but the relevant parts would look the same if you used `MathGraphicsJPanel`. You will be using this window throughout this section, so do not close it if you are evaluating the code as you read this section.

```
frame = JavaNew["com.wolfram.jlink.MathFrame"];
frame@setLayout[JavaNew["java.awt.BorderLayout"]];
mathCanvas = JavaNew["com.wolfram.jlink.MathCanvas"];
frame@add["Center", mathCanvas];
frame@setSize[400, 400];
frame@layout[];
mathCanvas@setMathCommand["Plot[x, {x,0,1}]"];
JavaShow[frame];
```

As you can see, it is as simple as calling the canvas' `setMathCommand()` method. The argument to `setMathCommand()` is a string giving the code to be evaluated. This code must *return* a graphics expression, not just cause one to be produced. For example, `setMathCommand["Plot[x,{x,0,1}];"]` will not work because the trailing semicolon causes the expression to evaluate to `Null`. The image is automatically rendered at the correct size, and centered in the canvas if the actual image size produced by *Mathematica* does not completely fill the requested area (as is often the case with typeset output).

Calling `setMathCommand()` again resets the image.

```
mathCanvas@setMathCommand["Plot3D[Sin[x Cos[y]], {x,0,2Pi}, {y,0,2Pi}]"];
```

If the plotting command depends on variables in your *Mathematica* session, you can call `recompute()` to cause the graphic to be recomputed and rendered. For example, this displays a slow animation in the window.

```
n = 1.0;
mathCanvas@setMathCommand["Plot3D[Sin[n x Cos[y]], {x,0,2Pi}, {y,0,2Pi}]"];
Do[n += 0.1; mathCanvas@recompute[]; Pause[1], {10}]
```

Because you supply the expression as a string, remember to escape any quote marks inside the string with a backslash.

```
mathCanvas@setMathCommand["Plot[x, {x,0,1}, PlotLabel->\"This is a plot\"]"];
```

A `MathCanvas` can also display typeset expressions. The default behavior of `MathCanvas` is to expect that the expression supplied in `setMathCommand()` will evaluate to a graphics object, which should be rendered. To get it to instead typeset the return value, call the `setImageType()` method, supplying the constant `TYPESET`.

```
mathCanvas@setImageType[MathCanvas`TYPESET];
mathCanvas@setMathCommand["Integrate[Sqrt[x] Sqrt[1+x], x]"];
```

To switch back to displaying graphics, call mathCanvas@setImageType[MathCanvas`GRAPHICS]. The default format for typeset output is StandardForm. To switch to TraditionalForm, use the setUsesTraditionalForm() method. You call recompute() here because changing the output type does not force the image to be redrawn.

```
mathCanvas@setUsesTraditionalForm[True];
mathCanvas@recompute[];
```

Graphics are rendered using *Mathematica*'s Display command, which is fast and does not require the notebook front end to be running. For higher quality, though, particularly for 3D graphics, an alternative method is available that uses the front end for rendering services. You can switch to using this technique by calling the setUsesFE() method.

```
(* First, change back to graphics mode from typeset mode. *)
mathCanvas@setImageType[MathCanvas`GRAPHICS];

mathCanvas@setUsesFE[True];
mathCanvas@setMathCommand["Plot3D[Sin[x Cos[y]], {x,0,2Pi}, {y,0,2Pi}]"];
```

You might want to compare the resulting plot with setUsesFE[True] and setUsesFE[False].

An important point about using the front end for rendering is that when the computation to produce the image is performed, the front end must be in a state where it is receptive to requests for services from the kernel. There are two times when this is the case: either a cell in the front end is currently evaluating (as will be the case when you are calling setMathCommand() or recompute() from a *Mathematica* program), or ShareFrontEnd has been called. Looking at it from the other direction, the only time it will not work is if ShareKernel is in use, but not ShareFrontEnd, and the computation is triggered by an event in Java. The rule is that if you want to involve the front end for rendering, and you want to call setMathCommand() or recompute() from Java in response to a user action in a modeless interface, you need to use ShareFrontEnd; ShareKernel is not enough. Modal and modeless interfaces and ShareFrontEnd are discussed in the section "Creating Windows and Other User Interface Elements".

Drawing Using Java's Graphics Functions

You saw that the setMathCommand() method of the MathCanvas and MathGraphicsJPanel classes lets you supply a *Mathematica* expression whose output is to be displayed. You can also use a MathCanvas or MathGraphicsJPanel to display a Java Image by using the setImage() method instead of setMathCommand().

Now look at a simple example of drawing into a Java window from *Mathematica*. You will continue to use the same window and `MathCanvas` you have been working with. If this program used a `MathGraphicsJPanel` instead, the portions of the code related to drawing would look exactly the same. To draw into the `MathCanvas`, you create an offscreen image of the same dimensions, get a graphics context for drawing onto it, draw, and then use the `setImage()` method of `MathCanvas` to cause the offscreen image to be displayed. Drawing into an offscreen image and then blitting it to the screen is a standard technique for flicker-free drawing.

```
offscreen = mathCanvas@createImage[mathCanvas@getSize[]@width,
                                   mathCanvas@getSize[]@height];
g = offscreen@getGraphics[];
g@drawRect[100, 100, 200, 150];
mathCanvas@setImage[offscreen];
```

Programs that want to draw manually into a Java window from *Mathematica* will generally all have this same structure. It takes just a few more lines of code to turn our `MathCanvas` into a scribble program. Here is the complete program (this code is also provided as the file Scribble.nb in the JLink/Examples/Part1 directory).

```
Scribble[] :=
    JavaBlock[
        Module[{frame, mathCanvas, offscreen, g, mml, pts},
            frame = JavaNew["com.wolfram.jlink.MathFrame"];
            frame@setLayout[JavaNew["java.awt.BorderLayout"]];
            mathCanvas = JavaNew["com.wolfram.jlink.MathCanvas"];
            frame@add["Center", mathCanvas];
            frame@setSize[400, 400];
            frame@layout[];
            JavaShow[frame];
            (* Now create the offscreen image and the graphics context
               for drawing into it.
            *)
            offscreen = mathCanvas@createImage[mathCanvas@getSize[]@width,
                                               mathCanvas@getSize[]@height];
            g = offscreen@getGraphics[];
            (* Now create the MathMouseMotionListener that will do the drawing
               and set its mouseDragged event handler callback.
            *)
            mml = JavaNew["com.wolfram.jlink.MathMouseMotionListener"];
            mml@setHandler["mouseDragged", "mouseDraggedFunc"];
            mathCanvas@addMouseMotionListener[mml];
            mouseDraggedFunc[_, x_, y_, _] :=
                (g@drawLine[pts[[-1, 1]], pts[[-1, 2]], x, y];
                 mathCanvas@setImage[offscreen];
                 mathCanvas@repaintNow[];
                 AppendTo[pts, {x,y}];);
            (* Initialize the pts list and run the program modally. *)
            pts = {{0,0}};
            frame@setModal[];
            DoModal[];
            pts
        ]
    ]
```

Run the program, then click and drag the mouse to draw in the window. Close the window to end the program and the `Scribble` function will return the list of points drawn.

```
pts = Scribble[];
```

If you examine the list of points returned, you will see that they are based on Java's coordinate system, which has (0, 0) in the upper left. If you want to plot the points in a *Mathematica* graphic, you have to invert the *y* values. This is demonstrated in the Scribble.nb example notebook.

There is one new `MathCanvas` method demonstrated in this program, `repaintNow()`. In a computation-intensive program like this, where events are being fired on the user interface thread very quickly, and the handlers for these events take a nontrivial amount of time to execute, Java will sometimes delay repainting the window. The drawing becomes very chunky, with no visual effect for a while and then suddenly all the lines drawn in the last few seconds will appear. Even calling the standard `repaint()` method after every new point will not ensure that the window is updated in a timely manner. To solve this problem, the `repaintNow()` method is provided, which forces an immediate redraw of the canvas. If your program relies on smooth visual feedback from user events that fire rapidly, you should call `repaintNow()` also, even if it does not seem necessary on your system. There can be very significant differences between different platforms and different Java runtimes on the responsiveness of the screen updating mechanism.

The ability to draw in response to events in a `MathCanvas` or `MathGraphicsJPanel` opens up the possibility for some impressive interactive demonstrations, tutorials, and so on. Two of the larger example programs provided draw into a `MathCanvas` from *Mathematica:* BouncingBalls (in the section "BouncingBalls: Drawing in a Window") and Spirograph (in the section "Spirograph").

Bitmaps

You have seen how to draw into a `MathCanvas` or `MathGraphicsJPanel` by using an offscreen image. Another type of image that you can create with *Mathematica* code and display using `setImage()` is a bitmap. In this example you will create an indexed-color bitmap out of *Mathematica* data and display it. You will use an 8-bit color table, meaning that every data point in the image will be treated as an index into a 256-element list of colors. You could use a larger color table if desired.

You closed the frame window in the Scribble example, so you must first create a new frame and canvas for the bitmap.

```
frame = JavaNew["com.wolfram.jlink.MathFrame"];
frame@setLayout[JavaNew["java.awt.BorderLayout"]];
mathCanvas = JavaNew["com.wolfram.jlink.MathCanvas"];
frame@add["Center", mathCanvas];
frame@setSize[450, 450];
frame@layout[];
JavaShow[frame];
```

Here is the color table. It is an array of {r,g,b} triplets, with each color component being in the range 0..255. In this example, colors with low indices are mostly blue, and ones with high indices are mostly red.

```
colors = Table[{i, 0, 255 - i}, {i, 0, 255}];
```

The data is a 400×400 matrix of integers in the range 0..255 (because they are indices into the 256-element color table). In a real application, this data might be read from a file or computed in some more sophisticated way. If the range of numbers in the data did not span 0..255, you would have to scale it into that range, or a larger range if you wanted to use a deeper color table.

```
data = Table[Round[255 (0.5 + Sin[x]Cos[y]/2)],
            {x, Pi/100., 4Pi, Pi/100.}, {y, Pi/100., 4Pi, Pi/100.}];
```

Here you create the Java objects that represent the color model and bitmap. You can read the standard Java documentation on these classes for more information.

```
colorModel = JavaNew["java.awt.image.IndexColorModel", 8, 256,
                              Flatten[colors], 0, False];
bitmap = JavaNew["java.awt.image.MemoryImageSource", 400, 400,
                            colorModel, Flatten[data], 0, 400];
```

Now create an Image out of the bitmap and display it.

```
image = frame@getToolkit[]@createImage[bitmap];
mathCanvas@setImage[image];
```

The Java Console Window

J/Link provides a convenient means to display the Java "console" window. Any output written to the standard System.out and System.err streams will be directed to this window. If you are calling Java code that writes diagnostic information to System.out or System.err, then you can see this output while your program runs. Like most *J/Link* features, the console window can be used easily from either *Mathematica* or Java programs (its use from Java code is described in "Writing Java Programs that use *Mathematica*"). To use it from *Mathematica*, call the ShowJavaConsole function.

`ShowJavaConsole[]`	display the Java console window and begin capturing output written to `System.out` and `System.err`
`ShowJavaConsole["stream"]`	display the Java console window and begin capturing output written to the specified stream, which should be *"stdout"* for `System.out` or *"stderr"* for `System.err`
`ShowJavaConsole[None]`	stop all capturing of output

Showing the console window.

```
ShowJavaConsole[]
```

```
«JavaObject[com.wolfram.jlink.ui.ConsoleWindow] »
```

Capturing of output only begins when you call `ShowJavaConsole`, so when the window first appears it will not have any content that might have been previously written to `System.out` or `System.err`. You will also note that the *J/Link* console window displays version information about the *J/Link* Java component and the Java runtime itself. Calling `ShowJavaConsole` when the window is already open will cause it to come to the foreground.

To demonstrate, you can write some output from *Mathematica*. If you executed the `ShowJavaConsole[]` given earlier, then you will see "Hello from Java" printed in the window.

```
LoadJavaClass["java.lang.System"];
System`out@println["Hello from Java"]
```

Although it is convenient to demonstrate writing to the window using *Mathematica* code like this, this is typically done from Java code instead. Actually, there is one common circumstance where it is quite useful to use the Java console window for diagnostic output written from *Mathematica* code. This is the case where you have a "modeless" Java user interface (as described in the section "Creating Windows and Other User Interface Elements") and you have not used the `ShareFrontEnd` function. Recall that in this circumstance, output from calls to `Print` in *Mathematica* will not appear in the notebook front end. If you write to `System.out` instead, as in the example, then you will always be able to see the output. You might want to do this in other circumstances just to avoid cluttering up your notebook with debugging output.

Using JavaBeans

JavaBeans is Java's component architecture. Beans are reusable components that can be manipulated visually in a builder tool. At the code level, a Bean is essentially just a normal Java class that conforms to a particular design pattern with respect to how its methods are named and how it supports events and persistence.

JavaBeans has not been mentioned up to this point because there really is not anything special to be said. Beans are just Java classes, and they can be used and called like any other classes. It is probably the case that many Java classes you use from *Mathematica* will be Beans, whether they advertise themselves to be or not. This is especially true for user interface components.

Beans are typically designed to be used in a visual builder tool, where the programmer is not writing code and calling named methods directly. Instead, a Bean exposes "properties" to the builder tool, which can be examined and set using a property editor window. In a typical simple example, a Bean might have methods named `setColor` and `getColor`, and by virtue of this it would be said to have a property named "color". A property editor would have a line showing the name "color" and an edit field where you could type in a color. It might even have a fancy editor that puts up a color picker window to let you visually select a desired color.

For the purposes of a visual builder tool or other type of automated manipulation, beans try to hide the low-level details of actual method names. If you want to call methods in a Bean class from *Mathematica* code, you call them by name in the usual way, without any consideration of the "Bean-ness" of the class.

Note that it would be quite possible to add *Mathematica* functions to *J/Link* that would provide explicit support for Bean properties. For example, a function `BeanSetProperty` could be written that would take a Bean object, a property name as a string, and the value to set the property to. Then, instead of writing what is currently required:

```
bean@setColor[Color`green]
```

you could write:

```
BeanSetProperty[bean, "color", Color`green]
```

The `BeanSetProperty` function lets you write code that manipulates nebulous things called properties instead of calling specific methods in the Bean class. If you do not see any particular advantage in the `BeanSetProperty` style, then you know why there is no special Bean support along these lines in *J/Link*. The advantages of working with properties versus directly calling methods accrues only when you are using a builder tool and not actually writing code by hand.

If you are interested, here are simplistic implementations of `BeanSetProperty` and `BeanGetProperty`:

```
BeanSetProperty[bean_?JavaObjectQ, propName_String, val_] :=
    Module[{methName = "set" <> ToUpperCase[StringTake[propName, 1]] <>
            StringDrop[propName, 1]},
        Through[(bean @@ ToHeldExpression[methName])[val]]
    ]

BeanGetProperty[bean_?JavaObjectQ, propName_String] :=
    Module[{methName = "get" <> ToUpperCase[StringTake[propName, 1]] <>
            StringDrop[propName, 1]},
        Through[(bean @@ ToHeldExpression[methName])[]]
    ]
```

To make use of events that a JavaBean fires, you can use one of the standard `MathListener` classes, as described in the section "Creating Windows and Other User Interface Elements". JavaBeans often fire `PropertyChangeEvents`, and you can arrange for *Mathematica* code to be executed in response to these events by using a `MathPropertyChangeListener` or a `MathVetoableChangeListener`.

Hosting Applets

J/Link gives you the ability to run most applets in their own window directly from *Mathematica*. Although this may seem immensely useful, given the vast number of applets that have been created, most applets do not export any useful public methods. They are generally standalone pieces of functionality, and thus they benefit little from the scriptability that *J/Link* provides. Still, there are many applets that may be useful to launch from a *Mathematica* program.

Note that this section is not about writing applets that use the *Mathematica* kernel. That topic is covered in "Writing Applets".

`AppletViewer["`*applet class*`"]`	runs the named applet class in its own window. The default width and height are 300 pixels
`AppletViewer["`*applet class*`",`*params*`]`	runs the named applet class in its own window, supplying it the given parameters, which is a list of "name=value" specifications like those used in an HTML page

Running applets.

J/Link includes an `AppletViewer` function for running applets. This function takes care of all the steps of creating the applet instance, providing a frame window to hold it, and starting it running. The first argument to `AppletViewer` is the fully qualified name of the applet class. The second argument is an optional list of parameters in "name=value" format, corresponding to the parameters supplied to an applet in an HTML page that hosts it. For example, if the <applet> tag in a web page that hosts an applet looks like this:

```
<applet code="SomeApplet.class" width=400 height=300>
     <param name=foo value=bar>
</applet>
```

you would call AppletViewer like this:

```
AppletViewer["SomeApplet", {"width=400", "height=300", "foo=bar"}];
```

You will typically supply at least "WIDTH=" and "HEIGHT=" specifications to control the width and height of the applet window. If you do not specify these parameters, the default width and height are 300 pixels.

An excellent example of an applet that is useful to *Mathematica* users is LiveGraphics3D, written by Martin Kraus. LiveGraphics3D is an interactive viewer for *Mathematica* 3D graphics. It gives you the ability to rotate and zoom images, view them in stereo, and more. If you want to try the following example, you will need to get the LiveGraphics3D materials, available from http://wwwvis.informatik.uni-stuttgart.de/~kraus/LiveGraphics3D/. Make sure you put live.jar onto your CLASSPATH before trying that example, or use the AddToClassPath feature of *J/Link* to make it available.

First, load the PolyhedronOperations` package and create the graphic to display. The LiveGraphics3D documentation gives a more general-purpose function for turning a *Mathematica* graphics expression into appropriate input for the LiveGraphics3D applet but, for many examples, using ToString, InputForm, and N is sufficient.

```
<< PolyhedronOperations`
dodec = ToString[InputForm[
    N[Graphics3D[Stellate[Normal[PolyhedronD ata["Dodecahedron", "Faces"]]]]]]];
```

You specify the image to be displayed via the INPUT parameter, which takes a string giving the InputForm representation of the graphic.

```
AppletViewer["Live", {"INPUT=" <> dodec, "WIDTH=400", "HEIGHT=400"}];
```

The Live applet has a number of keyboard and mouse controls for manipulating the image. You can read about them in the LiveGraphics3D documentation. Try Alt+S to switch into a stereo view.

When you are done with an applet, just click the window's close box.

If the applet needs to refer to other files, you should be aware that AppletViewer sets the document base to be the directory specified by the "user.dir" Java system property. This will normally be *Mathematica*'s current directory (given by Directory[]) at the time that InstallJava was called.

Most applets expose no public methods useful for controlling from *Mathematica*, so there is nothing to do but start them up with AppletViewer and then let the user close the window when they are finished. The Live applet is an exception—it provides a full set of methods to allow the view point, spin, and so on to be modified by *Mathematica* code. These methods are in the Live class, so to call them you need an instance of the Live class. The way you used AppletViewer earlier does not give us any instance of the applet class. The construction and destruction of the applet instance was hidden within the internals of AppletViewer. You can also call AppletViewer with an instance of an applet class instead of just the class name. This lets you manage the lifetime of the applet instance.

```
applet = JavaNew["Live"];
AppletViewer[applet, {"INPUT=" <> dodec, "WIDTH=400", "HEIGHT=400"}];
```

Now you can call methods on the applet instance. See the LiveGraphics3D documentation for the full set of methods. This scriptability opens up lots of possibilities, such as programming "flyby" views of objects, or creating buttons that jump the image into certain orientations or spins.

```
applet@setMagnification[0.5];
```

When you are done, you call ReleaseJavaObject to release the applet instance. This can be done before or after the applet window is closed.

```
ReleaseJavaObject[applet]
```

Periodical Tasks

The section "Creating Windows and Other User Interface Elements" described the ShareKernel function and how it allows Java and the notebook front end to share the kernel's attention. A side benefit of this functionality is that it becomes easy to provide a means whereby users can schedule arbitrary *Mathematica* programs to run at periodical intervals during a session. Say you have a source that provides continuously updated financial data and you want to have some variables in *Mathematica* constantly reflect the current values. You have written a program that goes out and reads from the source to get the information, but you have to manually run this program all the time while you are working. A better solution would be to set up a periodical task that pulls the data from the source and sets the variables every 15 seconds.

`AddPeriodical[expr,secs]`	cause *expr* to be evaluated every *secs* seconds while the kernel is idle
`RemovePeriodical[id]`	stop scheduling of the periodical represented by *id*
`Periodical[id]`	return a list `{HoldForm[expr], secs}` showing the expression and time interval associated with the periodical represented by *id*
`Periodicals[]`	return a list of the *id* numbers of all currently scheduled periodicals
`SetPeriodicalInterval[id]`	reset the periodical interval for the periodical task represented by *id*
`$ThisPeriodical`	holds the *id* of the currently executing periodical task

Controlling periodical tasks.

You can set up such a task with the `AddPeriodical` function.

```
id = AddPeriodical[updateFinancialData[], 15];
```

`AddPeriodical` returns an integer ID number that you must use to identify the task—for example, when it comes time to stop scheduling it by calling `RemovePeriodical`. `AddPeriodical` relies on kernel sharing, so it calls `ShareKernel` if it has not already been called. There is no limit on the number of periodicals that can be established.

After scheduling that task, `updateFinancialData[]` will be executed every 15 seconds while the kernel is idle. Note that periodical tasks are run only when the kernel is not busy—they do not interrupt other evaluations. If the kernel is in the middle of another evaluation when the allotted 15 seconds elapses, the task will wait to be executed until immediately after the computation finishes. Any such delayed periodicals are guaranteed to be executed as soon as the kernel finishes with the current computation. They cannot be indefinitely delayed if the user is busy with numerous computations in the front end or in Java. The converse to these facts is also true—if a periodical is executing when the user evaluates a cell in the front end, the evaluation will not be able to start until all periodicals finish, but it is guaranteed to start immediately thereafter.

To remove a single periodical task, use `RemovePeriodical`, supplying the ID number of the periodical as the argument. To remove all periodical tasks, use `RemovePeriodical[Periodicals[]]`. Periodical tasks are all removed if you call `UnshareKernel[]` with no arguments, which turns off all kernel sharing. You would then need to use `AddPeriodical` again to reestablish periodical tasks.

You can reset the scheduling interval for a periodical task by calling `SetPeriodicalInterval`, which is new in *J/Link* 2.0. This line makes the financial data periodical execute every 10 seconds, instead of 15 as shown earlier.

```
SetPeriodicalInterval[id, 10]
```

Sometimes you might want to change the interval for a periodical task or remove it entirely from within the code of the task itself. `$ThisPeriodical` is a variable that holds the ID of the currently executing periodical task. It will only have a value during the execution of a periodical task. You use `$ThisPeriodical` from within your periodical task to obtain its ID so that you can call `RemovePeriodical` or `SetPeriodicalInterval`.

Periodical tasks do not necessarily have anything to do with Java, nor do they need to use Java. Technically, Java does not even need to be running. However, because Java is used by the internals of `ShareKernel` to yield the CPU, if Java is not running then setting a periodical task will cause the kernel to keep the CPU continuously busy. Periodical task functionality is included in *J/Link* because it is a simple extension to `ShareKernel`, and it does have some nice uses in association with Java.

A final note about periodical tasks is that they do not cause output to appear in the front end. Look at this attempt.

```
id = AddPeriodical[Print["hello"], 10];
```

The programmer expects to get `hello` printed in his notebook every 10 seconds, but nothing happens. During the time when periodicals are executed, `$ParentLink` is not assigned to the front end (or Java). Results or side effects like `Print` output, messages, or graphics vanish into the ether.

Before proceeding, clean up the periodical tasks you created.

```
RemovePeriodical[Periodicals[]];
```

Some Special Number Classes

Preamble

There is a set of special number-related classes in Java that *J/Link* maps to their *Mathematica* numeric representation. Like strings and arrays, objects of these number classes have an important property: although they are objects in Java, they have a meaningful "by value"

representation in *Mathematica*, so it is convenient for *J/Link* to automatically convert them to numbers as they are returned from Java to *Mathematica*, and back to objects as they are sent from *Mathematica* to Java.

These classes are the so-called "wrapper" classes that represent primitive types (`Byte`, `Integer`, `Long`, `Double`, and so on), `BigDecimal` and `BigInteger`, and any class used to represent complex numbers. The treatment of these classes is described in this section.

The "Wrapper" Classes: Integer, Float, Boolean, and Others

Java has a set of so-called "wrapper" classes that represent primitive types. These classes are `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `Boolean`. The wrapper classes hold single values of their respective primitive types, and are necessary to allow everything in Java to be represented as a subclass of Object. This lets various utility methods and data structures that deal with objects handle primitive types in a straightforward way. It is also necessary for Java's reflection capabilities.

If you have a Java method that returns one of these objects, it will arrive in *Mathematica* as an integer (for `Byte`, `Character`, `Short`, `Integer`, and `Long`), real number (for `Float` and `Double`), or the symbols `True` or `False` (for `Boolean`). Likewise, a Java method that takes one of these objects as an argument can be called from *Mathematica* with the appropriate raw *Mathematica* value. The same rules hold true for arrays of these objects, which are mapped to lists of values.

In the unlikely event that you want to defeat these automatic "pass by value" semantics, you can use the `ReturnAsJavaObject` and `JavaObjectToExpression` functions, discussed in "References and Values".

Complex Numbers

You have seen that Java number types (e.g., `byte`, `int`, `double`) are returned to *Mathematica* as integers and reals, and integers and reals are converted to the appropriate types when sent as arguments to Java. What about complex numbers? It would be nice to have a Java class representing complex numbers that mapped directly to *Mathematica*'s `Complex` type, so that automatic conversions would occur as they were passed back and forth between *Mathematica* and Java. Java does not have a standard class for complex numbers, so *J/Link* lets you name the class that you want to participate in this mapping.

SetComplexClass["*classname*"]	set the class to be mapped to complex numbers in *Mathematica*
GetComplexClass[]	return the class currently used for complex numbers

Setting the class for complex numbers.

You can use any class you like as long as it has the following properties:

1. A public constructor that takes two doubles (the real and imaginary parts, in that order)
2. Methods that return the real and imaginary parts, having the following signatures:

```
public double re();
public double im();
```

Say that you are doing some computations with complex numbers in Java, and you want to interact with these methods from *Mathematica*. You like to use the complex number class available from `netlib`. This class is named `ORG.netlib.math.complex.Complex` and is available at http://www.netlib.org/java/. You use the `SetComplexClass` function to specify the name of the class:

```
SetComplexClass["ORG.netlib.math.complex.Complex"];
```

Now any method or field that takes an argument of type `ORG.netlib.math.complex.Complex` will accept a *Mathematica* complex number, and any object of class `ORG.netlib.math.complex.Complex` returned from a method or field will automatically be converted into a complex number in *Mathematica*. The same holds true for arrays of complex numbers.

Note that you must call `SetComplexClass` before you load any classes that use complex numbers, not merely before you call any methods of the class.

BigInteger and BigDecimal

Java has standard classes for arbitrary-precision floating-point numbers and arbitrary-precision integers. These classes are `java.math.BigDecimal` and `java.math.BigInteger`, respectively. Because *Mathematica* effortlessly handles such "bignums," *J/Link* maps `BigInteger` to *Mathematica* integers and `BigDecimal` to *Mathematica* reals. What this means is that any Java method or field that takes, say, a `BigInteger` can be called from *Mathematica* by passing an integer. Likewise, any method or field that returns a `BigDecimal` will have the value returned to *Mathematica* as a real number.

Ragged Arrays

Java allows arrays that are deeper than one dimension to be "ragged," or non-rectangular, meaning that they do not have the same length at every position at the same level. For example, `{{1,2,3},{4,5},{6,7,8}}` is a ragged two-dimensional array. *J/Link* allows you to send and receive ragged arrays, but it is not the default behavior. The reason for this is simply efficiency—the *MathLink* library has functions that allow very efficient transfer of rectangular arrays of most primitive types (e.g., `byte`, `int`, `double`, and so on), whereas ragged ones have to be picked apart tediously with a series of individual calls to get every piece. This all happens deep inside *J/Link*, so you do not have to be concerned with the mechanics of array passing, but it has a huge impact on speed. To maximize speed, *J/Link* assumes that arrays of primitive types are rectangular. You can toggle back and forth between allowing and rejecting ragged arrays by calling the `AllowRaggedArrays` function with either `True` or `False`.

`AllowRaggedArrays[True]`	allow ragged (i.e., nonrectangular) arrays to be sent to Java

Ragged array support.

With `AllowRaggedArrays[True]`, sending of arrays deeper than one dimension is greatly slowed. Here is an example of array behavior and how it is affected. Assume the class `Testing` has the following method, which takes a two-dimensional array of ints and simply returns it:

```
public static int[][] intArrayIdentity(int[][] a) {
    return a;
}
```

Look what happens if you call it with a ragged array.

```
LoadClass["Testing"];
Testing`intArrayIdentity[{{1, 2, 3}, {4, 5}}]
```

```
Java::argxs1 :
  The static method Testing`intArrayIdentity was called with an incorrect
    number or type of arguments. The argument was {{1,2,3},{4,5}}.
$Failed
```

An error occurs because the *Mathematica* definition for the `Testing`intArrayIdentity()` function requires that its argument be a two-dimensional rectangular array of integers. The call never even gets out of *Mathematica*.

Here you turn on support for ragged arrays, and the call works. This requires modifications in both the *Mathematica*-side type checking on method arguments and the Java-side array-reading routines.

```
AllowRaggedArrays[True]
Testing`intArrayIdentity[{{1, 2, 3}, {4, 5}}]
{{1, 2, 3}, {4, 5}}
```

It is a good idea to turn off support for ragged arrays as soon as you no longer need it, since it slows arrays down so much.

```
AllowRaggedArrays[False]
```

Implementing a Java Interface with Mathematica Code

You have seen how *J/Link* lets you write programs that use existing Java classes. You have also seen how you can wire up the behavior of a Java user interface via callbacks to *Mathematica* via the `MathListener` classes. You can think of any of these `MathListener` classes, such as `MathActionListener`, as a class that "proxies" its behavior to arbitrary user-defined *Mathematica* code. It is as if you have a Java class that has its implementation written in *Mathematica*. This functionality is extremely useful because it greatly extends the set of programs you can write purely in *Mathematica*, without resorting to writing our own Java classes.

`ImplementJavaInterface["interfaceName",{"methName"->"mathFunc",...}]`
create an instance of a Java class that implements the named Java interface by calling back to *Mathematica* according to the given mappings of Java methods to *Mathematica* functions

Implementing a Java interface entirely in *Mathematica*.

It would be nice to be able to take this behavior and generalize it, so that you could take *any* Java interface and implement its methods via callbacks to *Mathematica* functions, and do it all without having to write any Java code. The `ImplementJavaInterface` function, new in *J/Link* 2.0, lets you do precisely that. This function is easier to understand with a concrete example. Say you are writing a *Mathematica* program that uses *J/Link* to display a Java window with a Swing menu, and you want to script the behavior of the menu in *Mathematica.* The Swing `JMenu` class fires events to registered `MenuListeners`, so what you need is a class that implements `MenuListener` by calling into *Mathematica*. A quick glance at the section on MathListen-

ers reveals that *J/Link* does not provide a `MathMenuListener` class for you. You could choose to write your own implementation of such a class, and in fact this would be very easy, even trivial, since you would make it a subclass of `MathListener` and inherit virtually all the functionality you would need. For the sake of this discussion, assume that you choose not to do that, perhaps because you do not know Java or you do not want to deal with all the extra steps required for that solution. Instead, you can use `ImplementJavaInterface` to create such a Java class with a single line of *Mathematica* code:

```
mathMenuListener =
    ImplementJavaInterface["javax.swing.event.MenuListener",
        {"menuSelected" -> "menuSelectedFunc",
         "menuCanceled" -> "menuCanceledFunc",
         "menuDeselected" -> "menuDeselectedFunc"}
    ];
myMenu@addMenuListener[mathMenuListener];

...

(* Later, define the three Mathematica event-handler functions: *)
menuSelectedFunc[menuEvent_] := ...

menuCanceledFunc[menuEvent_] := ...

menuDeselectedFunc[menuEvent_] := ...
```

The first argument to `ImplementJavaInterface` is the Java interface or list of interfaces you want to implement. The second argument is a list of rules that associate the name of a Java method from one of the interfaces with the name of a *Mathematica* function to call to implement that method. The *Mathematica* function will be called with the same arguments that the Java method takes. What `ImplementJavaInterface` returns is a Java object of a newly created class that implements the named interface(s). You use it just like any `JavaObject` obtained by calling `JavaNew` or through any other means. It is just as if you had written your own Java class that implemented the named interface by calling the associated *Mathematica* functions, and then called `JavaNew` to create an instance of that class.

It is not necessary to associate every method in the interface with a *Mathematica* function. Any Java methods you leave out of your list of mappings will be given a default Java implementation that returns `null`. If this is not an appropriate return value for the method (e.g., if the method returns an `int`) and the method gets called at some point an exception will be thrown. Generally, this exception will propagate to the top of the Java call stack and be ignored, but it is recommended that you implement all the methods in the Java interface.

The ImplementJavaInterface function makes use of the "dynamic proxy" capability introduced in Java 1.3. It will not work in Java versions earlier than 1.3. All Java runtimes bundled with *Mathematica* 4.2 and later are at Version 1.3 or later. If you have *Mathematica* 4.0 or 4.1, the ImplementJavaInterface function is another reason to make sure you have an up-to-date Java runtime for your system.

At first glance, the ImplementJavaInterface function might seem to give us the capability to write arbitrary Java classes in the *Mathematica* language, and to some extent that is true. One important thing you cannot do is extend, or subclass, an existing Java class. You also cannot add methods that do not exist in the interface you are implementing. Event-handler classes are a good example of the type of classes for which this facility is useful. You might think that the MathListener classes are rendered obsolete by ImplementJavaInterface, and it is true that their functionality can be duplicated with it. The MathListener classes are still useful for Java versions earlier than 1.3, but most importantly they are useful for writing pure Java programs that call *Mathematica*. Using a class implemented in *Mathematica* via ImplementJavaInterface in a Java program that calls *Mathematica* would be possible, but quite cumbersome. If you want a dual-purpose class that is as easy to use from *Mathematica* as from Java, you should write your own subclass of MathListener. One *poor* reason for choosing to use ImplementJavaInterface instead of writing a custom Java class is that you are worried about complicating your application by requiring it to include its own Java classes in addition to *Mathematica* code. As explained in "Deploying Applications That Use *J/Link*", it is extremely easy to include supporting Java classes in your application. Your users will not require any extra installation steps nor will they need to modify the Java class path.

Writing Your Own Installable Java Classes

Preamble

The previous sections have shown how to load and use existing Java classes. This gives *Mathematica* programmers immediate access to the entire universe of Java classes. Sometimes, though, existing Java classes are not enough, and you need to write your own.

J/Link essentially obliterates the boundary between Java and *Mathematica*, letting you pass expressions of any type back and forth and use Java objects in *Mathematica* in a meaningful way. This means that when writing your own Java classes to call from *Mathematica*, you usually do not need to do anything special. You write the code in exactly the same way as you would if

you wanted to use the class only from Java. (One important exception to this rule is that because it is comparatively slow to call into Java from *Mathematica*, you *might* need to design your classes in a way that will not require an excessive number of method calls from *Mathematica* to get the job done. This issue is discussed in detail in "Overhead of Calls to Java".)

In some cases, you might want to exert more direct control over the interaction with *Mathematica*. For example, you might want a method to return something different to *Mathematica* than what the method itself returns. Or you might want the method to not just return something, but also trigger a side effect in *Mathematica*—for example, printing something or displaying a message under certain conditions. You can even have an extended "dialog" with *Mathematica* before your method returns, perhaps invoking multiple computations in *Mathematica* and reading their results. You might also want to write a class of the `MathListener` type that calls into *Mathematica* as the result of some event triggered in Java.

If you do not want to do any of these things, then you can happily ignore this section. The whole point of *J/Link* is to make unnecessary the need to be concerned about the interaction with *Mathematica* through *MathLink*. Most programmers who want to write Java classes to be used from *Mathematica* will just write Java classes, period, without thinking about *Mathematica* or *J/Link*. Those programmers who want more control, or want to know more about the possibilities available with *J/Link*, read on.

The issues discussed in this section require some knowledge of *MathLink* programming (or, more precisely, *J/Link* programming using the Java methods that use *MathLink*), which is discussed in detail in "Writing Java Programs that use *Mathematica*". The fact that you meet some of these methods and issues here is a consequence of the false but useful dichotomy, noted in the Introduction, between using *MathLink* to write "installable" functions to be called from *Mathematica* and using *MathLink* to write front ends for *Mathematica*. *MathLink* is always used in the same way, it is just that virtually all of it is handled for you in the installable case. This section is about how to go beyond this default behavior, so you will be making direct *J/Link* calls to read and write to the link. Thus you will encounter concepts, classes, and methods in this section that are not explained until "Writing Java Programs That Use *Mathematica*".

Some of the discussion in this section will compare and contrast the process of writing an installable program in C. This is designed to help experienced *MathLink* programmers understand how *J/Link* works, and also to convince you that *J/Link* is a superior solution to using C, C++, or FORTRAN.

Installable Functions—The Old Way

Writing a so-called "installable" or "template" program in C requires a number of steps. If you have a file foo.c that contains a function `foo`, to call it from *Mathematica* you must first write a template (.tm) file that contains a template entry describing how you want `foo` to be called from *Mathematica*, what types of arguments it takes, and what it returns. You then pass this .tm file through a tool called `mprep`, which writes a file of C code that manages some, possibly all, of the *MathLink*-related aspects of the program. You also need to write a simple `main` routine, which is always the same. You then compile all of these files, resulting in an executable for just one platform.

Two big drawbacks of this method are that you need to write a template entry for every single function you want to call (imagine doing that for a whole function library), and the compiled program is not portable to other platforms. The biggest drawback, however, is that there is no automatic support for anything but the simplest types. If you want to do something as basic as returning a list of integers, you need to write the *MathLink* calls to do that yourself. And forget about object-oriented programming, as there is no way to pass "objects" to *Mathematica*.

Installable Functions in Java

J/Link makes all those steps go away. As you have seen all throughout this tutorial, you can literally call any method in any class, without any preparation.

It is only in cases where the default behavior of calling a method and receiving its result is not enough that you need to write specialty Java code. The rest of this section will examine some of the special techniques that can be used.

Setting Up Definitions in *Mathematica* When Your Class Is Loaded

Template entries in .tm files required by installable *MathLink* programs written in C have two features that might appear to be lost in *J/Link*. The first feature is the ability to specify arbitrary *Mathematica* code to be evaluated when the program is first "installed." This is done by using the `:Evaluate:` line in a template entry. The second feature is the ability to specify the way in which the function is to be called from *Mathematica*, including the name of the *Mathematica* function that maps to the C function, its argument sequence, how those arguments are mapped to the ones provided to the C function, and possibly some processing to be done on them before they are sent. This information is specified in the `:Pattern:` and `:Arguments:` lines of a template entry.

These two features are related to each other, because they both rely on the ability to specify *Mathematica* code that is loaded when an external program is installed. *J/Link* gives you this ability and more, through two special methods called `onLoadClass()` and `onUnloadClass()`. When a class is loaded into *Mathematica*, either directly through `LoadJavaClass` or indirectly by calling `JavaNew`, it is examined to see if it has a method with the following signature:

```
public static void onLoadClass(KernelLink ml);
```

If such a method is present, it will be called after all the method and field definitions for the class are set up in *Mathematica*. Because a class can only be loaded once in a Java session, this method will only be called once in the lifetime of a single Java runtime, although it may be called more than once in the lifetime of a single *Mathematica* kernel (because the user can repeatedly launch and quit the Java runtime). The `KernelLink` that is provided as an argument to this method is of course the link back to *Mathematica*.

A typical use for this feature would be to define the text for an error message issued by one of the methods in the class. Here is an example:

```
public static void onLoadClass(KernelLink ml) throwsMathLinkException {
    ml.evaluate("MyClass::sun = \"The foo() method can only be called on
Sunday.\"");
    ml.discardAnswer();
}
```

Note that this method throws `MathLinkException`. Your `onLoadClass()` method can throw any exceptions you like (a `MathLinkException` would be typical). This will not interfere with the matching of the expected signature for `onLoadClass()`. If an exception is thrown during `onLoadClass`, it will be handled gracefully, meaning that the normal operation of `LoadJavaClass` will not be affected. The only exception to this rule is if your code throws an exception while it is interacting with the link to the kernel, and more specifically, in the period between the time that it sends a computation to the kernel and the time that it begins to read the result. In other words, exceptions you throw will not break the `LoadJavaClass` mechanism, but it is up to you to make sure that you do not screw up the link's state by starting something you do not finish.

Another reason to use `onLoadClass()` would be if you wanted to create a *Mathematica* function for users to call that "wrapped" a static method call, providing it with a preferred name or argument sequence. If you have a class named `MyClass` with the method `public static`

`void myMethod(double[a])`, the definition that will be automatically created for it in *Mathematica* will require that its argument be a list of real numbers or integers. Say you want to add a definition named `MyMethod`, having the traditional *Mathematica* capitalization, and you also want this function automatically to use `N` on its argument so that it will work for anything that will *evaluate* to a list of numbers, such as `{Pi, 2Pi, 3Pi}`. Here is how you would set up such an additional definition:

```
public static void onLoadClass(KernelLink ml) throwsMathLinkException {
    ml.evaluate("MyMethod[x_] := myMethod[N[x]]");
    ml.discardAnswer();
}
```

In other words, if you are not happy with the interface to the class that will automatically be created in *Mathematica*, you can use `onLoadClass()` to set up the desired definitions without changing the Java interface.

The *Mathematica* context that will be current when `onLoadClass()` is called is the context in which all the class' static methods and fields are defined. That is why in the preceding example the definition was made for `MyMethod` and not `MyClass`MyMethod`. This is important since you cannot know the correct context in your Java code because it is determined by the user via the `AllowShortContext` option to `LoadJavaClass`.

It is generally not a good idea to use `onLoadClass()` to send a lot of code to *Mathematica*. This will make the behavior of your class hard for people to understand because the *Mathematica* code is hidden, and also inflexible since you would have to recompile it to make changes to the embedded *Mathematica* code. If you have a lot of code that needs to accompany a Java class, it is better to put that code into a *Mathematica* package file that you or your users load. That is, rather than having users load a class that dumps a lot of code into *Mathematica*, you should have your users load a *Mathematica* package that loads your class. This will provide the greatest flexibility for future changes and maintenance.

Finally, there is no reason why your `onLoadClass()` method needs to restrict itself to making *J/Link* calls. You could perform operations specific to the Java side, for example, writing some debugging information to the Java console window, opening a file for writing, or whatever else you desire.

Similar to the handling of the `onLoadClass()` method, the `onUnloadClass()` method is called when a class is unloaded. Every loaded class is unloaded automatically by `UninstallJava` right before it quits the Java runtime. You can use `onUnloadClass()` to remove definitions created by `onLoadClass()`, or perform any other clean-up you would like. The signature of `onUnloadClass()` must be the following, although it can throw any exceptions:

```
public static void onUnloadClass(KernelLink ml);
```

Note that the meaning of loading and unloading classes here refers to being loaded by *Mathematica* with `LoadJavaClass` either directly or indirectly. It does not refer to the loading and unloading of classes internally by the Java runtime. Class loading by the Java runtime occurs when the class is first used, which may have occurred long before `LoadJavaClass` was called from *Mathematica*.

Manually Returning a Result to *Mathematica*

The default behavior of a Java method called from *Mathematica* is to return to *Mathematica* exactly what the method itself returns. There are times, however, when you want to return something else. For example, you might want to return an integer in some circumstances, and a symbol in others. Or you might want a method to return one thing when it is being called from Java, and return something different to *Mathematica*. In these cases, you will need to manually send a result to *Mathematica* before the method returns.

Say you are writing a file-reading class that you want to call from *Mathematica*. Because you want almost the identical behavior to the standard class `java.io.FileInputStream`, your class will be a subclass of it. The only changes you want to make are to provide some more *Mathematica*-like behavior. One example is that you want the `read` method to return not -1 when it reaches the end of the file, but rather the symbol `EndOfFile`, which is what *Mathematica*'s built-in file-reading functions return.

```
import java.io.*;
import com.wolfram.jlink.*;

public class MyFileReader extends FileInputStream {

    <<constructors, other methods deleted>>

    public int read() {

        int i = super.read();
        if (i == -1) {
            KernelLink link = StdLink.getLink();
            if (link != null) {
                link.beginManual();
                try {
                    link.putSymbol("EndOfFile");
                } catch (MathLinkException e) {}
            }
        }
        return i;
    }
}
```

If the file has reached the end, `i` will be -1, and you want to manually return something to *Mathematica*. The first thing you need to do is get a `KernelLink` object that can be used to communicate with *Mathematica*. This is obtained by calling the static method `StdLink.getLink()`. If you have written installable *MathLink* programs in C, you will recognize the choice of names here. A C program has a global variable named stdlink that holds the link back to *Mathematica*. *J/Link* has a `StdLink` class that has a few methods related to this link object.

The first thing you do is check whether `getLink()` returns `null`. It will never be `null` if the method is being called from *Mathematica*, so you can use this test to determine whether the method is being called from *Mathematica* or as part of a normal Java program. In this way, you can have a method that can be used from Java in the usual way when a *Mathematica* kernel is nowhere in sight. The `getLink()` call works no matter if the method is called directly from *Mathematica*, or indirectly as part of a chain of methods triggered by a call from *Mathematica*.

Once you have verified that a link back to the kernel exists, the first thing to do is inform *J/Link* that you will be sending the result back to *Mathematica* yourself, so it should not try automatically to send the method's return value. This is accomplished by calling the `beginManual()` method on the `KernelLink` object.

You *must* call `beginManual()` before you send any part of a result back to *Mathematica*. If you fail to do this, the link will get out of sync and the next *J/Link* call you make from *Mathematica* will probably hang. It is safe to call `beginManual()` more than once, so you do not have to worry that your method might be called from another method that has already called `beginManual()`.

Returning to the example program, the next thing after `beginManual()` is to make the required "put"-type calls to send the result back to *Mathematica* (in this case, just a single `putSymbol()`). As always, these calls can throw a `MathLinkException`, so you need to wrap them in a `try/catch` block. The `catch` handler is empty, since there really is not anything to do in the unlikely event of a *MathLink* error. The internal *J/Link* code that wraps all method calls will handle the cleanup and recovery from any *MathLink* error that might have occurred calling `putSymbol()`. You do not need to do anything for `MathLinkExceptions` that occur while you are putting a result manually. The method call will return `$Failed` to *Mathematica* automatically.

Installable programs written in C can also manually send results back. This is indicated by using the `Manual` keyword in the function's template entry. Thus for C programs the manual/automatic decision must be made at compile time, whereas with *J/Link* it is a runtime switch. You can have it both ways with *J/Link*—a normal automatic return in some circumstances and a manual return in others, as the preceding example demonstrates.

Requesting Evaluations by *Mathematica*

So far, you have seen only cases where a Java method has a very simple interaction with *Mathematica*. It is called and returns a result, either automatically or manually. There are many circumstances, however, where you might want to have a more complex interaction with *Mathematica*. You might want a message to appear in *Mathematica*, or some `Print` output, or you might want to have *Mathematica* evaluate something and return the answer to you. This is a completely separate issue from what you want to return to *Mathematica* at the *end* of your method—you can request evaluations from the body of a method whether it returns its final result manually or not.

In some sense, when you perform this type of interaction with *Mathematica* you are turning the tables on *Mathematica*, reversing the "master" and "slave" roles for a moment. When *Mathematica* calls into Java, the Java code is acting as the slave, performing a computation and returning control to *Mathematica*. In the middle of a Java method, however, you can call back into *Mathematica*, temporarily turning it into a computational server for the Java side. Thus you would expect to encounter essentially all the same issues that are discussed in "Writing Java Programs That Use *Mathematica*", and you would need to understand the full *J/Link* Java-side API.

The full treatment of the `MathLink` and `KernelLink` interfaces is presented in "Writing Java Programs That Use *Mathematica*". This section discusses a few special methods in `KernelLink` that are specifically for use by "installed" methods. You have already seen one, the `beginManual()` method. Now you will treat the `message()`, `print()`, and `evaluate()` methods.

The task of issuing a *Mathematica* message from a Java method and triggering some `Print` output are so commonly done that the `KernelLink` interface has special methods for these operations. The method `message()` performs all the steps of issuing a *Mathematica* message. It comes in two signatures:

```
public void message(String symtag, String arg);
public void message(String symtag, String[] args);
```

The first form is for when you just have a single string argument to be slotted into the message text, and the second form is for if the message text needs two or more arguments. You can pass `null` as the second argument if the message text needs no arguments.

The `print()` method performs all the steps necessary to invoke *Mathematica*'s `Print` function:

```
public void print(String s);
```

Here is an example method that uses both. Assume that the following messages are defined in *Mathematica* (this could be from loading a package or during this class' `onLoadClass()` method):

```
Foo::arg = "The `1` argument to foo must be greater than or equal to 0."
```

Here is the Java code:

```
public static double foo(double x, double y) {

    KernelLink link = StdLink.getLink();
    if (link != null) {
        link.print("inside foo");
        if (x < 0)
            link.message("Foo::arg", "first");
        if (y < 0)
            link.message("Foo::arg", "second");
    }
    return Math.sqrt(x) * Math.sqrt(y);
}
```

Note that `print()` and `message()` send the required code to *Mathematica* and also read the result from the link (it will always be the symbol `Null`). They do not throw `MathLinkException` so you do not have to wrap them in `try/catch` blocks.

Here is what happens when you call `foo()`:

```
LoadJavaClass["MyClass", StaticsVisible → True];
foo[1.0, -2.0]
```

```
inside foo
```

```
Foo::arg : The second argument to foo must be greater than or equal to 0.
```

```
Indeterminate
```

Note that you automatically get `Indeterminate` returned to *Mathematica* when a floating-point result from Java is NaN ("Not-a-Number").

The methods `print()` and `message()` are convenience functions for two special cases of the more general notion of sending intermediate evaluations to *Mathematica* before your method returns a result. The general means of doing this is to wrap whatever you send to *Mathematica* in `EvaluatePacket`, which is a signal to the kernel that this is not the final result, but rather something that it should evaluate and send the result back to Java. You can explicitly send the `EvaluatePacket` head, or you can use one of the methods in `KernelLink` that use `EvaluatePacket` for you. These methods are:

```
void evaluate (String s) throws MathLinkException;
String evaluateToInputForm (String s, int pageWidth);
String evaluateToOutputForm (String s, int pageWidth);
byte[] evaluateToImage (String s, int width, int height);
byte[] evaluateToTypeset (String s, int pageWidth, boolean useStdForm);
```

These methods are discussed in "Writing Java Programs that use *Mathematica*" (actually, they also come in several more flavors with other argument sequences). Here is a simple example:

```
public static double foo(double x, double y) {

    KernelLink link = StdLink.getLink();
    if (link != null) {
        try {
            link.evaluate("2+2");
            // Wait for, and then read, the answer.
            link.waitForAnswer();
            int sum1 = link.getInteger();

            // evaluateToOutputForm makes the result come back as a
            // string formatted in OutputForm, and all in one step
            // (no waitForAnswer call needed).
            String s = link.evaluateToOutputForm("3+3");
            int sum2 = Integer.parseInt(s);

            // If you want, put the whole evaluation piece by piece,
            // including the EvaluatePacket head.
            link.putFunction("EvaluatePacket");
            link.putFunction("Plus", 2);
            link.put(4);
            link.put(4);
            link.waitForAnswer();
            int sum3 = link.getInteger();
        } catch (MathLinkException e) {
            // The only type of mathlink error we are likely to get
            // is from a "get" function when what we are trying to
            // get is not the type of expression that is waiting. We
            // just clear the error state, throw away the packet we
            // are reading, and let the method finish normally.
            link.clearError();
            link.newPacket();
        }
    }
    return Math.sqrt(x) * Math.sqrt(y);
}
```

Throwing Exceptions

Any exceptions that your method throws will be handled gracefully by *J/Link*, resulting in the printing of a message in *Mathematica* describing the exception. This was discussed in "How Exceptions Are Handled". If you are sending computations to *Mathematica* as described in the previous section, you need to make sure that an exception does not interrupt your code unexpectedly. In other words, if you start a transaction with *Mathematica*, make sure you complete it or you will leave the link out of sync and future calls to Java will probably hang.

Making a Method Interruptible

If you are writing a method that may take a while to complete, you should consider making it interruptible from *Mathematica*. In C *MathLink* programs, a global variable named `MLAbort` is provided for this purpose. In *J/Link* programs, you call the `wasInterrupted()` method in the `KernelLink` interface:

```
public boolean wasInterrupted();
```

Here is an example method that performs a long computation, checking every 100 iterations whether the user tried to abort it (using the **Interrupt Evaluation** or **Abort Evaluation** commands in the **Evaluation** menu).

```
public int foo() {

    KernelLink link = StdLink.getLink();
    for (int i = 0; i < 10000, i++) {
        ... perform one step ...
        if (i % 100 == 0 && link.wasInterrupted())
            return 0;  // Return value will not be seen by Mathematica.
    }
    return 42;
}
```

This method returns 0 if it detects an attempt by the user to abort, but this value will never be seen by *Mathematica*. This is because *J/Link* causes a method or constructor call that is aborted to return `Abort[]`, whether or not you detect the abort in your code. Therefore, if you detect an abort and want to honor the user's request, just return some value right away. When *J/Link* returns `Abort[]`, the user's entire computation is aborted, just as if the `Abort[]` was embedded in *Mathematica* code. This means that you do not have to be concerned with any details of propagating the abort back to *Mathematica*—all you have to do is return prematurely if you detect an abort request, and the rest is handled for you.

J/Link makes no distinction between an interrupt request and an abort request; they each cause `wasInterrupted()` to return `true`. Recall that *Mathematica* has separate commands for interrupting and aborting computations. The "Abort" operation (Alt+. on Windows) causes the entire computation to end as soon as possible and return `$Aborted`. The "Interrupt" operation (Alt+, on Windows) brings up a dialog box with further choices. If this **Interrupt** dialog box is triggered when a Java method is executing, it has a different set of buttons than when normal *Mathematica* code is executing. One of the options is **Send Abort to Linked Program** and another is **Send Interrupt to Linked Program**. Both of these choices have the same effect for Java methods, which is to cause `wasInterrupted()` to return `true` and the call to return `Abort[]` when it completes. The third button is **Kill Linked Program**, which will cause the Java runtime to quit. If you call a Java method that is not interruptible, killing the Java runtime in this way is the only way to make the method call terminate (you can also kill the Java runtime using process control features of your operating system).

Sometimes you might want a Java method to detect an abort and do something other than cause the entire *Mathematica* computation to abort. For example, you might want a loop to stop and return its results up to that point. Note that this is not generally recommended. Users expect a program to abort and return `$Aborted` when they issue an abort request. In some cases, however, especially if the code is not intended for use by a large community, you might find it useful to use an abort as a "message" to communicate some information to your Java code instead of just having the computation aborted. This idea is similar to *Mathematica*'s `CheckAbort` function, which allows you to detect an abort and absorb it so that it does not propagate further and abort the entire computation. To "absorb" the abort in your Java code so that *J/Link* does not return `Abort[]`, simply call the `clearInterrupt()` method:

```
public void clearInterrupt();
```

Here is an example:

```
public int foo() {

    KernelLink link = StdLink.getLink();
    for (int i = 0; i < 10000, i++) {
        ... perform one step ...
        if (i % 100 == 0 && link.wasInterrupted()) {
            link.clearInterrupt();
            return resultSoFar; // This is the value that will be returned
to Mathematica
        }
    }
    ...
    return 42;
}
```

Writing Your Own Event Handler Code

"Handling Events with *Mathematica* Code: The "MathListener" Classes" introduced the topic of triggering calls into *Mathematica* as a response to events fired in Java, such as clicking a button. A set of classes derived from `MathListener` is provided by *J/Link* for this purpose. You are not required to use the provided `MathListener` classes, of course. You can write your own classes to handle events and put calls into *Mathematica* directly into their code. All the event handler classes in *J/Link* are derived from the abstract base class `MathListener`, which takes care of all the details of interacting with *Mathematica*, and also provides the `setHandler()` methods that you use to associate events with *Mathematica* code. Users who want to write their own `MathListener`-style classes (for example, for one of the Swing-specific event listener interfaces, which *J/Link* does not provide) are strongly encouraged to make their classes subclasses of `MathListener` to inherit all this functionality. You should examine the source code for `MathListener`, and also one of the concrete classes derived from it (`MathActionListener` is probably the simplest one) to see how it is written. You can use this as a starting point for your own implementation.

There is a new feature of *J/Link* 2.0 that should be pointed out in this context. This is the `ImplementJavaInterface` *Mathematica* function, which lets you implement any Java interface entirely in *Mathematica* code. `ImplementJavaInterface` is described in more detail in "Implementing a Java Interface with *Mathematica* Code", but a common use for it would be to

create event-handler classes that implement a "Listener"-type interface for which *J/Link* does not have a built-in `MathListener`. This is discussed in more detail in "Implementing a Java Interface with *Mathematica* Code", and if you choose this technique, then you do not have to worry about any of the issues in this section because they are handled for you.

If you are going to write a Java class, and you choose not to derive your class from `MathListener`, there are two very important rules that *must* be adhered to when writing event-handler code that calls into *Mathematica*. To be more precise, these rules apply whenever you are writing code that needs to call into *Mathematica* at a point when *Mathematica* is not currently calling into Java. That may sound confusing, but it is really very simple. "Requesting Evaluations by *Mathematica*" showed how to request evaluations by *Mathematica* from within a Java method. In this case, *Mathematica* has called your Java method, and while *Mathematica* is waiting for the result, your code calls back to perform some computation. This works fine as described in that earlier section, because at the point the code calls back into *Mathematica*, *Mathematica* is in the middle of a call to Java. This is a true "callback"—*Mathematica* has called Java, and during the handling of this call, Java calls back to *Mathematica*. In contrast, consider the case where some Java code executes in response to a button click. When the button click event fires, *Mathematica* is probably not in the middle of a call to Java.

Special considerations are necessary in the latter case because there are two threads in the Java runtime that are using *MathLink*. The first one is created and used by the internals of *J/Link* to handle standard calls into Java originating in *Mathematica* as described throughout this tutorial. The second one is the Java user interface thread (sometimes called the AWT thread), which is the one on which your event handler code will be called. You need to make sure that your use of the link back to the kernel on the user interface thread does not interfere with *J/Link*'s internal thread.

The following code shows an idealized version of the `actionPerformed()` method in the `MathActionListener` class. The actual code in `MathActionListener` is different, because this work is farmed out to the parent class, `MathListener`, but this example shows the correct flow of operations. This is the code that is executed when the associated object's action occurs (like a button click).

```
public void actionPerformed(ActionEvent e) {
    KernelLink ml = StdLink.getLink();
    StdLink.requestTransaction();
    synchronized (ml) {
        try {
            // Send the code to perform the user's requested operation.
            ml.putFunction("EvaluatePacket", 1);
            ... code to put rest of expression to evaluate goes here ...
            ml.endPacket();
            ml.discardAnswer();
        } catch (MathLinkException exc) {
            ...
        }
    }
}
```

The first rule to note in this code is that the complete transaction with *Mathematica*, which includes sending the code to evaluate and completely reading the result, is wrapped in a `synchronized(ml)` block. This is how you ensure that the user interface thread has exclusive access to the link for the entire transaction. The second rule is that the `synchronized(ml)` statement must be preceded by a call to `StdLink.requestTransaction()`. This call will block until the kernel is at a point where it is ready to accommodate evaluations originating in Java. The call must occur before the `synchronized(ml)` block begins, and once you call it you must make sure that you send something to *Mathematica*. In other words, when `requestTransaction()` returns, the kernel will be blocking in an attempt to read from the Java link. The kernel will be stuck in this state until you send it something, so you must protect against a Java exception being thrown after you call `requestTransaction()` but before you send anything. Typically you will do this simply by calling `requestTransaction()` immediately before the `synchronized(ml)` block begins and you start sending something.

It was just said that `StdLink.requestTransaction()` will block until the kernel is ready to accept evaluations originating in Java. To be specific, it will block until one of the following conditions occurs:

- *Mathematica* executes `DoModal`
- *Mathematica* executes `ServiceJava`

- Kernel sharing has been turned on via `ShareKernel` or `ShareFrontEnd`, and the kernel is not busy with another computation
- *Mathematica* is already in the middle of a call to Java
- Java is not being used from *Mathematica* (`InstallJava` has not been called)

These conditions should make sense given the discussion about creating user interface elements in the section "Creating Windows and Other User Interface Elements". `DoModal`, `ShareKernel`, and `ServiceJava` are the three ways in which you direct the kernel's attention to the Java link so that it can detect incoming request for computations.

If you make the common mistake of inadvertently triggering a call to *Mathematica* from Java before you have called `DoModal` or `ShareKernel`, the Java user interface thread will hang. This can be easily remedied by calling `DoModal`, `ShareKernel`, or `ServiceJava` afterwards (`ServiceJava` may need to be called more than once, if more than one event callback is queued up).

If the rule about when it is necessary to use `StdLink.requestTransaction()` and `synchronized(ml)` is confusing, you will be happy to learn that it is fine to use these constructs in any code that calls *Mathematica*. In code that does not need them, they are pointless, but harmless, and will not cause the calling thread to block. If you are writing a Java method that needs to call *Mathematica* and there is any chance that it might be called from the user interface thread, add the `StdLink.requestTransaction()` and `synchronized(ml)`.

Debugging Your Java Classes

You can use your favorite debugger to debug Java code that is called from *Mathematica*. The only issue is that you typically have to launch a Java program inside the debugger to do this. The Java program that you need to launch is the one that is normally launched for you when you call `InstallJava`. The class that contains *J/Link*'s `main()` method is `com.wolfram.jlink.Install`. Thus, the command line to start *J/Link* that is executed internally by `InstallJava` is typically

```
java –classpath /path/to/JLink.jar com.wolfram.jlink.Install
```

There may be additions or modifications to this depending on the options to `InstallJava`, and also some extra *MathLink*-specific arguments are tacked on at the end. To use a debugger, you just have to launch Java with the appropriate command-line arguments that allow you to establish the link to *Mathematica* manually.

If you use a development environment that has an integrated debugger, then the debugger probably has a setting for the main class to use (the class whose `main()` method will be invoked) and a setting for command-line arguments. For example, in WebGain Visual Café, you can set these values in the **Project** panel of the **Project/Options** dialog. Set the main class to be `com.wolfram.jlink.Install`, and the arguments to be something like this:

```
(On Windows:)
-linkmode listen -linkname foo

(On Unix/Linux:)
-linkmode listen -linkprotocol tcp -linkname 1234
```

Then start the debugging session. You should see the *J/Link* copyright notice printed and then Java will wait for *Mathematica* to connect. To do this, go to your *Mathematica* session, make sure the JLink.m package has been read in, and execute:

```
(* On Windows: *)
ReinstallJava[LinkConnect["foo"]]

(* On Unix: *)
ReinstallJava[LinkConnect["1234", LinkProtocol -> "TCP"]]
```

This works because `ReinstallJava` can take a `LinkObject` as its argument, in which case it will not try to launch Java itself. This allows you to manually establish the *MathLink* connection between Java and *Mathematica*, then feed that link to `ReinstallJava` and let it do the rest of the work of preparing the *Mathematica* and Java sides for interacting with each other.

If you like to use a command-line debugger like jdb, you can do the following:

```
C:\>jdb
Initializing jdb...
> run com.wolfram.jlink.Install -linkmode listen -linkname foo
running ...
main[1] J/Link (tm)
Copyright (C) 1999-2000, Wolfram Research, Inc. All Rights Reserved.
www.wolfram.com
Version 1.1

Current thread "main" died. Execution continuing...
>
```

The message about the main thread dying is normal. Now jdb is ready for commands. First, though, you have to execute in your *Mathematica* session the `LinkConnect` and `ReinstallJava` lines shown earlier. This example was for Windows, so Unix users will have to adjust the `run` line to reflect the proper arguments:

```
> run com.wolfram.jlink.Install -linkmode listen -linkprotocol tcp
-linkname 1234
```

Deploying Applications that use J/Link

This section discusses some issues relevant to developers who are creating add-ons for *Mathematica* that use *J/Link*.

J/Link uses its own custom class loader that allows it to find classes in a set of locations beyond the startup class path. As described in "Dynamically Modifying the Class Path", users can grow this set of extra locations to search for classes by calling the `AddToClassPath` function. One of the motivations for having a custom class loader was to make it easy for application developers to distribute applications that have parts of their implementation in Java. If you structure your application directory properly, your users will be able to install it simply by copying it into any standard location for *Mathematica* applications. *J/Link* will be able to find your Java classes immediately, without users having to perform any classpath-related operations or even restart Java.

If your *Mathematica* application uses *J/Link* and includes its own Java components, you should create a Java subdirectory in your application directory. You can place any jar files that your application needs into this Java subdirectory. If you have loose class files (not bundled into a jar file), they should go into an appropriately nested subdirectory of the Java directory. "Appropriately nested" means that if your class is in the Java package `com.somecompany.math`, then its class file goes into the com/somecompany/math subdirectory of the Java directory. If the class is not in any package, it can go directly into the Java directory. *J/Link* can also find native libraries and resources your application needs. Native libraries must be in a subdirectory of your Java/Libraries directory that is named after the `$SystemID` of the platform on which it is installed. Here is an example directory structure for an application that uses *J/Link*:

```
MyApp/
    ... other files and directories used by the application ...
    Java/
        MyAppClasses.jar
        MyImage.gif
        Libraries/
            Windows/
                MyNativeLibrary.dll
            PowerMac/
                MyNativeLibrary
            Darwin/
                libMyNativeLibrary.jnilib
            Linux/
                libMyNativeLibrary.so
            ... and so on for other Unix platforms
```

Your application directory must be placed into one of the standard locations for *Mathematica* applications. These locations are listed as follows. In this notation, `$InstallationDirectory`/AddOns/Applications means "The AddOns/Applications subdirectory of the directory whose value is given by the *Mathematica* variable `$InstallationDirectory`."

`$UserAddOnsDirectory`/Applications (Mathematica 4.2 and later only)

`$AddOnsDirectory`/Applications (Mathematica 4.2 and later only)

`$InstallationDirectory`/AddOns/Applications

`$InstallationDirectory`/AddOns/ExtraPackages

Coding Tips

Here are a few tips on producing high-quality applications. These suggestions are guided by mistakes that developers frequently make.

Call `InstallJava` in the body of a function or functions, not when your package is read in. It is best to avoid side effects during the reading of a package. Users expect reading in a package to be fast and to do nothing but load definitions. If you launch Java at this time, and it fails, it could cause a mysterious hang in the loading process. It is better to call `InstallJava` in the code of one or more of your functions. You probably do not need to call `InstallJava` in every single function that uses Java. Most applications have a few "major" functions that users

are likely to use almost exclusively, or at least at the start of their session. If your application does not have this property, then provide an initialization function that your users must call first, and call `InstallJava` inside it.

Call `InstallJava` with no arguments. You cannot know what options your users need for Java on their systems, so do not override what they may have set up. It is the user's responsibility to make sure that they call `SetOptions` to customize the options for `InstallJava` as necessary. Typically this would be done in their `init.m` file.

Make sure you use `JavaBlock` and/or `ReleaseJavaObject` to avoid leaking object references. You cannot know how others will use your code, so you need to be careful to avoid cluttering up their sessions with a potentially large number of useless objects. Sometimes you need to create an object that persists beyond the lifetime of a single *Mathematica* function, like a viewer window. In such cases, use a `MathFrame` or `MathJFrame` as your top-level window and use its `onClose()` method to specify *Mathematica* code that releases all outstanding objects and unregisters kernel or front end sharing you may have used. If this is not possible, provide a cleanup function that users can call manually. Use `LoadedJavaObjects` to look at the list of objects referenced in *Mathematica* before and after your functions run; it should not grow in length.

If you use `ShareKernel` or `ShareFrontEnd`, make sure you save the return values from these functions and pass them as arguments to `UnshareKernel` and `UnshareFrontEnd`. Do not call `UnshareFrontEnd` or `UnshareKernel` with no arguments, as this will shut down sharing even if other applications are using it.

Do not assume that the Java runtime will not be restarted during the lifetime of your application. Although users are strongly discouraged to call `UninstallJava` or `ReinstallJava`, it happens. It is unavoidable that some applications will fail if the Java runtime is shut down at an inopportune time (e.g., when they have a Java window displayed), but there are steps you can take to increase the robustness of your application in the face of Java shutdowns and restarts. One step was already given as the first tip listed—call `InstallJava` at the start of your "major" functions. Another step is to avoid caching `JavaClass` or `JavaObject` expressions unnecessarily, as these will become invalid if Java restarts. An example of this is calling `InstallJava` and then `LoadJavaClass` and `JavaNew` several times when your package file is read in, and storing the results in private variables for the lifetime of your package. This

is problematic if Java is restarted. Never store JavaClass expressions—call LoadJavaClass whenever there is any doubt about whether a class has been loaded into the current Java runtime. Calling LoadJavaClass is very inexpensive if the class has already been loaded. If you have a JavaObject that is very expensive to create and therefore you feel it necessary to cache it over a long period of time in a user's session, consider using the following idiom to test whether it is still valid whenever it is used. The JavaObjectQ test will fail if Java has been shut down or restarted since the object was last created, so you can then restart Java and create and store a new instance of the object.

```
SomeFunction[] :=
    Module[{...},
        If[!JavaObjectQ[$myCachedExpensiveJavaObject],
            InstallJava[];
            $myCachedExpensiveJavaObject = JavaNew[...];
        ];
        ... use $myCachedExpensiveJavaObject ...
    ]
```

Do not call UninstallJava or ReinstallJava in your application. You need to coexist politely with other applications that may be using Java. Do not assume that when your package is done with Java, the user is done with it as well. Only users should ever call UninstallJava, and they should probably never call it either. There is no cost to leaving Java running. Likewise, users will rarely call ReinstallJava unless they are doing active Java development and need to reload modified versions of their classes.

Example Programs

Introduction

This section will work through some example programs. These examples are intended to demonstrate a wide variety of techniques and subtleties. Discussions include some nuances in the implementations and touch on most of the major issues in *J/Link* programming.

This will take a relatively rigorous approach, and in particular it will be careful to avoid leaking references. As discussed in the section "JavaBlock", JavaBlock and ReleaseJavaObject are the tools in this fight, but if you find yourself becoming the least bit confused about the subject, just ignore it completely. For many casual, personal uses of *J/Link*, you can forget about memory management issues, and just let Java objects pile up.

J/Link includes a number of notebooks with sample programs, including most of the programs developed in this section. These notebooks can be found in the <Mathematica dir>/SystemFiles/Links/JLink/Examples/Part1 directory.

A Beep Function

Here is a very simple example. *Mathematica* does not have a `Beep` function to provide simple alerts. But Java has a `beep()` method and, by virtue of that, *Mathematica* has one too.

```
Beep[] :=
 (
  LoadJavaClass["java.awt.Toolkit"];
  Toolkit`getDefaultToolkit[]@beep[]
 )
```

You will notice a short delay the first time `Beep[]` is executed. This is due to the `LoadJavaClass` call, which only takes measurable time the first time it is called for any given class.

```
Beep[]
```

This is a perfectly good beep function, and many users will not need to go beyond this. If you are writing code for others to use, however, you will probably want to embellish this code a little bit. Here is a more professional version of the same function.

```
BetterBeep[]:=
    JavaBlock[
        InstallJava[];
        LoadJavaClass["java.awt.Toolkit"];
        Toolkit`getDefaultToolkit[]@beep[];
    ]
```

Note that the first thing you do is call `InstallJava`. It is a good habit to call `InstallJava` in functions that use *J/Link*, at least if you are writing code for others to use. If `InstallJava` has already been called, subsequent calls will do nothing and return very quickly. The whole program is wrapped in `JavaBlock`. As discussed in the section "JavaBlock", `JavaBlock` automates the process of releasing references to objects returned to *Mathematica*. The `getDefaultToolkit()` method returns a `Toolkit` object, so you want to release the `JavaObject` that gets created in *Mathematica*. The `getDefaultToolkit()` method returns a reference to the same `Toolkit` object every time it is called, so even if you do not call `JavaBlock`, you will only "leak" one object in an entire session. You could also write `Beep` using an explicit call to `ReleaseJavaObject`.

```
(* Alternative version *)
BetterBeep2[]:=
    Module[{toolkit},
        InstallJava[];
        LoadJavaClass["java.awt.Toolkit"];
        toolkit = Toolkit`getDefaultToolkit[];
        toolkit@beep[];
        ReleaseJavaObject[toolkit]
    ]
```

The advantage to using JavaBlock is that you do not have to think about what, if any, methods might return objects, and you do not have to assign them to variables.

Formatting Dates

Here is an example of a computation performed in Java. Java provides a number of powerful date- and calendar-oriented classes. Say you want to create a nicely formatted string showing the time and date. In this first step you create a new Java Date object representing the current date and time.

```
date = JavaNew["java.util.Date"]
```

```
«JavaObject[java.util.Date] »
```

Next you load the DateFormat class and create a formatter capable of formatting dates.

```
LoadJavaClass["java.text.DateFormat"];
dateFormatter = DateFormat`getInstance[]
```

```
«JavaObject[java.text.SimpleDateFormat] »
```

Now you call the format() method, passing the Date object as its argument.

```
dateFormatter@format[date]
```

```
10/9/00 4:56 AM
```

There are many different ways in which dates and times can be formatted, including respecting a user's locale. Java also has a useful number-formatting class, an example of which was given in "An Optimization Example".

A Progress Bar

A simple example of a popup user interface for a *Mathematica* program is a progress bar. This is an example of a "non-interactive" user interface, as defined in "Interactive and Non-Interactive Interfaces", because it does not need to call back to *Mathematica* or return a result to *Mathematica*. The implementation uses the Swing user interface classes, because Swing has a built-in class for progress bars. (You cannot run this example unless you have Swing installed. It comes as a standard part of Java 1.2 or later, but you can get it separately for Java 1.1.x. Most Java development tools that are still at Version 1.1.x come with Swing.) The complete code for this example is also provided in the file ProgressBar.nb in the JLink/Examples/Part1 directory.

The code is commented to point out the general structure. There are several classes and methods used in this code that may be unfamiliar to you. Just keep in mind that this is completely standard Java code translated into *Mathematica* using the *J/Link* conventions. It is line-for-line identical to a Java program that does the same thing.

This code is presented as a complete program, but this does not suggest that it should be developed that way. The interactive nature of *J/Link* lets you tinker with Java objects a line at a time, experimenting until you get things just how you want them. Of course, this is how *Mathematica* programs are typically written, and *J/Link* lets you do the same with Java objects and methods.

You can create a function `ShowProgressBar` that prepares and displays a progress bar dialog. The bar will be used to show percentage completion of a computation. You can supply the initial percent completed or use the default value of zero. `ShowProgressBar` returns the `JProgressBar` object because the bar needs to be updated later by calling `setValue()`. Note that because you return the `bar` object from the `JavaBlock`, it is not released like all other new Java objects created within this `JavaBlock`. This is a new behavior of `JavaBlock` in *J/Link* 2.0. If what is returned from a `JavaBlock` is precisely a single Java object (and not, for example, a list of objects), then this object is not released. `JavaBlock` is discussed in the section "JavaBlock".

```
ShowProgressBar[title_String:"Computation Progress",
                  caption_String:"Percent complete:",
                  percent_Integer:0
                ] :=
    JavaBlock[
        Module[{frame, panel, label, bar},
            InstallJava[];
            bar = JavaNew["javax.swing.JProgressBar"];
            frame = JavaNew["javax.swing.JFrame", title];
            frame@setSize[300, 110];
            frame@setResizable[False];
            frame@setLocation[400, 400];
            panel = JavaNew["javax.swing.JPanel"];
            panel@setLayout[Null];
            frame@getContentPane[]@add[panel];
            label = JavaNew["javax.swing.JLabel", caption];
            label@setBounds[20, 10, 260, 20];
            panel@add[label];
            bar@setBounds[20, 40, 260, 30];
            bar@setMinimum[0];
            bar@setMaximum[100];
            bar@setValue[percent];
            panel@add[bar];
            JavaShow[frame];
            bar
        ]
    ]
```

You also need a function to close the progress dialog and clean up after it. Only two things need to be done. First, the `dispose()` method must be called on the top-level frame window that contains the bar. Second, if you want to avoid leaking object references, you need to call `ReleaseJavaObject` on the bar object because it is the only object reference that escaped the `JavaBlock` in `ShowProgressBar`. You need to call `dispose()` on the `JFrame` object you created in `ShowProgressBar`, but you did not save a reference to it. The `SwingUtilities` class has a handy method `windowForComponent()` that will retrieve this frame, given the bar object.

```
DestroyProgressBar[bar_?JavaObjectQ] :=
    JavaBlock[
        LoadJavaClass["javax.swing.SwingUtilities"];
        SwingUtilities`windowForComponent[bar]@dispose[];
        ReleaseJavaObject[bar]
    ]
```

The bar dialog has a close box in it, so a user can dismiss it prematurely if desired. This would take care of disposing the dialog, but you would still need to release the bar object. `DestroyProgressBar` (and the bar's `setValue()` method) is safe to call whether or not the user closed the dialog.

Here is how you would use the progress bar in a computation. The call to `ShowProgressBar` displays the bar dialog and returns a reference to the bar object. Then, while the computation is running, you periodically call the `setValue()` method to update the bar's appearance. When the computation is done, you call `DestroyProgressBar`.

```
bar = ShowProgressBar[];
n = 0;
While[n <= 5,
    bar@setValue[n/5 * 100];
    Pause[1]; (* This simulates the time-consuming computation. *)
    n++
];
DestroyProgressBar[bar];
```

An easy way to test whether your code leaks object references is to call `LoadedJavaObjects[]` before and after the computation. If the list of objects gets longer, then you have forgotten to use `ReleaseJavaObject` or improperly used `JavaBlock`.

It can take several seconds to load all the Swing classes used in this example. This means that the first time `ShowProgressBar` is called, there will be a significant delay. You could avoid this delay by using `LoadJavaClass` ahead of time to explicitly load the classes that appear in `JavaNew` statements.

The dialog appears onscreen with its upper left at the coordinates (400, 400). It is left as an exercise to the reader to make it centered on the screen. (Hint: the `java.awt.Toolkit` class has a `getScreenSize()` method).

Finally, because the progress bar uses the Swing classes, you can play with the look-and-feel options that Swing provides. Specifically, you can change the theme at runtime. The progress bar window is not very complicated, so it changes very little in going from one look-and-feel theme to another, but this demonstrates how to do it. The effect is much more dramatic for more complex windows.

First, create a new progress bar window.

```
bar = ShowProgressBar[];
```

Now load some classes from which you need to call static methods.

```
LoadJavaClass["javax.swing.UIManager"];
LoadJavaClass["javax.swing.SwingUtilities"];
```

The default look and feel is the "metal" theme. You can change it to the native style look for your platform as follows (it helps to be able to see the window when doing this).

```
JavaBlock[
    UIManager`setLookAndFeel[UIManager`getSystemLookAndFeelClassName[]];
    frame = SwingUtilities`windowForComponent[bar];
    SwingUtilities`updateComponentTreeUI[frame]
]
```

Clean up.

```
DestroyProgressBar[bar]
```

A Simple Modal Input Dialog

You saw one example of a simple modal dialog in "Modal Windows". Presented here is another one—a basic dialog that prompts the user to enter an angle, with a choice of whether it is being specified in degrees or radians. This will demonstrate a dialog that returns a value to a running *Mathematica* program when it is dismissed, much like *Mathematica*'s built-in `Input` function, which requests a string from the user before returning. Dialogs like this one are not "modal" in the traditional sense that they must be closed before other Java windows can be used, but rather they are modal with respect to the kernel, which is kept busy until they are dismissed (that is, until `DoModal[]` returns). The section "Creating Windows and Other User Interface Elements" discusses modal and modeless Java windows in detail.

The code is rather straightforward and warrants little in the way of commentary. In creating the window and the controls within it, it exactly mirrors the Java code you would use if you were writing the program in Java. One technique it demonstrates is determining whether the **OK** or

Cancel button was clicked to dismiss the dialog. This is done by having the `MathActionListener` objects assigned to the two buttons return different things in addition to calling `EndModal[]`. Recall that `DoModal[]` returns whatever the code that calls `EndModal[]` returns, so here you have the **OK** button execute `(EndModal[]; True)&`, a pure function that ignores its arguments, calls `EndModal[]`, and returns `True`, whereas the **Cancel** button executes `(EndModal[]; False)&`. Thus, `DoModal[]` returns `True` if the **OK** button was clicked, or `False` if the **Cancel** button was clicked. It will return `Null` if the window's close box was clicked (this behavior comes from the `MathFrame` itself).

It may take several seconds to display the dialog the first time `GetAngle[]` is called. This is due to the one-time cost of loading the several large AWT classes required. Subsequent invocations of `GetAngle[]` will be much quicker.

The complete code for this example is also provided in the file ModalInputDialog.nb in the JLink/Examples/Part1 directory.

```
GetAngle[] :=
    JavaBlock[
        Module[{frm, inputField, cbGroup, degBox, radBox,
                label, okButton, cancelButton, wasOKButton, angle},

            InstallJava[]; (* In case the user has not called it already. *)

            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            label = JavaNew["java.awt.Label", "Enter an angle:"];
            inputField = JavaNew["java.awt.TextField"];
            cbGroup = JavaNew["java.awt.CheckboxGroup"];
            degBox = JavaNew["java.awt.Checkbox", "degrees", cbGroup, True];
            radBox = JavaNew["java.awt.Checkbox", "radians", cbGroup, False];
            okButton = JavaNew["java.awt.Button", "OK"];
            cancelButton = JavaNew["java.awt.Button", "Cancel"];

            frm@setLayout[Null];
            frm@add[label];
            frm@add[inputField];
            frm@add[degBox];
            frm@add[radBox];
            frm@add[okButton];
            frm@add[cancelButton];

            frm@setBounds[200, 200, 200, 160];
            label@setBounds[20, 30, 150, 20];
            inputField@setBounds[20, 70, 60, 28];
            degBox@setBounds[100, 60, 80, 20];
            radBox@setBounds[100, 80, 80, 20];
            okButton@setBounds[40, 120, 50, 20];
            cancelButton@setBounds[100, 120, 50, 20];
            frm@setResizable[False];

            okButton@addActionListener[
                JavaNew["com.wolfram.jlink.MathActionListener",
```

```
                        "(EndModal[]; True)&"]
                ];
                cancelButton@addActionListener[
                    JavaNew["com.wolfram.jlink.MathActionListener",
                        "(EndModal[]; False)&"]
                ];

                (* Now make the window visible and bring it to the foreground. *)
                JavaShow[frm];

                frm@setModal[];
                wasOKButton = DoModal[];
                (* Even though the window may have been closed, it is perfectly
                   OK to extract values from the controls in the window.
                *)
                If[TrueQ[wasOKButton],
                    angle = ToExpression[inputField@getText[]];
                    If[angle =!= Null && degBox@getState[], angle *= Pi/180],
                (* else *)
                    (* We will get here if the Cancel button was clicked
                       (wasOKButton will be False), or the dialog was closed
                       by clicking in its close box (wasOKButton will be Null).
                    *)
                    angle = $Failed
                ];
                (* If the cancel or OK buttons were clicked, frm is still
                   visible, so we dispose it here.
                *)
                frm@dispose[];
                angle
            ]
        ]
```

Now invoke it.

```
GetAngle[]
```

π

A File Chooser Dialog Box

A useful feature for *Mathematica* programs is to be able to produce a file chooser dialog, such as the typical **Open** or **Save** dialog boxes. You could use such a dialog box to prompt a user for an input file or a file into which to write data. This is easily accomplished in a cross-platform way with Java, specifically with the `JFileChooser` class in the standard Swing library. The code for such a dialog box is provided in the file FileChooserDialog.nb in the JLink/Examples/Part1 directory.

Mathematica 4.0 introduced a new "experimental" function called `FileBrowse[]` that displays a file browser in the front end. Although this function is usable, it has several shortcomings compared to the Java technique presented next. One of the limitations is that it requires that the front end be in use. Another is that it is not customizable, so you always get a **Save file as:**

dialog box and the concomitant behavior, which is not appropriate for an **Open**-type dialog box. The `JFileChooser` class used here allows very sophisticated customization, including setting the initial directory, masking out files based on their names or properties, controlling the title and text on the various buttons, supplying functions to validate the choice before the dialog box is allowed to be dismissed, allowing for multiple file selection, and allowing directories to be selected instead of files.

Although this example is a short program, the code has some unfortunate complexity (meaning "ugliness") in it related to making this special type of dialog window come to the foreground on all platforms. For this reason, the code is not presented here. Instead, some topics in the program code will be mentioned; you can read the full code and its associated comments in the example file if you are interested in the implementation details.

The `FileChooserDialog` function takes three string arguments. The first is the title of the dialog box (for example, **Select a data file to import**), the second is the text to appear on what is essentially the **OK** button (typically this will be **Open** or **Save**), and the third is the directory in which to start. You can also supply no arguments and get a default Open dialog box that starts in the kernel's current directory.

Although this is a "modal" dialog box, there is no need to use `DoModal`, because the `showDialog()` method will not return until the user dismisses the dialog box. Recall that `DoModal` is a way to force *Mathematica* to stall until the dialog box or other window is dismissed. Here, you get this behavior for free from `showDialog()`. The other thing that `DoModal` does is put the kernel into a loop where it is ready to receive input from Java, so you can script some of the functionality of the dialog via callbacks to *Mathematica*. The file chooser dialog box does not need to use *Mathematica* in any way until it returns the selected file, so you have no need for this other aspect that `DoModal` provides.

A second point of interest is in the name of the constant that `showDialog()` returns to indicate that the user clicked the **Save** or **Open** button instead of the **Cancel** button. The name of this constant in Java is `JFileChooser.APPROVE_OPTION`. Java names map to *Mathematica* symbols, so they must be translated if they contain characters that are not legal in *Mathematica* symbols, such as the underscore. Underscores are converted to a "U" when they appear in symbols, so the *Mathematica* name of this constant is `JFileChooser`APPROVEUOPTION`. See "Underscores in Java Names" for more information.

Sharing the Front End: Palette-Type Buttons

As discussed in the section "Creating Windows and Other User Interface Elements", one of the goals of *J/Link* is to allow Java user interface elements to be as close as possible to first-class members of the notebook front end environment in the way notebook and palette windows are. One of the ways this is accomplished is with the `ShareKernel` function, which allows Java windows to share the kernel's attention with notebook windows. Such Java windows are referred to as "modeless," not in the traditional sense of allowing other Java windows to remain active, but modeless with respect to the kernel, meaning that the kernel is not kept busy while they are open.

Beyond the ability to have Java windows share the kernel with the front end, it would be nice to allow actions in Java to cause effects in notebook windows, such as printing something, displaying a graph, or any of the notebook-manipulation commands like `NotebookApply`, `NotebookPrint`, `SelectionEvaluate`, `SelectionMove`, and so on. A good example of this is palette buttons. A palette button can cause the current selection to be replaced by something else and the resulting expression to be evaluated in place.

The `ShareFrontEnd` function lets actions in Java modeless windows trigger events in a notebook window just like can be done from palette buttons or *Mathematica* code you evaluate manually in a notebook. Remember that you get automatically the ability to interact with the front end when you use a *modal* dialog (i.e., when `DoModal` is running). When Java is being run in a modal way, the kernel's `$ParentLink` always points at the front end, so all side effect outputs get sent to the front end automatically. A modal window would not be acceptable for the palette example here because the palette needs to be an unobtrusive enhancement to the *Mathematica* environment—it cannot lock up the kernel while it is alive. `ShareKernel` allows Java windows to call *Mathematica* without tying up the kernel, and `ShareFrontEnd` is an extension to `ShareKernel` (it calls `ShareKernel` internally) that allows such "modeless" Java windows to interact with the front end. `ShareFrontEnd` is discussed in more detail in "Sharing the Front End".

In the `PrintButton` example that follows, a simple palette-type button is developed in Java that prints its label at the current cursor position in the active notebook. Because of current limitations with `ShareFrontEnd`, this example will not work with a remote kernel; the same machine must be running the kernel and the front end.

```
PrintButton[label_String] :=
    JavaBlock[
        Module[{frm, button, listener, tok},
            InstallJava[];
            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            button = JavaNew["java.awt.Button"];
            frm@add[button];
            frm@pack[];
            button@setLabel[label];
            listener = JavaNew["com.wolfram.jlink.MathActionListener",
                               "printButtonFunc"];
            button@addActionListener[listener];
            tok = ShareFrontEnd[];
            frm@onClose["UnshareFrontEnd[" <> ToString[tok] <> "]"];
            JavaShow[frm]
        ]
    ]

printButtonFunc[event_, _] :=
    JavaBlock[
        NotebookApply[SelectedNotebook[], event@getSource[]@getLabel[]];
        (* We need to explicitly release the event object, since it was
           sent to Mathematica before the JavaBlock was entered. *)
        ReleaseJavaObject[event]
    ]
```

Now invoke the `PrintButton` function to create and display the palette. Click the button to see the button's label (`foo` in this example) inserted at the current cursor location. When you are done, click the window's close box.

```
PrintButton["foo"]
```

The code is mostly straightforward. As usual, you use the `MathFrame` class for the frame window because it closes and disposes of itself when its close box is clicked. You create a `MathActionListener` that calls `buttonFunc` and you assign it to the button. From the table in the section Handling Events with *Mathematica* Code: The "MathListener" Classes, you know that `buttonFunc` will be called with two arguments, the first of which is the `ActionEvent` object. From this object you can obtain the button that was clicked and then its label, which you insert at the current cursor location using the standard `NotebookApply` function. One subtlety is that you need to specify `SelectedNotebook[]` as the target for notebook operations like `NotebookApply`, `NotebookWrite`, `NotebookPrint`, and so on, which take a notebook as an argument. Because of implementation details of `ShareFrontEnd`, the notebook given by `EvaluationNotebook[]` is not the correct target (after all, there is no evaluation currently in progress in the front end when the button is clicked).

The important thing to note in PrintButton is the use of ShareFrontEnd and UnshareFrontEnd. As discussed earlier, ShareFrontEnd puts Java into a state where it forwards everything other than the result of a computation to the front end, and puts the front end into a state where it is able to receive it. This is why the Print output triggered by clicking the Java button, which would normally be sent to Java (and just discarded there), appears in the front end. Front end sharing (and also kernel sharing) should be turned off when they are no longer needed, but if you are writing code for others to use you cannot just blindly shut sharing down—the user could have other Java windows open that need sharing. To handle this issue, ShareFrontEnd (and ShareKernel) works on a register/unregister principle. Every time you call ShareFrontEnd, it returns a token that represents a request for front end sharing. If front end sharing is not on, it will be turned on. When a program no longer needs front end sharing, it should call UnshareFrontEnd, passing the token from ShareFrontEnd as the argument. Only when all requests for sharing have been unregistered in this way will sharing actually be turned off.

The onClose() method of the MathFrame class lets you specify *Mathematica* code to be executed when the frame is closed. This code is executed after all event listeners have been notified, so it is a safe place to turn off sharing. In the onClose() code, you call UnshareFrontEnd with the token returned by ShareFrontEnd. Using the onClose() method in this way allows us to avoid writing a cleanup function that users would have to call manually after they were finished with the palette. It is not a problem to leave front end sharing turned on, but it is desirable to have your program alter the user's session as little as possible.

Now expand this example to include more buttons that perform different operations. The complete code for this example is provided in the file Palette.nb in the JLink/Examples/Part1 directory.

The first thing you do is separate the code that manages the frame containing the buttons from the code that produces a button. In this way you will have a reusable palette frame that can hold any number of different buttons. The ShowPalette function here takes a list of buttons, arranges them vertically in a frame window, calls ShareFrontEnd, and displays the frame in front of the user's notebook window.

```
ShowPalette[buttons:{__?JavaObjectQ}] :=
    JavaBlock[
        Module[{frm, tok},
            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            frm@setLayout[JavaNew["java.awt.GridLayout", 0, 1]];
            frm@add[#]& /@ buttons;
            ReleaseJavaObject[buttons];
            frm@pack[];
            tok = ShareFrontEnd[];
            frm@onClose["UnshareFrontEnd[" <> ToString[tok] <> "]"];
            JavaShow[frm];
        ]
    ]
```

Note that you do not return anything from the `ShowPalette` function—specifically, you do not return the frame object itself. This is because you do not need to refer to the frame ever again. It is destroyed automatically when its close box is clicked (remember, this is a feature of the `MathFrame` class). Because you do not need to keep references to any of the Java objects you create, the entire body of `ShowPalette` can be wrapped in `JavaBlock`.

Now create a reusable `PaletteButton` function that creates a button. You have to pass in only two things: the label text you want on the button and the function (as a string) you want to have invoked when the button is clicked. This is sufficient to allow completely arbitrary button behavior, as the entire functionality of the button is tied up in the button function you pass in as the second argument.

```
PaletteButton[label_String, buttonFunc_String] :=
    JavaBlock[
        Module[{button, listener},
            button = JavaNew["java.awt.Button"];
            button@setLabel[label];
            listener = JavaNew["com.wolfram.jlink.MathActionListener", buttonFunc]
            button@addActionListener[listener];
            button
        ]
    ]
```

You will use the `PaletteButton` function to create four buttons. The first is just the print button just defined, the behavior of which is specified by `printButtonFunc`.

```
btn1 = PaletteButton["foo", "printButtonFunc"];
```

The second will duplicate the functionality of the buttons in the standard **AlgebraicManipulation** front end palette. These buttons wrap a function (e.g., `Expand`) around the current selection and evaluate the resulting expression in place. Here is how you create the button and define the button function for that operation.

```
btn2 = PaletteButton["Expand[■]", "applyButtonFunc"];

applyButtonFunc[event_, _] :=
    JavaBlock[
        With[{nb = SelectedNotebook[]},
            NotebookApply[nb, event@getSource[]@getLabel[], All];
            ReleaseJavaObject[event];
            SelectionEvaluate[nb]
        ];
    ]
```

The third button will create a plot. All you have to do is call a plotting function—the work of directing the graphics output to a new cell in the frontmost notebook is handled internally by *J/Lin*k as a result of having front end sharing turned on via `ShareFrontEnd`.

```
btn3 = PaletteButton["Create Plot", "plotButtonFunc"];

plotButtonFunc[event_, _] :=
    (
        Plot[x, {x, 0, 1}];
        ReleaseJavaObject[event];
    )
```

The final button demonstrates another method for causing text to be inserted at the current cursor location. The first example of this, `printButtonFunc`, uses `NotebookApply`. You can also just call `Print`—as with graphics, `Print` output is automatically routed to the frontmost notebook window by *J/Link* when front end sharing is on. This quick-and-easy `Print` method works fine for many situations when you want something to appear in a notebook window, but using `NotebookApply` is a more rigorous technique. You will see some differences in the effects of these two buttons if you put the insertion point into a `StandardForm` cell and try them.

```
btn4 = PaletteButton["foo", "printButtonFunc2"];

printButtonFunc2[event_, _] :=
    JavaBlock[
        Print[event@getSource[]@getLabel[]];
        ReleaseJavaObject[event];
    ]
```

Now you are finally ready to create the palette and show it.

```
ShowPalette[{btn1, btn2, btn3, btn4}]
```

In closing, it must be noted that although this example has demonstrated some useful techniques, it is not a particularly valuable way to use `ShareFrontEnd`. In creating a simple palette of buttons, you have done nothing that the front end cannot do all by itself. The real uses you will find for `ShareFrontEnd` will presumably involve aspects that cannot be duplicated within the front end, such as more sophisticated dialog boxes or other user interface elements.

Real-Time Algebra: A Mini-Application

This example will put together everything you have learned about modal and modeless Java user interfaces. You will implement the same "mini-application" (essentially just a dialog box) in both modal and modeless flavors. The application is inspired by the classic *MathLink* example program RealTimeAlgebra, originally written for the NeXT computer by Theodore Gray and then done in HyperCard by Doug Stein and John Bonadies. The original RealTimeAlgebra provides an input window into which the user types an expression that depends on certain parameters, an output window that displays the result of the computation, and some sliders that are used to vary the values of the parameters. The output window updates as the sliders are moved, hence the name RealTimeAlgebra. Our implementation of RealTimeAlgebra will be very simplistic, with only a single slider to modify the value of one parameter.

The complete code for this example is provided in the file RealTimeAlgebra.nb in the JLink/Examples/Part1 directory.

Here is the function that creates and displays the window.

```
CreateWindow[] :=
    Module[{frame, slider, listener},
        InstallJava[];
        (* inText and outText are globals, because we need to refer to
           them by name in the scrollFunc. This also means we must
           create them outside the JavaBlock below.
        *)
        inText = JavaNew["java.awt.TextArea", "Expand[(x+1)^a]", 8, 40];
        outText = JavaNew["java.awt.TextArea", 8, 40];
        (* This frame could be created inside the JavaBlock, because it is returned
           from the JavaBlock and therefore will not be released, but it makes
           our intentions more clear to create it outside.
        *)
        frame = JavaNew["com.wolfram.jlink.MathFrame", "RealTimeAlgebra"];
        JavaBlock[
            frame@setLayout[JavaNew["java.awt.BorderLayout"]];
            (* Note that we can refer to the Scrollbar`HORIZONTAL constant within the JavaNew
               command that first loads the Scrollbar class. Its value will not need to be
               resolved until that class has been loaded and all necessary definitions created.
            *)
            slider = JavaNew["java.awt.Scrollbar", Scrollbar`HORIZONTAL, 0, 1, 0, 20];
            frame@add[slider, ReturnAsJavaObject[BorderLayout`NORTH]];
            frame@add[outText, ReturnAsJavaObject[BorderLayout`CENTER]];
            frame@add[inText, ReturnAsJavaObject[BorderLayout`SOUTH]];
            frame@pack[];
            (* Use a fixed-width font for the output window to preserve
               formatting of multi-line expressions. *)
            outText@setFont[JavaNew["java.awt.Font", "Courier", Font`PLAIN, 12]];
            listener = JavaNew["com.wolfram.jlink.MathAdjustmentListener"];
            listener@setHandler["adjustmentValueChanged", "sliderFunc"];
            slider@addAdjustmentListener[listener];
            frame@setLocation[200, 200];
            JavaShow[frame];
        ];
        frame
    ]

(* This is what will be called in response to moving the slider position: *)

sliderFunc[evt_, type_, scrollPos_] :=
    outText@setText[
        Block[{a = scrollPos}, ToString[ToExpression[inText@getText[]]]]
    ]
```

The `sliderFunc` function is called by the `MathAdjustmentListener` whenever the slider's position changes. It gets the text in the `inputText` box, evaluates it in an environment where `a` has the value of the slider position (the range for this is 0..20, as established in the `JavaNew` call that creates the slider), and puts the resulting string into the `outText` box. It then calls `ReleaseJavaObject` to release the first argument, which is the `AdjustmentEvent` object itself. This is the only object passed in as an argument (the other two arguments are integers). If you are wondering how you determine the argument sequence for `sliderFunc`, you get it from the `MathListener` table in the section Handling Events with *Mathematica* Code: The "MathListener"

Classes. Note that you need to refer by name to the input and output text boxes in `sliderFunc`, so you cannot make their names local variables in the `Module` of `CreateWindow`, and of course they cannot be created inside that function's `JavaBlock`.

There is one interesting thing in the code that deserves a remark. Look at the lines where you add the three components to the frame. What is the `ReturnAsJavaObject` doing there? The method being called here is in the `Frame` class, and has the following signature:

```
void add(Component comp, Object constraints);
```

The second argument, `constraints`, is typed only as `Object`. The value you pass in depends on the layout manager in use, but typically it is a string, as is the case here (`BorderLayout`NORTH`, for example, is just the string "NORTH"). The problem is that *J/Link* creates a definition for this signature of `add` that expects a `JavaObject` for the second argument, and *Mathematica* strings do not satisfy `JavaObjectQ`, although they are converted to Java string objects when sent. This means that you can only pass strings to methods that expect an argument of type `String`. In the rare cases where a Java method is typed to take an `Object` and you want to pass a string from *Mathematica*, you must first create a Java `String` object with the value you want, and pass that object instead of the raw *Mathematica* string. You have encountered this issue several times before, and you have used `MakeJavaObject` as the trick to get the raw string turned into a reference to a Java `String` object. `MakeJavaObject[BorderLayout`NORTH]` would work fine here, but it is instructive to use a different technique (it also saves a call into Java). `BorderLayout`NORTH` calls into Java to get the value of the `BorderLayout.NORTH` static field, but in the process of returning this string object to *Mathematica*, it gets converted to a raw *Mathematica* string. You need the object reference, not the raw string, so you wrap the access in `ReturnAsJavaObject`, which causes the string, which is normally returned by value, to be returned in the form of a reference.

Getting back to the **RealTimeAlgebra** dialog box, you are now ready to run it as a modal window. You write a special modal version that uses `CreateWindow` internally.

```
RealTimeAlgebraModal[] :=
    JavaBlock[
        (* In the modal case, we can wrap the whole thing in JavaBlock
           and be sure that all the objects will get released, including
           the inText and outText ones needed during event handling.
        *)
        Module[{frm},
            frm = CreateWindow[];
            frm@setModal[];
            DoModal[];
        ]
    ]
```

Note that the whole function is wrapped in JavaBlock. This is an easy way to make sure that all object references created in *Mathematica* while the dialog is running are treated as temporary and released when DoModal finishes. This saves you having to properly use JavaBlock and ReleaseJavaObject in all the handler functions used for your MathListener objects (you will notice that these calls are absent from the sliderFunc function).

Now run the dialog. The RealTimeAlgebraModal function will not return until you close the **RealTimeAlgebra** window, which is what you mean when you call this a "modal" interface.

```
RealTimeAlgebraModal[]
```

It may take several seconds before the window appears the first time. As always, this is the one-time cost of loading all the necessary classes. Play around by dragging the slider, and try changing the text in the input box, for example, to N[Pi, 2 *a*].

Recall that while *Mathematica* is evaluating DoModal[], any Print output, messages, graphics, or any other output or commands other than the result of computations triggered from Java will be sent to the front end. To see this in action, try putting Print[*a*] in the input text box (you will want to arrange windows on your screen so that you can see the notebook window while you are dragging the slider). Next, try Plot[Sin[*a x*], {*x*, 0, 4 Pi}].

Quit RealTimeAlgebra by clicking the window's close box. In addition to closing and disposing of the window, this causes EndModal[] to be executed in *Mathematica*, which then causes DoModal to return. The disposing of the window is due to using the MathFrame class for the window, and executing EndModal[] is the result of calling the setModal() method of MathFrame, as discussed in "Modal Windows".

Now implement RealTimeAlgebra as a modeless window. The CreateWindow function can be used unmodified. The only difference is how you make *Mathematica* able to service the computations triggered by dragging the slider. For a modal window, you use DoModal to force *Mathematica* to pay attention exclusively to the Java link. The drawback to this is that you cannot use the kernel from the notebook front end until DoModal ends. To allow the notebook front end and Java to share the kernel's attention, you use ShareKernel.

```
RealTimeAlgebraModeless[] :=
    Module[{frm, token},
        frm = CreateWindow[];

token = ShareKernel[];

(* We use the MathFrame onClose method to specify code to
          be executed when the frame is closed.The use here is
          typical--we clean up the object references that need to
          persist throughout the lifetime of the window (otherwise
          we would leak these references),and we call UnshareKernel
          to unregister this application's request for kernel sharing.
       *)
       frm@onClose[
          "ReleaseJavaObject[inText, outText]; UnshareKernel[" <> ToString[token] <> "];"
       ];

ReleaseJavaObject[frm]
    ]
```

Now run it.

```
RealTimeAlgebraModeless[]
```

`RealTimeAlgebraModeless` returns immediately after the window is displayed, leaving the front end and the **RealTimeAlgebra** window able to use the kernel for computations.

You still need a little bit of polish on the modeless version, however. First, to avoid leaking object references, you must change `sliderFunc`. With the modal version, you did not bother to use `JavaBlock` or `ReleaseJavaObject` in `sliderFunc` because you had `DoModal` wrapped in `JavaBlock`. Every call to `sliderFunc`, or any other `MathListener` handler function, occurs entirely within the scope of `DoModal`, so you can handle all object releasing at this level. With a modeless interface, you no longer have a single function call that spans the lifetime of the window. Thus, you put memory-management functions in our handler functions. Here is the new `sliderFunc`.

```
sliderFunc[evt_, type_, scrollPos_] :=
    JavaBlock[
        outText@setText[

    Block[{a = scrollPos}, ToString[ToExpression[inText@getText[]]]]
        ];
        ReleaseJavaObject[evt]
    ]
```

The `JavaBlock` here is unnecessary because the code it wraps creates no new object references. Out of habit, though, you wrap these handlers in `JavaBlock`. You need to explicitly call `ReleaseJavaObject` on `evt`, which is the `AdjustmentEvent` object, because its reference is created in *Mathematica* before `sliderFunc` is entered, so it will not be released by the `JavaBlock`. The `type` and `scrollPos` arguments are integers, not objects.

Try setting the input text to `Print[a]`. Notice that nothing appears in the front end when you move the slider, in contrast to the modal case. With a modeless interface, the Java link is the kernel's `$ParentLink` during the times when the kernel is servicing a request initiated from the Java side. Thus, the output from `Print` and graphics goes to Java, not the notebook front end. (The Java side ignores this output, in case you are wondering.) To get this output sent to the front end instead, use `ShareFrontEnd`.

```
ShareFrontEnd[];
```

Now if you set the input text to, say, `Print[a]` or `Plot[a x, {x, 0, a}]`, you will see the text and graphics appearing in the front end.

When you are finished, quit `RealTimeAlgebra` by clicking its close box. Then turn off front end sharing that was turned on in the previous input.

```
UnshareFrontEnd[]
```

A modal interface is simpler than a modeless one in terms of how it uses *Mathematica*, and is therefore the preferred method unless you specifically need the modeless attribute. `ShareKernel` and `ShareFrontEnd` are complex functions that put the kernel into an unusual state. They work fine, but do not use them unnecessarily.

GraphicsDlg: Graphics and Typeset Output in a Window

It is useful to be able to display *Mathematica* graphics and typeset expressions in your Java user interface, and this is easy to do using *J/Link*'s `MathCanvas` class. This example demonstrates a simple dialog box that allows the user to type in a *Mathematica* expression and see the output in the form of a picture. If the expression is a plotting or other graphics function, the resulting image is displayed. If the expression is not a graphic, then it is typeset in `TraditionalForm` and displayed as a picture. The example is first presented in modal form and then in modeless form using `ShareKernel` and `ShareFrontEnd`.

This example also demonstrates a trivial example of using Java code that was created by a drag-and-drop GUI builder of the type present in most Java development environments. For layout of simple windows, it is easy enough to do everything from *Mathematica*. This method was chosen for all the examples in this tutorial, writing no Java code and instead scripting the creation and layout of controls in windows with *Mathematica* calls into Java. This has the advantage of not requiring any Java classes to be written and compiled. For more complex windows, however, you will probably find it much easier to create the controls, arrange them in position, set their properties in a GUI builder, and let it generate Java code for you. You might also want to write some additional Java code by hand.

If you choose this route, the question becomes, "How do I connect the Java code thus generated with *Mathematica*?" Any public fields or methods can be called directly from *Mathematica*, but your GUI builder may not have made public all the ones you need to use. You could make these fields and methods public or add some new public methods that expose them. The latter approach is probably preferable since it does not involve modifying the code that the GUI builder wrote, which could confuse the builder or cause it to overwrite your changes in future modifications.

The complete code for this example is provided in the JLink/Examples/Part1/GraphicsDlg directory. Some of the code is in Java.

This example uses the GUI builder in the WebGain Visual Café Java development environment. The builder was used to create a frame window with three controls. The frame window was made to be a subclass of `MathFrame` because you want to inherit the `setModal()` method. In the top left is an AWT `TextArea` that serves as the input box for the expression. To its right is an **Evaluate** button. Occupying the rest of the window is a `MathCanvas`.

Up to this point, no code has been written by hand at all—everything has been done automatically as components were dropped into the frame and their properties set. All that is left to do is to wire up the button so that when it is clicked the input text is taken and supplied as to the `MathCanvas` via its `setMathCommand()` method. You could write that code in Java, using Visual Café's Interaction Wizard to wire up this event (similar facilities exist in other Java GUI builders). You would have to write some Java code by hand, as the code's logic is more complex than can be handled by graphical tools for creating event handlers.

Rather than doing that, move to *Mathematica* to script the rest of the behavior because it is easier and more flexible. You will need to access the `TextArea`, `Button`, and `MathCanvas` objects from *Mathematica*, but the GUI builder made these nonpublic fields of the frame class. Thus, you need to add three public methods that return these objects to the frame class.

```
public Button getEvalButton()       {return evalButton;}
public TextArea getInputTextArea()  {return inputTextArea;}
public MathCanvas getMathCanvas()   {return mathCanvas;}
```

That is all you need to do to the Java code created by the GUI builder.

The GUI builder created a subclass of `MathFrame` that is named `GraphicsDlg`. It also gave it a `main()` method that does nothing but create an instance of the frame and make it visible. You will not bother with the `main()` method, choosing instead to do those two steps manually, since you need a reference to the frame.

Needed before the code is run is a demonstration of one more feature of *J/Link*—the ability to add directories to the class search path dynamically. You need to load the Java classes for this example, but they are not on the Java class path. With *J/Link*, you can add the directory in which the classes reside to the search path by calling `AddToClassPath`. This will work exactly as written in *Mathematica* 4.2 and later. You will need to modify the path if you have an earlier version of *Mathematica*.

```
classDir = ToFileName[{$TopDirectory, "SystemFiles", "Links", "JLink",
                       "Examples", "Part1", "GraphicsDlg"}];
InstallJava[];
AddToClassPath[classDir];
```

Here is the first implementation of the *Mathematica* code to create and run the graphics dialog. This runs the dialog in a modal loop.

```
DoGraphicsDialogModal[] :=
    JavaBlock[
        Module[{frm, btn, listener},
            InstallJava[];
            (* We named the MathFrame subclass GUI builder created "MyFrame". *)
            frm = JavaNew["GraphicsDlg"];
            (* Here we call one of the accessor methods we had to add
               by hand to the GraphicsDlg class.
            *)
            btn = frm@getEvalButton[];
            listener = JavaNew["com.wolfram.jlink.MathActionListener"];
            listener@setHandler["actionPerformed", "btnFunc"];
            btn@addActionListener[listener];
            JavaShow[frm];
            frm@setModal[];
            DoModal[]
        ]
    ]

btnFunc[event_, _] :=
    JavaBlock[
        Module[{frm, expr, textArea, inputText, mathCanvas},
            frm = event@getSource[]@getParent[];
            (* Here we call two of the accessor methods we had to add
               by hand to the GraphicsDlg class.
            *)
            textArea = frm@getInputTextArea[];
            mathCanvas = frm@getMathCanvas[];

            inputText = textArea@getText[];
            (* We have to evaluate the expression ahead of time to determine
               whether it is a graphics object or not, so we can decide
               whether it display it as a plot or as a typeset result.
            *)
            expr = Block[{$DisplayFunction = Identity}, ToExpression[inputText]];
            If[MatchQ[expr, _Graphics | _Graphics3D | _SurfaceGraphics |
                                DensityGraphics | _ContourGraphics],
                mathCanvas@setImageType[MathCanvas`GRAPHICS],
            (* else *)
                mathCanvas@setImageType[MathCanvas`TYPESET];
                mathCanvas@setUsesTraditionalForm[True]
            ];
            mathCanvas@setMathCommand[ToString[expr, InputForm]];
            ReleaseJavaObject[event]
        ]
    ]
```

As mentioned in the section "Creating Windows and Other User Interface Elements" only the notebook front end can perform the feat of taking a typeset (i.e., "box") expression and creating a graphical representation of it. Thus, the `MathCanvas` can render typeset expressions provided that it has a front end available to farm out the chore of creating the appropriate representation. The front end is used to run this example, but it is really because you are running the Java dialog "modally" that everything works the way it does. All the while the dialog is up, the front end is waiting for a result from a computation (`DoModal[]`), and therefore it is receptive to requests from the kernel for various services. As far as the front end is con-

cerned, the code for DoModal invoked the request for typesetting, even though it was actually triggered by clicking a Java button.

Now run the dialog.

```
DoGraphicsDialogModal[]
```

What if you are not happy with the restriction of running the dialog modally? Now you want to have the dialog remain open and active while not interfering with normal use of the kernel from the front end. As discussed in "Modal Windows" and "Real-Time Algebra: A Mini-Application", you get a lot of useful behavior regarding the front end for free when you run your Java user interface modally. One of these features is that the front end is kept receptive to the various sorts of requests the kernel can send to it (such as for typesetting services). You know you can run a Java user interface in a "modeless" way by using ShareKernel, but then you give up the ability to have the kernel use the front end during computations initiated by actions in Java. Luckily, the ShareFrontEnd function exists to restore these features for modeless windows.

Re-implement the graphics dialog in modeless form.

```
DoGraphicsDialogModeless[] :=
    JavaBlock[
        Module[{frm, btn, listener, tok},
            InstallJava[];
            frm = JavaNew["GraphicsDlg"];
            btn = frm@getEvalButton[];
            listener = JavaNew["com.wolfram.jlink.MathActionListener"];
            listener@setHandler["actionPerformed", "btnFunc"];
            btn@addActionListener[listener];
            tok = ShareFrontEnd[];
            frm@onClose["UnshareFrontEnd[" <> ToString[tok] <> "]"];
            JavaShow[frm]
        ]
     ]
```

The code shown is exactly the same as DoGraphicsDialogModal except for the last few lines. You call ShareFrontEnd here instead of setModal and DoModal. That is the only difference—the rest of the code (including btnFunc) is exactly the same. Notice also that you use the onClose() method of MathCanvas to execute code that unregisters the request for front end sharing when the window is closed.

Run the modeless version. Note how you can continue to perform computations in the front end while the window is active.

```
DoGraphicsDialogModeless[]
```

This new version functions exactly like the modeless version except that it does not leave the front end hanging in the middle of a computation. It is interesting to contrast what happens if you turn off front end sharing (but you need to leave kernel sharing on or the Java dialog will break completely). You can do this by replacing `ShareFrontEnd` and `UnshareFrontEnd` in `DoGraphicsDialogModeless` with `ShareKernel` and `UnshareKernel`. Now if you use the dialog you will find that it fails to render typeset expressions, producing just a blank window, but it still renders graphics normally (unless they have some typeset elements in them, such as a plot label). All the functionality is kept intact except for the ability of the kernel to make use of the front end for typesetting services.

BouncingBalls: Drawing in a Window

This example demonstrates drawing in Java windows using the Java graphics API directly from *Mathematica*. It also demonstrates the use of the `ServiceJava` function to periodically allow event handler callbacks into *Mathematica* from Java. The issues surrounding `ServiceJava` and how it compares to `DoModal` and `ShareKernel` are discussed in greater detail in "Manual" Interfaces: The ServiceJava Function.

The full code is a little too long to include here in its entirety, but it is available in the sample file BouncingBalls.nb in the JLink/Examples/Part1 directory. Here is an excerpt that demonstrates the use of `ServiceJava`.

```
...
mwl = JavaNew["com.wolfram.jlink.MathWindowListener"];
mwl@setHandler["windowClosing", "(keepOn = False)&"];
mathCanvas@addWindowListener[mwl];
keepOn = True;
While[keepOn,
    g@setColor[bkgndColor];
    g@fillRect[0, 0, 300, 300];
    drawBall[g, #]& /@ balls;
    mathCanvas@setImage[offscreen];
    balls = recomputePosition /@ balls;
    ServiceJava[]
];
...
```

A MathWindowListener is used to set keepOn = False when the window is closed, which will cause the loop to terminate. While the window is up, mouse clicks will cause new balls to be created, appended to the balls list, and set in motion. This is done with a MathMouseListener (not shown in the code). Thus, *Mathematica* needs to be able to handle calls originating from user actions in Java. As discussed in the section "Creating Windows and Other User Interface Elements", there are three ways to enable *Mathematica* to do this: DoModal (modal interfaces), ShareKernel or ShareFrontEnd (modeless interfaces), and ServiceJava (manual interfaces). A modal loop via DoModal would not be appropriate here because the kernel needs to be computing something at the same time it is servicing calls from Java (it is computing the new positions of the balls and drawing them). ShareKernel would not help because that is a way to give Java access to the kernel *between* computations triggered from the front end, not *during* such computations.

You need to periodically point the kernel's attention at Java to service requests if any are pending, then let the kernel get back to its other work. The function that does this is ServiceJava, and the code above is typical in that it has a loop that calls ServiceJava every time through. The calls from Java that ServiceJava will handle are the ones from mouse clicks to create new balls and when the window is closed.

Spirograph

This example is just a little fun to create an interesting, nontrivial application—an implementation of a simple Spirograph-type drawing program. It is run as a modal window, and it demonstrates drawing into a Java window from *Mathematica,* along with a number of MathListener objects for various event callbacks. It uses the Java Graphics2D API, so it will not run on systems that have only a Java 1.1.x runtime.

The code for this example can be found in the file Spirograph.nb in the JLink/Examples/Part1 directory.

One of the things you will notice is that on a reasonably fast machine, the speed is perfectly acceptable. There is nothing to suggest that the entire functionality of the application is scripted from *Mathematica*. It is very responsive despite the fact that a large number of callbacks to *Mathematica* are triggered. For example, the cursor is changed as you float the mouse over various regions of the window (it changes to a resize cursor in some places), so there is a constant flow of callbacks to *Mathematica* as you move the mouse. This example demonstrates the feasibility of writing a sophisticated application entirely in *Mathematica*.

This application was written in *Mathematica*, but it could also have been written entirely in Java, or a combination of Java and *Mathematica*. An advantage of doing it in *Mathematica* is that you generally can be much more productive. Spirograph would have taken at least twice as long to write in Java. It is invaluable to be able to write and test the program a line at a time, and to debug and modify it while it is running. Even if you intend to eventually port the code to pure Java, it can be very useful to begin writing it in *Mathematica*, just to take advantage of the scripting mode of development.

Modal programs like this are best developed using `ShareFrontEnd`, then made modal only when they are complete. Making it modeless while it is being developed is necessary to be able to build and debug it interactively, because while it is running you can continue to use the front end to modify the code, make new definitions, add debugging statements, and so on. Using `ShareFrontEnd` instead of `ShareKernel` for modeless operation lets *Mathematica* error and warning messages generated by event callbacks, and `Print` statement inserted for debugging, show up in the notebook window. Only when everything is working as desired do you add the `DoModal[]` call to turn it into a modal window.

A Piano Keyboard

With the inclusion of the Java Sound API in Java 1.3 and later, it becomes possible to write Java programs that do sophisticated things with sound, such as playing MIDI instruments. The Piano.nb example in the JLink/Examples/Part1 directory displays a keyboard and lets you play it by clicking the mouse. A popup menu at the top lists the available MIDI instruments. This example was created precisely because it is so far outside the limitations of traditional *Mathematica* programming. Using *J/Link*, you can actually write a short and completely portable program, entirely in the *Mathematica* language, that displays a MIDI keyboard and lets you play it! With just a little more work, the code could be modified to record a sequence played and then return it to *Mathematica*, where you could manipulate it by transposing, altering the tempo, and so on.

Advanced Topics in J/Link

Calling Java from Mathematica

Preamble

J/Link provides *Mathematica* users with the ability to interact with arbitrary Java classes directly from *Mathematica*. You can create objects and call methods directly in the *Mathematica* language. You do not need to write any Java code, or prepare in any way the Java classes you want to use. You also do not need to know anything about *MathLink*. In effect, all of Java becomes a transparent extension to *Mathematica*, almost as if every existing and future Java class were written in the *Mathematica* language itself.

This facility is called "installable Java" because it generalizes the ability that *Mathematica* has always had to plug in extensions written in other languages through the `Install` function. You will see later how *J/Link* vastly simplifies this procedure for Java compared to languages like C or C++. In fact, *J/Link* makes the procedure go away completely, which is why Java becomes a transparent extension to *Mathematica*.

Although Java is often referred to as an interpreted language, this is really a misnomer. To use Java you must write a complete program, compile it, and then execute it (some environments exist that let you interactively execute lines of Java code, but these are special tools, and similar tools exist for traditional languages like C). *Mathematica* users have the luxury of working in a true interpreted, interactive environment that lets them experiment with functions and build and test programs a line at a time. *J/Link* brings this same productive environment to Java programmers. You could say that *Mathematica* becomes a scripting language for Java.

To *Mathematica* users, then, the "installable Java" feature of *J/Link* opens up the expanding universe of Java classes as an extension to *Mathematica*; for Java users, it allows the extraordinarily powerful and versatile *Mathematica* environment to be used as a shell for interactively developing, experimenting with, and testing Java classes.

Loading the J/Link Package

The first step is to load the *J/Link* package file.

```
Needs["JLink`"]
```

Launching the Java Runtime

InstallJava

The next step is to launch the Java runtime and "install" it into *Mathematica*. The function for this is `InstallJava`.

`InstallJava[]`	launch the Java runtime and prepare it for use from *Mathematica*
`ReinstallJava[]`	quit and restart the Java runtime if it is already running
`JavaLink[]`	give the `LinkObject` that is being used to communicate with the Java runtime

Launching the Java runtime.

```
In[1]:= InstallJava[]

Out[1]= LinkObject[d:\jdk122\bin\java, 5, 2]
```

`InstallJava` can be called more than once in a session. On every call after the first, it does nothing. Thus, it is safe to call `InstallJava` in any program you write, without considering whether the user has already called it.

`InstallJava` creates a command line that is used to launch the Java runtime (typically called "java") and specify some initial arguments for it. In rare cases you will need to control what is on this command line, so `InstallJava` takes a number of options for this purpose. Most users will not need to use these options, and in fact you should avoid them. Programmers should not assume that they have the ability to control the launch of the Java runtime, as it might already be running. If for some reason you absolutely must apply options to control the launch of the Java runtime, use `ReinstallJava` instead of `InstallJava`.

`ClassPath->None`	use the default class path of your Java runtime
`ClassPath->"dirs"`	use the specified directories and jar files
`CommandLine->"cmd"`	use the specified command line to launch the Java runtime, instead of "java"

Options for `InstallJava`.

Controlling the Command Used to Launch Java

An important option to InstallJava and ReinstallJava is CommandLine. This specifies the first part of the command line used to launch Java. One use for this option is if you have more than one Java runtime installed on your system, and you want to invoke a specific one:

```
ReinstallJava[CommandLine → "d:\\full\\path\\to\\java.exe"]
```

By default, InstallJava will launch the Java runtime that is bundled with *Mathematica* 4.2 and later. If you have an earlier version of *Mathematica,* the default command line that will be used is java on most systems. If the java executable is not on your system path, you can use InstallJava to point at it. Another use for this option is to specify arguments to Java that are not covered by other options. Here is an example that specifies verbose garbage collection and defines a property named foo to have the value bar.

```
ReinstallJava[CommandLine → "/path/to/java -verbosegc -Dfoo=bar"]
```

Overriding the Class Path

The class path is the set of directories in which the Java runtime looks for classes. When you launch a Java program from your system's command line, the class path used by Java includes some default locations and any locations specified in the CLASSPATH environment variable, if it exists. If you use the –classpath command-line option to specify a set of locations, however, then the CLASSPATH environment variable is ignored. The ClassPath option to InstallJava and ReinstallJava works the same way. If you leave it at the default value, Automatic, then *J/Link* will include the contents of the CLASSPATH environment variable in its class search path. If you set it to None or a string, then the contents of CLASSPATH are not used. If you set it to be a string, use the same syntax that you would use for setting the CLASSPATH environment variable, which is different for Windows and Unix:

```
ReinstallJava[ClassPath → "c:\\my\\java\\dir;d:\\MyJavaStuff.jar"] (* Windows *)

ReinstallJava[ClassPath → "/my/java/dir:/home/me/MyJavaStuff.jar"]
(* Unix/Linux *)
```

J/Link has its own mechanism for controlling the class search path that is very flexible. Not only does *J/Link* automatically search for classes in *Mathematica* application directories, it also lets you dynamically add new search locations while the Java runtime is running. This means that using the ClassPath option to configure the class path when Java first launches is not very important. One setting for the ClassPath option that is sometimes useful is None, to prevent

J/Link from finding any classes from the contents of CLASSPATH. You might want to do this if you had an experimental version of some class in a development directory and you wanted to make sure that *J/Link* used that version in preference to an older one that was present on your CLASSPATH. "The Java Class Path" presents a complete treatment of the subject of how *J/Link* searches for classes, and how to add locations to this search path.

Loading Classes

LoadJavaClass

LoadJavaClass["classname"]	load the specified class into Java and *Mathematica*
LoadClass["classname"]	deprecated name from earlier versions of *J/Link*; use LoadJavaClass instead

Loading classes.

To use a Java class in *Mathematica*, it must first be loaded into the Java runtime and certain definitions must be set up in *Mathematica*. This is accomplished with the LoadJavaClass function. LoadJavaClass takes a string specifying the fully qualified name of the class (i.e., the full hierarchical name with all the periods):

```
urlClass = LoadJavaClass["java.net.URL"]
JavaClass[java.net.URL]
```

The return value is an expression with head JavaClass. This JavaClass expression can be used in many places in *J/Link*, so you might want to assign it to a variable as done here. Virtually everywhere in *J/Link* where a class needs to be specified as an argument, you can use either a JavaClass expression, the fully qualified class name as a string, or an object of the class. Note that you cannot create a valid JavaClass expression by simply typing it in—it must be returned by LoadJavaClass.

When a class has been loaded, you can call static methods in the class, create objects of the class, and invoke methods and access fields of these objects. You can use any *public* constructors, methods, or fields of a class.

`StaticsVisible->True`	make static methods and fields accessible by just their names, not in a special context
`AllowShortContext->False`	make static methods and fields accessible only in their fully qualified class context
`UseTypeChecking->False`	suppress the type checking that is normally inserted in definitions for calls into Java

Options for `LoadJavaClass`.

"The Java Class Path" discusses the details of how and where *J/Link* finds classes. *J/Link* will be able to find classes on the class path, in the special Java extensions directory, and in a set of extra directories that users can control even while *J/Link* is running.

When to Call LoadJavaClass

It is often the case that you do not need to explicitly load a class with `LoadJavaClass`. As described later, when you create a Java object with `JavaNew`, you can supply the class name as a string. If the class has not already been loaded, `LoadJavaClass` will be called internally by `JavaNew`. In fact, anytime a Java object is returned to *Mathematica* its class is loaded automatically if necessary. This would seem to imply that there is little reason to use `LoadJavaClass`. There are a number of reasons why you would want or need to use `LoadJavaClass` explicitly:

- You need to call a static method of a class and you will not create, or have not yet created, an object of that class. A class must be loaded before any of its static methods can be called.
- You need to use one of the options to `LoadJavaClass`. When `LoadJavaClass` is called internally by `JavaNew`, it is called with the default option settings.
- You want to see errors associated with loading a class reported at a well-defined time.
- You want to control where your users experience the initial delay associated with loading a class. Loading a class can take several seconds if it or one of its parent classes is very large (although it rarely takes that long). You might want to avoid a mysterious delay in a function that users expect to be very quick.
- You want to hang on to the `JavaClass` expression returned by `LoadJavaClass` to use it in other functions. Although all functions that take a `JavaClass` can also take a class name string, you might prefer to use a named `JavaClass` variable for readability purposes. It is also slightly faster than using a string, but this will not be perceptible unless you are using it many times in a loop.
- You feel that it makes your code more self-documenting.

The operation of loading a class in *J/Link* is only done once in a *J/Link* session (a session is the period between `InstallJava` and `UninstallJava`). You can call `LoadJavaClass` on a given class as many times as you want, and every call after the first one immediately returns the `JavaClass` expression without doing any work. This is important, as it means that you never have to worry whether a class has been loaded already—if you are not sure, call `LoadJavaClass`.

Developers writing code for a wide audience should always call `LoadJavaClass` on any classes they need in every function that needs them. It is not suitable to call `LoadJavaClass` in the body of your package code when it is read in, as the user may quit and restart the Java runtime (i.e., `UninstallJava` and `InstallJava`) after your package was read. To be safe, every user-level function that uses *J/Link* should call `InstallJava` and `LoadJavaClass` (if `LoadJavaClass` is necessary; see the following). Both calls execute very quickly if they are not needed.

As mentioned already, loading a class can take several seconds in some cases. When a class is loaded, all of its superclasses are loaded in succession, walking up the inheritance hierarchy. Because a given class is only actually loaded once, if you load another class that shares some of the same superclasses as a previously loaded class, these superclasses will not have to be loaded again. This means that loading the second class will be much quicker than the first if any of the shared superclasses were large. An example of this is loading classes in the java.awt package. The class `java.awt.Component` is very large, so the first time you load a class that inherits from it, say `java.awt.Button`, there will be a noticeable delay. Subsequent loading of other classes derived from `Component` will be much quicker.

Contexts and Visibility of Static Members

`LoadJavaClass` has two options that let you control the naming and visibility of static methods and fields. To understand these options, you need to understand the problems they help to solve. This explanation gets a bit ahead since how to call Java methods has not been discussed. When a class is loaded, definitions are created in *Mathematica* that allow you to call methods and access fields of objects of that class. Static members are treated quite differently from nonstatic ones. None of these issues arise for nonstatic members, so only static members are discussed in this section. Say you have a class named `com.foobar.MyClass` that contains a static method named `foo`. When you load this class, a definition must be set up for `foo` so that it can be called by name, something like `foo[`*`args`*`]`. The question becomes: In what context do you want the symbol `foo` defined, and do you want this context to be visible (i.e., on `$ContextPath`)?

J/Link always creates a definition for `foo` in a context that mirrors its fully qualified classname: `com`foobar`MyClass`foo`. This is done to avoid conflicting with symbols named `foo` that might be present in other contexts. However, you might find it clumsy to have to call `foo` by typing the full context name every time, as in `com`foobar`MyClass`foo[`*args*`]`. The option `AllowShortContext -> True` (this is the default setting) causes *J/Link* to also make definitions for `foo` accessible in a shortened context, one that consists of just the class name without the hierarchical package name prefix. In the example, this means that you could call `foo` as simply `MyClass`foo[`*args*`]`. If you need to avoid use of the short context because there is already a context of the same name in your *Mathematica* session, you can use `AllowShortContext -> False`. This forces all names to be put only in the "deep" context. Note that even with `AllowShortContext -> True`, names for statics are also put into the deep context, so you can always use the deep context to refer to a symbol if you desire.

`AllowShortContext`, then, lets you control the context where the symbol names are defined. The other option, `StaticsVisible`, controls whether this context is made visible (put on `$ContextPath`) or not. The default is `StaticsVisible -> False`, so you have to use a context name when referring to a symbol, as in `MyClass`foo[`*args*`]`. With `StaticsVisible -> True`, `MyClass`` will be put on `$ContextPath`, so you could just write `foo[`*args*`]`. Having the default be `True` would be a bit dangerous—every time you load a class a potentially large number of names would suddenly be created and made visible in your *Mathematica* session, opening up the possibility for all sorts of "shadowing" problems if symbols of the same names were already present. This problem is particularly acute with Java, because method and field names in Java typically begin with a lowercase letter, which is also the convention for user-defined symbols in *Mathematica*. Some Java classes define static methods and fields with names like `x`, `y`, `width`, and so on, so shadowing errors are very likely to occur (see "Contexts" for a discussion of contexts and shadowing problems).

For these reasons `StaticsVisible -> True` is recommended only for classes that you have written, or ones whose contents you are familiar with. In such cases, it can save you some typing, make your code more readable, and prevent the all-too-easy bug of forgetting to type the package prefix. A classic example would be implementing the venerable "addtwo" *MathLink* example program. In Java, it might look like this:

```
public class AddTwo {
    public static int addtwo(int i, int j) {return i + j;}
}
```

With the default `StaticsVisible -> False`, you would have to call addtwo as `AddTwo`addtwo[3, 4]`. Setting `StaticsVisible -> True` lets you write the more obvious `addtwo[3, 4]`.

Be reminded that these options are only for *static* methods and fields. As discussed later, non-statics are handled in a way that makes context and visibility issues go away completely.

Inner Classes

Inner classes are public classes defined inside another public class. For example, the class `javax.swing.Box` has an inner class named `Filler`. When you refer to the `Filler` class in a Java program, you typically use the outer class name, followed by a period, then the inner class name:

```
Box.Filler f = new Box.Filler(…);
```

You can use inner classes with *J/Link*, but you need to use the true internal name of the class, which has a `$`, not a period, separating the outer and inner class names:

```
filler = JavaNew["java.swing.Box$Filler", …]
```

If you look at the class files produced by the Java compiler, you will see these `$`-separated class names for inner classes.

Conversion of Types Between Java and Mathematica

Before you encounter the operations of creating Java objects and calling methods, you should examine the mapping of types between *Mathematica* and Java. When a Java method returns a result to *Mathematica*, the result is automatically converted into a *Mathematica* expression. For example, Java integer types (e.g., byte, short, int, and so on), are converted into *Mathematica* integers, and Java real number types (float, double) are converted into *Mathematica* reals. The following table shows the complete set of conversions. These conversions work both ways—for example, when a *Mathematica* integer is sent to a Java method that requires a byte value, the integer is automatically converted to a Java byte.

Java type	*Mathematica type*
`byte, char, short, int, long`	`Integer`
`Byte, Character, Short, Integer, Long, BigInteger`	`Integer`
`float, double`	`Real`
`Float, Double, BigDecimal`	`Real`
`boolean`	`True or False`
`String`	`String`
`array`	`List`
controlled by user (see "Complex Numbers")	`Complex`
`Object`	`JavaObject`
`Expr`	any expression
`null`	`Null`

Corresponding types in Java and *Mathematica*.

Java arrays are mapped to *Mathematica* lists of the appropriate depth. Thus, when you call a method that takes a `double[]`, you might pass it `{1.0, 2.0, N[Pi], 1.23}`. Similarly, a method that returns a two-deep array of integers (i.e., `int[][]`) might return to *Mathematica* the expression `{{1, 2, 3}, {5, 3, 1}}`.

In most cases, *J/Link* will let you supply a *Mathematica* integer to a method that is typed to take a real type (`float` or `double`). Similarly, a method that takes a `double[]` could be passed a list of mixed integers and reals. The only times when you cannot do this are the rare cases where a method has two signatures that differ only in a real versus integer type at the same argument slot. For example, consider a class with these methods:

```
public void foo(byte b, Object obj);
public void foo(float f, Object obj);
public void bar(float f, Object obj);
```

J/Link would create two *Mathematica* definitions for the method `foo`—one that required an integer for the first argument and invoked the first signature, and one that required a real number for the first argument and invoked the second signature. The definition created for the method `bar` would accept an integer or a real for the first argument. In other words, *J/Link* will automatically convert integers to reals, except in cases where such conversion makes it ambigu-

ous as to which signature of a given method to invoke. This is not strictly true, though, as *J/Link* does not try as hard as it possibly could to determine whether real versus integer ambiguity is a problem at every argument position. The presence of ambiguity at one position will cause *J/Link* to give up and require exact type matching at all argument positions. This is starting to sound confusing, but you will find that in most cases *J/Link* allows you to pass integers or lists with integers to methods that take reals or arrays of reals, respectively, as arguments. In cases where it does not, the call will fail with an error message, and you will have to use *Mathematica*'s `N` function to convert all integers to reals explicitly.

Creating Objects

To instantiate Java objects, use the `JavaNew` function. The first argument to `JavaNew` is the object's class, specified either as a `JavaClass` expression returned from `LoadJavaClass` or as a string giving the fully qualified class name (i.e., having the full package prefix with all the periods). If you wish to supply any arguments to the object's constructor, they follow as a sequence after the class.

`JavaNew[`*cls*`,`*arg1*`,…]`	construct a new object of the specified class and return it to *Mathematica*
`JavaNew["`*classname*`",`*arg1*`,…]`	construct a new object of the specified class and return it to *Mathematica*

Constructing Java objects.

For example, this will create a new `Frame`.

```
frm = JavaNew["java.awt.Frame"]
```

```
«JavaObject[java.awt.Frame] »
```

The return value from `JavaNew` is a strange expression that looks like it has the head `JavaObject`, except that it is enclosed in angle brackets. The angle brackets are used to indicate that the form in which the expression is displayed is quite different from its internal representation. These expressions will be referred to as `JavaObject` expressions. `JavaObject` expressions are displayed in a way that shows their class name, but you should consider them opaque, meaning that you cannot pick them apart or peer into their insides. You can only use them in *J/Link* functions that take `JavaObject` expressions. For example, if *obj* is a `JavaObject`, you cannot use `First[`*obj*`]` to get its class name. Instead, there is a *J/Link* function, `ClassName[`*obj*`]`, for this purpose.

JavaNew invokes a Java constructor appropriate for the types of the arguments being passed in, and then returns to *Mathematica* what is, in effect, a reference to the object. That is how you should think of JavaObject expressions—as references to Java objects very much like object references in the Java language itself. What is returned to *Mathematica* is not large no matter what type of object you are constructing. In particular, the object's data (that is, its fields) are not sent back to *Mathematica*. The actual object remains on the Java side, and *Mathematica* gets a reference to it.

The Frame class has a second constructor, which takes a title in the form of a string. Here is how you would call that constructor.

```
frm = JavaNew["java.awt.Frame", "My Example Frame"]
```

```
«JavaObject[java.awt.Frame] »
```

Note that simply constructing a Frame does not cause it to appear. That requires a separate step (calling the frame's show or setVisible methods will work, but as you will see later, *J/Link* provides a special function, JavaShow, to make Java windows appear and come to the foreground).

The previous examples specified the class by giving its name as a string. You can also use a JavaClass expression, which is a special expression returned by LoadJavaClass that identifies a class in a particularly efficient manner. When you specify the class name as a string, the class is loaded if it has not already been.

```
frameClass = LoadJavaClass["java.awt.Frame"];
frm = JavaNew[frameClass, "My Example Frame"];
```

JavaNew is not the only way to get a reference to a Java object in *Mathematica*. Many methods and fields return objects, and when you call such a method, a JavaObject expression is created. Such objects can be used in the same way as ones you explicitly construct with JavaNew.

At this point, you may be wondering about things like reference counts and how objects returned to *Mathematica* get cleaned up. These issues are discussed in "Object References in *Mathematica*".

J/Link has two other functions for creating Java objects, called MakeJavaObject and MakeJavaExpr. These specialized functions are described in the section "MakeJavaObject and MakeJavaExpr".

Calling Methods and Accessing Fields

Syntax

The *Mathematica* syntax for calling Java methods and accessing fields is very similar to Java syntax. The following box compares the *Mathematica* and Java ways of calling constructors, methods, fields, static methods, and static fields. You can see that *Mathematica* programs that use Java are written in almost exactly the same way as Java programs, except *Mathematica* uses `[]` instead of `()` for arguments, and *Mathematica* uses `@` instead of Java's `.` (dot) as the "member access" operator.

An exception is that for static methods, *Mathematica* uses the context mark `` ` `` in place of Java's dot. This parallels Java usage also, as Java's use of the dot in this circumstance is really as a scope resolution operator (like `::` in C++). Although *Mathematica* does not use this terminology, its scope resolution operator is the context mark. Java's hierarchical package names map directly to *Mathematica*'s hierarchical contexts.

	constructors
Java:	`MyClass obj=new MyClass (args);`
Mathematica:	`obj=JavaNew["MyClass",args];`
	methods
Java:	`obj.methodName (args);`
Mathematica:	`obj@methodName[args]`
	fields
Java:	`obj.fieldName=1;` `value=obj.fieldName;`
Mathematica:	`obj@fieldName=1;` `value=obj@fieldName;`
	static methods
Java:	`MyClass.staticMethod (args);`
Mathematica:	`` MyClass`staticMethod[args]; ``
	static fields
Java:	`MyClass.staticField=1;` `value=MyClass.staticField;`
Mathematica:	`` MyClass`staticField=1;`` `` value=MyClass`staticField; ``

Java and *Mathematica* syntax comparison.

You may already be familiar with @ as a *Mathematica* operator for applying a function to an argument: f@x is equivalent to the more commonly used f[x]. *J/Link* does not usurp @ for some special operation—it is really just normal function application slightly disguised. This means that you do not have to use @ at all. The following are equivalent ways of invoking a method:

```
(* These are equivalent *)
obj@method[args];
obj[method[args]];
```

The first form preserves the natural mapping of Java's syntax to *Mathematica*'s, and it will be used exclusively in this tutorial.

When you call methods or fields and get results back, *J/Link* automatically converts arguments and results to and from their *Mathematica* representations according to the table in "Conversion of Types between Java and *Mathematica*".

Method calls can be chained in *Mathematica* just like in Java. For example, if meth1 returns a Java object, you could write in Java obj.meth1().meth2(). In *Mathematica*, this becomes obj@meth1[]@meth2[]. Note that there is an apparent problem here: *Mathematica*'s @ operator groups to the right, whereas Java's dot groups to the left. In other words, obj.meth1().meth2() in Java is really (obj.meth1()).meth2() whereas obj@meth1[]@meth2[] in *Mathematica* would normally be obj@(meth1[]@meth2[]). I say "normally" because *J/Link* automatically causes chained calls to group to the left like Java. It does this by defining rules for JavaObject expressions, not by altering the properties of the @ operator, so the global behavior of @ is not affected. This chaining behavior only applies to method calls, not fields. You cannot do this:

```
(* These are incorrect. You cannot chain calls after a field access. *)
x = obj@field@method[args];
x = obj@field1@field2;
```

You would have to split these up into two lines. For example, the second line above would become:

```
temp = obj@field1;
x = temp@field2;
```

In Java, like other object-oriented languages, method and field names are scoped by the object on which they are called. In other words, when you write obj.meth(), Java knows that you are calling the method named meth that resides in obj's class, even though there may be other methods named meth in other classes. *J/Link* preserves this scoping for *Mathematica* symbols

so that there is never a conflict with existing symbols of the same name. When you write obj@meth[], there is no conflict with any other symbols named meth in the system—the symbol meth used by *Mathematica* in the evaluation of this call is the one set up by *J/Link* for this class. Here is an example using a field. First, you create a Point object.

```
pt = JavaNew["java.awt.Point"]
```

```
«JavaObject[java.awt.Point] »
```

The Point class has fields named x and y, which hold its coordinates. A user's session is also likely to have symbols named x or y in it, however. You set up a definition for x that will tell you when it is evaluated.

```
x := Print["gotcha"]
```

Now set a value for the field named x (this would be written as pt.x = 42 in Java).

```
pt@x = 42;
```

You will notice that "gotcha" was not printed. There is no conflict between the symbol x in the Global` context that has the Print definition and the symbol x that is used during the evaluation of this line of code. *J/Link* protects the names of methods and fields on the right-hand side of @ so that they do not conflict with, or rely on, any definitions that might exist for these symbols in visible contexts. Here is a method example that demonstrates this issue differently.

```
frm = JavaNew["java.awt.Frame"];
frm@show[]
```

Even though a new symbol show is being created here, the show that is used by *J/Link* is the one that resides down in the java`awt`Frame context, which has the necessary definitions set up for it.

In summary, for nonstatic methods and fields, you never have to worry about name conflicts and shadowing, no matter what context you are in or what the $ContextPath is at the moment. This is not true for static members, however. Static methods and fields are called by their full name, without an object reference, so there is no object out front to scope the name. Here is a simple example of a static method call that invokes the Java garbage collector. You need to call LoadJavaClass before you call a static method to make sure the class has been loaded.

```
LoadJavaClass["java.lang.Runtime"];
Runtime`gc[];
```

The name scoping issue is not usually a problem with statics, because they are defined in their own contexts (`Runtime`` in this example). These contexts are usually not on `$ContextPath`, so you do not have to worry that there is a symbol of the same name in the `Global`` context or in a package that has been read. There is more discussion of this issue in the section on `LoadJavaClass`, because `LoadJavaClass` takes options that determine the contexts in which static methods are defined and whether or not they are put on `$ContextPath`. If there is already a context named `Runtime`` in your session, and it has its own symbol `gc`, you can always avoid a conflict by using the fully hierarchical context name that corresponds to the full class name for a static member.

```
java`lang`Runtime`gc[];
```

Finally, just as in Java, you can call a static method on an object if you like. In this case, since there is an object out front, you get the name scoping. Here you call a static method of the `Runtime` class that returns the current `Runtime` object (you cannot create a `Runtime` object with `JavaNew`, as `Runtime` has no constructors). You then invoke the (static) method `gc` on the object, and you can use `gc` without any context prefix.

```
runtime = Runtime`getRuntime[];
runtime@gc[];
```

Underscores in Java Names

Java names can have characters in them that are not legal in *Mathematica* symbols. The only common one is the underscore. *J/Link* maps underscores in class, method, and field names to "U". Note that this mapping is only used where it is necessary—when names are used in symbolic form, not as strings. For example, assume you have a class named `com.acme.My_Class`. When you refer to this class name as a string, you use the underscore.

```
LoadJavaClass["com.acme.My_Class"];
JavaNew["com.acme.My_Class"];
```

But when you call a static method in such a class, the hierarchical context name is symbolic, so you must convert the underscore to `U`.

```
com`acme`MyUClass`staticMethod[];
MyUClass`staticMethod[];
```

The same rule applies to method and field names. Many Java field names have underscores in them, for example `java.awt.Frame.TOP_ALIGNMENT`. To refer to this method in code, use the `U`.

```
LoadJavaClass["java.awt.Frame"];
Frame`TOPUALIGNMENT
0.
```

In cases where you supply a string, leave the underscore.

```
Fields["java.awt.Frame", "*_ALIGNMENT"]
static final float BOTTOM_ALIGNMENT
static final float CENTER_ALIGNMENT
static final float LEFT_ALIGNMENT
static final float RIGHT_ALIGNMENT
static final float TOP_ALIGNMENT
```

Getting Information about Classes and Objects

J/Link has some useful functions that show you the constructors, methods, and fields available for a given class or object.

`Constructors[`*cls*`]`	return a table of the public constructors and their arguments
`Constructors[`*obj*`]`	constructors for this object's class
`Methods[`*cls*`]`	return a table of the public methods and their arguments
`Methods[`*cls*`,"`*pat*`"]`	show only methods whose names match the string pattern *pat*
`Methods[`*obj*`]`	show methods for this object's class
`Fields[`*cls*`]`	return a table of the public fields
`Fields[`*cls*`,"`*pat*`"]`	show only fields whose names match the string pattern *pat*
`Fields[`*obj*`]`	show fields for this object's class
`ClassName[`*cls*`]`	return, as a string, the name of the class represented by *cls*
`ClassName[`*obj*`]`	return, as a string, the name of this object's class
`GetClass[`*obj*`]`	return the `JavaClass` representing this object's class
`ParentClass[`*obj*`]`	return the `JavaClass` representing this object's parent class
`InstanceOf[`*obj*`,`*cls*`]`	return `True` if this object is an instance of *cls*, `False` otherwise
`JavaObjectQ[`*expr*`]`	return `True` if *expr* is a valid reference to a Java object, `False` otherwise

Getting information about classes and objects.

You can give an object or a class to Constructors, Methods, and Fields. The class can be specified either by its full name as a string, or as a JavaClass expression:

```
urlClass = LoadJavaClass["java.net.URL"];
urlObject = JavaNew["java.net.URL", "http://www.wolfram.com"];
(* The next three lines are equivalent *)
Methods[urlClass]
Methods[urlObject]
Methods["java.net.URL"]
```

The declarations returned by these functions have been simplified by removing the Java keywords public, final (removed only for methods, not fields), synchronized, native, volatile, and transient. The declarations will always be public, and the other modifiers are probably not relevant for use via *J/Link*.

Methods and Fields take one option, Inherited, which specifies whether to include members inherited from superclasses and interfaces or show only members declared in the class itself. The default is Inherited -> True.

Inherited->False	show only members that are declared in the class itself, not inherited from superclasses or interfaces

Option for Methods and Fields.

There are additional functions that give information about objects and classes. These functions are ClassName, GetClass, ParentClass, InstanceOf, and JavaObjectQ. They are self-explanatory, for the most part. The InstanceOf function mimics the Java language's instanceof operator. JavaObjectQ is useful for writing patterns that match only valid Java objects:

```
Stringify[obj_?JavaObjectQ] := obj[toString[]]
```

JavaObjectQ returns True if and only if its argument is a valid reference to a Java object or if it is the symbol Null, which maps to Java's null object.

Quitting or Restarting Java

When you are finished with using Java in a *Mathematica* session, you can quit the Java runtime by calling UninstallJava[].

UninstallJava[]	quit the Java runtime
ReinstallJava[]	restart the Java runtime

Quitting the Java runtime.

In addition to quitting Java, `UninstallJava` clears out the many symbols and definitions created in *Mathematica* when you load classes. All outstanding `JavaObject` expressions will become invalid when Java is quit. They will no longer satisfy `JavaObjectQ`, and they will show up as raw symbols like `JLink`Objects`JavaObject12345678` instead of `<< JavaObject[classname] >>`.

Most users will have no reason to call `UninstallJava`. You should think of the Java runtime as an integral part of the *Mathematica* system—start it up, and then just leave it running. All code that uses *J/Link* shares the same Java runtime, and there may be packages that you are using that make use of Java without you even knowing it. Shutting down Java might compromise their functionality. Developers writing packages should *never* call `UninstallJava` in their packages. You cannot assume that when your application is done with *J/Link*, your users are done with it as well.

About the only common reason to need to stop and restart Java is when you are actively developing Java classes that you want to call from *Mathematica*. Once a class is loaded into the Java runtime, it cannot be unloaded. If you want to modify and recompile your class, you need to restart Java to reload the modified version. Even in this circumstance, though, you will not be calling `UninstallJava`. Instead, you will call `ReinstallJava`, which simply calls `UninstallJava` followed by `InstallJava` again.

Version Information

J/Link provides three symbols that supply version information. These symbols provide the same type of information as their counterparts in *Mathematica* itself, except that they are in the `JLink`Information`` context, which is not on `$ContextPath`, so you must specify them by their full names.

`JLink`Information`$Version`	a string giving full version information
`JLink`Information`$VersionNumber`	a real number giving the current version number
`JLink`Information`$ReleaseNumber`	an integer giving the release number (the last digit in a full x.x.x version specification)
`ShowJavaConsole[]`	the console window will show version information for the Java runtime and the *J/Link* Java component

J/Link version information.

```
JLink`Information`$Version
```

```
J/Link Version 4.0.1
```

```
JLink`Information`$VersionNumber
```

```
4.
```

```
JLink`Information`$ReleaseNumber
```

```
1
```

The `ShowJavaConsole[]` function, described in "The Java Console Window", will also display some useful version information. It shows the version of the Java runtime being used and the version of the portion of *J/Link* that is written in Java. The version of the *J/Link* Java component should match the version of the *J/Link Mathematica* component.

Controlling the Class Path: How J/Link Finds Classes

The Java Class Path

The class path tells the Java runtime, compiler, and other tools where to find third-party and user-defined classes—classes that are not Java "extensions" or part of the Java platform itself. The class path has always been a source of confusion among Java users and programmers.

Java can find classes that are part of the standard Java platform (so-called "bootstrap" classes), classes that use the so-called "extensions" mechanism, and classes on the class path, which is controlled by the `CLASSPATH` environment variable or by command-line options when Java is launched. *J/Link* can load and use any classes that the Java runtime can find through these normal mechanisms. In addition, *J/Link* can find classes, resources, and native libraries that are in a set of extra locations, beyond what is specified on the class path at startup. This set of extra locations can be added to while Java is running.

J/Link provides two ways to alter the search path Java uses to find classes. The first way is via the `ClassPath` option to `ReinstallJava`. The second way, which is superior to modifying the class path at startup, is to add new directories and jar files to the special set of extra locations that *J/Link* searches. These two methods will be described in the next two subsections.

Overriding the Startup Class Path

For a class to be accessible via the standard Java class path, one of the following must apply:

- It is inside a .zip or .jar file that is itself named on the class path.
- It is a loose class file that is in an appropriately nested directory beneath a directory that is on the class path.

"Appropriately nested" means that the class file must be in a directory whose hierarchy mirrors the full package name of the class. For example, assume that the directory c:\MyClasses is on the class path. If you have a class that is not in a package (there is no package statement at the beginning of the code), its class file should be put directly into c:\MyClasses. If you have a class that is in the package `com.acme.stuff`, its class file would need to be in the directory c:\MyClasses\com\acme\stuff. Note that jar and zip files must be explicitly named on the class path—you cannot just toss them into a directory that is itself named on the class path. Directory issues are not relevant for jar and zip files, meaning that regardless of how hierarchically organized the classes inside a jar file are, you simply name the jar file itself on the class path and all the classes inside it can be found.

If you want to specify paths for classes that are not part of the standard Java platform or extensions, you can use the `ClassPath` option to `ReinstallJava`. The value that you supply for the `ClassPath` option is a string that names the desired directories and zip or jar files. This string is platform-dependent; the paths are specified in the native style for your platform, and the separator character is a colon on Unix and a semicolon on Windows. Here are typical specifications:

```
ReinstallJava[ClassPath → "c:\\MyJavaDir\\MyPackage.jar;c:\\MyJavaDir"]
(* Windows *)

ReinstallJava[ClassPath → "~/MyJavaDir/MyPackage.jar:~/MyJavaDir"]
(* Unix *)
```

The default setting for `ClassPath` is `Automatic`, which means to use the value of the `CLASSPATH` environment variable. If you set `ClassPath` to something else, then J/Link will ignore the `CLASSPATH` environment variable—it will not be able to find those classes. In other words, if you use a `ClassPath` specification, you lose the `CLASSPATH` environment variable. This is similar to the behavior of the `-classpath` command-line option to the Java runtime and compiler, if you are familiar with those tools.

It is recommended that users avoid the ClassPath option. If you need the dynamic control that the ClassPath option provides, you should use the more powerful and convenient AddToClassPath mechanism, described in the next section. The most common reason for using the ClassPath option is if you want to specifically prevent the contents of the CLASSPATH environment variable from being used. To do this, set ClassPath -> None.

Dynamically Modifying the Class Path

One thing that is inconvenient about the standard Java class path is that it cannot be changed after the Java runtime has been launched. *J/Link* has its own class loader that searches in a set of special locations beyond the standard Java class path. This gives *J/Link* an extremely powerful and flexible means of finding classes. To add locations to this extra set, use the AddToClassPath function.

AddToClassPath["*location*",...]	add the specified directories or jar files to *J/Link*'s class search path

Adding classes to the search path.

After Java has been started, you can call AddToClassPath whenever you wish, and it will take effect immediately. One convenient feature of this extra class search path is that if you add a directory, then any jar or zip files in that directory will be searched. This means that you do not have to name jar files individually, as you need to do with the standard Java class path. For loose class files, the nesting rules are the same as for the class path, meaning that if a class is in the package com.acme.stuff, and you called AddToClassPath["d:\\myClasses"], then you would need to put the class file into d:\MyClasses\com\acme\stuff.

Changes to the search path that you make with AddToClassPath only apply to the current Java session. If you quit and restart java, you will need to call AddToClassPath again.

In addition to the locations you add yourself with AddToClassPath, *J/Link* automatically includes any Java subdirectories of any directories in the standard *Mathematica* application locations ($UserBaseDirectory/AddOns/Applications, $BaseDirectory/AddOns/Applications, < *Mathematica dir* >/AddOns/Applications, and < *Mathematica dir* >/AddOns/ExtraPackages). This feature is designed to provide extremely easy deployment for developers who create applications for *Mathematica* that use Java and *J/Link* for part of their implementation. This is described in "Deploying Applications that use *J/Link*" in more detail, but even casual Java pro-

grammers who are writing classes to use with *J/Link* can take advantage of it. Just create a subdirectory of AddOns/Applications, say MyStuff, create a Java subdirectory within it, and toss class or jar files into it. *J/Link* will be able to find and use them. Of course, loose class files have to be placed into an appropriately nested subdirectory of the Java directory, corresponding to their package names (if any), as described.

The `AddToClassPath` function was introduced in *J/Link* 2.0. Previous versions of *J/Link* had a variable called `$ExtraClassPath` that specified a list of extra locations. You could add to this list like this:

```
AppendTo[$ExtraClassPath, "d:\\MyClasses"];
```

`$ExtraClassPath` was deprecated in *J/Link* 2.0, but it still works. One advantage of `$ExtraClassPath` over using `AddToClassPath` is that changes made to `$ExtraClassPath` persist across a restart of the Java runtime.

Examining the Class Path

The `JavaClassPath` function returns the set of directories and jar files in which *J/Link* will search for classes. This includes all locations added with `AddToClassPath` or `$ExtraClassPath`, as well as Java subdirectories of application directories in any of the standard *Mathematica* application locations. It does not display the jar files that make up the standard Java platform itself, or jar files in the Java extensions directory. Those classes can always be found by Java programs.

`JavaClassPath[]`	gives the complete set of directories and jar files in which *J/Link* will search for classes

Inspecting the class search path.

Using *J/Link*'s Class Loader Directly

As stated earlier, *J/Link* uses its own class loader to allow it to find classes and other resources in a dynamic set of locations beyond the startup class path. Essentially all the classes that you load using *J/Link* that are not part of the Java platform itself will be loaded by this class loader. One consequence of this is that calling Java's `Class.forName()` method from *Mathematica* will often not work.

```
LoadJavaClass["java.lang.Class"];
cls = Class`forName["some.class.that.only.JLink.can.find"]
```

```
Java::excptn : A Java exception occurred: java.lang.ClassNotFoundException:
    some.class.that.only.JLink.can.find
      at java.net.URLClassLoader$1.run(Unknown Source)
      at java.security.AccessController.doPrivileged(Native Method)
      at java.net.URLClassLoader.findClass(Unknown Source)
      at java.lang.ClassLoader.loadClass(Unknown Source)
      at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
      at java.lang.ClassLoader.loadClass(Unknown Source)
      at java.lang.ClassLoader.loadClassInternal(Unknown Source)
      at java.lang.Class.forName0(Native Method)
      at java.lang.Class.forName(Unknown Source)
      at
    sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
      at
    sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source).
$Failed
```

The problem is that `Class.forName()` finds classes using a default class loader, not the *J/Link* class loader, and this default class loader does not know about the special directories in which *J/Link* looks for classes (in fact, it does not even know about the startup class path, because of details of how *J/Link* launches Java). If you are translating Java code into *Mathematica*, or if you just want to get a `Class` object for a given class, watch out for this problem. The fix is to force *J/Link*'s class loader to be used. One way to do this is to use the three-argument form of `Class.forName()`, which allows you to specify the class loader to be used:

```
LoadJavaClass["com.wolfram.jlink.JLinkClassLoader"];
cls = Class`forName["some.class.that.only.JLink.can.find",
  True, JLinkClassLoader`getInstance[]]
```

An easier way is to use the static `classFromName` method of `JLinkClassLoader`:

```
cls = JLinkClassLoader`classFromName["some.class.that.only.JLink.can.find"]
```

You should think of this `classFromName()` method as being the replacement for `Class.forName()`. When you find yourself wanting to obtain a `Class` object from a class name given as a string, remember to use `JLinkClassLoader.classFromName()`.

`Class.forName()` is not very commonly found in Java code. One reason it is used is when an object needs to be created, but its class was not known at compile time. For example, the class name might come from a preferences file or be determined programmatically in some other way. Often, the very next line creates an instance of the class, like this:

```
// Java code
Class cls = Class.forName("SomeClassThatImplementsInterfaceX");
X obj = (X) cls.newInstance();
```

If you are translating code like this into a *Mathematica* program, this operation can be performed simply by calling `JavaNew`:

```
obj = JavaNew["SomeClassThatImplementsInterfaceX"]
```

The point here is that for a very common usage of `Class.forName()`, you do not have to translate it line-by-line into *Mathematica*—you can duplicate the functionality by calling `JavaNew`.

Performance Issues

Overhead of Calls to Java

The speed of Java programs is highly dependent on the Java runtime. On certain types of programs, for example, ones that spend most of their time in a tight number-crunching loop, the speed of Java can approach that of compiled, optimized C.

Java is a good choice for computationally intensive programs. Your mileage may vary, but do not rule out Java for any type of program before you have done some simple speed testing. For less demanding programs, where every ounce of speed is not necessary, the simplicity of using *J/Link* instead of programming traditional *MathLink* "installable" programs with C makes Java an obvious choice.

The speed issues with *J/Link* are not, for the most part, the speed of Java execution. Rather, the bottleneck is the rate at which you can perform calls into Java, which is itself limited mainly by the speed of *MathLink* and the processing that must be done in *Mathematica* for each call into Java. The maximum rate of calls into Java is highly dependent on which operating system and which Java runtime you use. A fast Windows machine can perform more than 5000 Java method calls per second, and considerably more if they are static methods, which require less preprocessing in *Mathematica*. On some operating systems the results will be less. You should keep in mind that there is a more or less fixed cost of a call into Java regardless of what the call

does, and on slow machines this cost could be as much as .001 seconds. Many Java methods will execute in considerably less time than this, so the total time for the call is often dominated by the fixed turnaround time of a *J/Link* call, not the speed of Java itself.

For most uses, the overhead of a call into Java is not a concern, but if you have a loop that calls into Java 500,000 times, you will have a problem (unless your program takes so long that the *J/Link* cost is negligible, in which case you have an even bigger problem!). If your *Mathematica* program is structured in a way that requires a great many calls into Java, you may need to refactor it to do more on the Java side and thus reduce the number of times you need to cross the Java-*Mathematica* boundary. This will probably involve writing some Java code, which unfortunately defeats the *J/Link* premise of being able to use *Mathematica* to script the functionality of an arbitrary Java program. There are uses of Java that just cannot be feasibly scripted in this way, and for these you will need to write more of the functionality in Java and less in *Mathematica*.

Speeding Up Sending Large Arrays

You can send and receive arrays of most "primitive" Java types (e.g., `byte`, `short`, `int`, `float`, `double`) nearly as fast as in a C-language program. The set of types that can be passed quickly corresponds to the set of types for which the *MathLink* C API has single functions to put arrays. The Java types `long` (these are 64 bits), `boolean`, and `String` do not have fast *MathLink* functions, and so sending or receiving these types is much slower. Try to avoid using extremely large arrays of these types (say, more than 100,000 elements) if possible.

A setting that has a big effect on the speed of moving multidimensional arrays is the one used to control whether "ragged" arrays are allowed. As discussed in "Ragged Arrays", the default behavior of *J/Link* is to require that all arrays be fully rectangular. But Java does not require that arrays conform to this restriction, and if you want to send or receive ragged arrays, you can call `AllowRaggedArrays[True]` in your *Mathematica* session. This causes *J/Link* to switch to a much slower method for reading and writing arrays. Avoid using this setting unless you need it, and switch it off as soon as you no longer require it.

When you load a class with a method that takes, say, an `int[][]`, the definition in *Mathematica* that *J/Link* creates for calling this method uses a pattern test that requires its argument to be a two-dimensional array of integers. If the array is quite large, say on the order of 500 by 500, this test can take a significant amount of time, probably similar to the time it takes to

actually transfer the array to Java. If you want to avoid the time taken by this testing of array arguments, you can set the variable $RelaxedTypeChecking to True. If you do this, you are on your own to ensure that the arrays you send are of the right type and dimensionality. If you pass a bad array, you will get a *MathLink* error, but this will not cause any problems for *J/Link* (other than that the call will return $Failed).

You probably do not want to leave $RelaxedTypeChecking set to True for a long time, and if you are writing code for others to use you certainly do not want to alter its value in their session. $RelaxedTypeChecking is intended to be used in a Block construct, where it is given the value of True for a short period:

```
Block[{$RelaxedTypeChecking = True}, obj[meth[someLargeArray]]]
```

$RelaxedTypeChecking only has an effect for arrays, which are the only types for which the pattern test that *J/Link* creates is expensive relative to the actual call into Java.

Another optimization to speed up *J/Link* programs is to use ReturnAsJavaObject to avoid unnecessary passing of large arrays or strings back and forth between *Mathematica* and Java. ReturnAsJavaObject is discussed in the section "ReturnAsJavaObject".

An Optimization Example

Next examine a simple example of steps you might take to improve the speed of a *J/Link* program. Java has a powerful DecimalFormat class you can use to format *Mathematica* numbers in a desired way for output to a file. Here you create a DecimalFormat object that will format numbers to exactly four decimal places.

```
fmt = JavaNew["java.text.DecimalFormat", "#.0000"];
```

To use the fmt object, you call its format() method, supplying the number you want formatted.

```
fmt@format[12.34]
12.3400
```

This returns a string with the requested format. Now suppose you want to use this ability to format a list of 20000 numbers before writing them to a file.

```
data = Table[Random[], {40 000}];

Map[fmt@format[#] &, data];
```

The Map call, which invokes the format method 40000 times, takes 46 seconds on a certain PC (this is wall clock time, not the result of the Timing function, which is not accurate for *MathLink* programs on most systems). Clearly this is not acceptable. As a first step, you try using MethodFunction because you are calling the same method many times.

```
methodFunc = MethodFunction[fmt, format];
```

Note that you use fmt as the first argument to MethodFunction. The first argument merely specifies the class; as with virtually all functions in *J/Link* that take a class specification, you can use an object of the class if you desire. The MethodFunction that is created can be used on any object of the DecimalFormat class, not just the fmt object.

```
Map[methodFunc[fmt, #] &, data];
```

Using methodFunc, this now takes 36 seconds. There is a slight speed improvement, much less than in earlier versions of *J/Link*. This means you are getting about 1100 calls per second, and it is still not fast enough to be useful. The only thing to do is to write your own Java method that takes an *array* of numbers, formats them all, and returns an array of strings. This will reduce the number of calls from *Mathematica* into Java 40000 down to one.

Here is the code for the trivial Java class necessary. Note that there is nothing about this code that suggests it will be called from *Mathematica* via *J/Link*. This is exactly the same code you would write if you wanted to use this functionality within Java.

```
public class FormatArray {
    public static String[] format(java.text.DecimalFormat fmt,double[] d) {
        String[] result=new String[d.length];
        for (int i = 0; i < d.length; i++)
            result[i] = fmt.format(d[i]);
        return result;
    }
}
```

This new version takes less than 2 seconds.

```
LoadJavaClass["FormatArray"];
FormatArray`format[fmt, data];
```

Reference Counts and Memory Management

Object References in *Mathematica*

The earlier treatment of JavaObject expressions avoided discussing deeper issues such as reference counts and uniqueness. Every time a Java object reference is returned to *Mathematica*, either as a result of a method or field or an explicit call to JavaNew, *J/Link* looks to see if a reference to this object has been sent previously in this session. If not, it creates a JavaObject expression in *Mathematica* and sets up a number of definitions for it. This is a comparatively time-consuming process. If this object has already been sent to *Mathematica*, in most cases *J/Link* simply creates a JavaObject expression that is identical to the one created previously. This is a much faster operation.

There are some exceptions to this last rule, meaning that sometimes when an object is returned to *Mathematica* a new and different JavaObject expression is created for it, even though this same object has previously been sent to *Mathematica*. Specifically, any time an object's hashCode() value has changed since the last time it was seen in *Mathematica*, the JavaObject expression created will be different. You do not really need to be concerned with the details of this, except to remember that SameQ is not a valid way to compare JavaObject expressions to decide whether they refer to the same object. You must use the SameObjectQ function.

SameObjectQ[*obj1*,*obj2*]	return True if the JavaObject expressions *obj1* and *obj2* refer to the same Java object, False otherwise

Comparing JavaObject expressions.

Here is an example.

```
pt = JavaNew["java.awt.Point", 1, 1]
```

```
«JavaObject[java.awt.Point]»
```

The variable pt refers to a Java Point object. Now put it into a container so you can get it back out later.

```
vec = JavaNew["java.util.Vector"];
vec@add[pt];
```

Now change the value of one of its fields. For a Point object, changing the value of one of its fields changes its hashCode() value.

```
pt@x = 2;
```

Now you compare the JavaObject expression given by pt and the JavaObject expression created when you ask for the first element of the Vector to be returned to *Mathematica*. Even though these are both references to the same Java object, the JavaObject expressions are different.

```
pt === vec@elementAt[0]
False
```

Because you cannot use SameQ (===) to decide whether two object references in *Mathematica* refer to the same Java object, *J/Link* provides a function, SameObjectQ, for this purpose.

```
SameObjectQ[pt, vec@elementAt[0]]
True
```

You may be wondering why the SameObjectQ function is necessary. Why not just call an object's equals() method? It certainly gives the correct result for this example.

```
pt@equals[vec@elementAt[0]]
True
```

The problem with this technique is that equals() does not always compare object references. Any class is free to override equals() to provide any desired behavior for comparing two objects of that class. Some classes make equals() compare the "contents" of the objects, such as the String class, which uses it for string comparison. Java provides two distinct equality operations, the == operator and the equals() method. The == operator always compares references, returning true if and only if the references point to the same object, but equals() is often overridden for some other type of comparison. Because there is no method call in Java that mimics the behavior of the language's == operator as applied to object references, *J/Link* needs a SameObjectQ function that provides that behavior for *Mathematica* programmers.

In an unusual case where you need to compare object references for equality a very large number of times, the comparative slowness of SameObjectQ compared to SameQ could become an issue. The only thing that could cause two JavaObject expressions that refer to the exact same Java object to be not SameQ is if the hashCode() value of the object changed between the times that the two JavaObject expressions were created. If you know this has not happened, then you can safely use SameQ as the test whether they refer to the same object.

ReleaseJavaObject

The Java language has a built-in facility called "garbage collection" for freeing up memory occupied by objects that are no longer in use by a program. Objects become eligible for garbage collection when no references to them exist anywhere, except perhaps in other objects that are also unreferenced. When an object is returned to *Mathematica*, either as a result of a call to `JavaNew` or as the return value of a method call or field access, the *J/Lin*k code holds a special reference to the object on the Java side to ensure that it cannot be garbage-collected while it is in use by *Mathematica*. If you know that you no longer need to use a given Java object in your *Mathematica* session, you can explicitly tell *J/Link* to release its reference. The function that does this is `ReleaseJavaObject`. In addition to releasing the *Mathematica*-specific reference in Java, `ReleaseJavaObject` clears out internal definitions for the object that were created in *Mathematica*. Any subsequent attempt to use this object in *Mathematica* will fail.

```
frm = JavaNew["java.awt.Frame"]
```

```
«JavaObject[java.awt.Frame] »
```

Now tell Java that you no longer need to use this object from *Mathematica*.

```
ReleaseJavaObject[frm]
```

It is now an error to refer to `frm`.

`ReleaseJavaObject[`*obj*`]`	let Java know that you are done using *obj* in *Mathematica*
`ReleaseObject[`*obj*`]`	deprecated; replaced by `ReleaseJavaObject` in *J/Link* 2.0
`JavaBlock[`*expr*`]`	all novel Java objects returned to *Mathematica* during the evaluation of *expr* will be released when *expr* finishes
`BeginJavaBlock[]`	all novel Java objects returned to *Mathematica* between now and the matching `EndJavaBlock[]` will be released
`EndJavaBlock[]`	release all novel objects seen since the matching `BeginJavaBlock[]`
`LoadedJavaObjects[]`	return a list of all objects that are in use in *Mathematica*
`LoadedJavaClasses[]`	return a list of all classes loaded into *Mathematica*

J/Link memory management functions.

Calling `ReleaseJavaObject` will not necessarily cause the object to be garbage-collected. It is quite possible that other references to it exist in Java. `ReleaseJavaObject` does not tell Java to throw the object away, only that it does not need to be kept around solely for *Mathematica*'s sake.

An important fact about the references that *J/Link* maintains for objects sent to *Mathematica* is that only one reference is kept for each object, no matter how many times it is returned to *Mathematica*. It is your responsibility to make sure that after you call `ReleaseJavaObject`, you never attempt to use that object through any reference that might exist to it in your *Mathematica* session.

```
frm = JavaNew["java.awt.Frame"];
b1 = JavaNew["java.awt.Button"];
```

The `add()` method of the `Frame` class returns the object added, so `b2` refers to the same object as `b1`:

```
b2 = frm@add[b1];
```

If you call `ReleaseJavaObject[b1]`, it is not the *Mathematica* symbol `b1` that is affected, but the Java object that `b1` refers to. Therefore, using `b2` is also an error (or any other way to refer to this same `Button` object, such as `%`).

Calling `ReleaseJavaObject` is often not necessary in casual use. If you are not making heavy use of Java in your session then you will usually not need to be concerned about keeping track of what objects may or may not be needed anymore—you can just let them pile up. There are special times, though, when memory use in Java will be important, and you may need the extra control that `ReleaseJavaObject` provides.

JavaBlock

`ReleaseJavaObject` is provided mainly for developers who are writing code for others to use. Because you can never predict how your code will be used, developers should always be sure that their code cleans up any unnecessary references it creates. Probably the most useful function for this is `JavaBlock`.

`JavaBlock` automates the process of releasing objects encountered during the evaluation of an expression. Often, a *Mathematica* program will need to create some Java objects with `JavaNew`, operate with them, perhaps causing other objects to be returned to *Mathematica* as the results of method calls, and finally return some result such as a number or string. Every Java object encountered by *Mathematica* during this operation is needed only during the lifetime of the program, much like the local variables provided in *Mathematica* by `Block` and `Module`, and in C, C++, Java, and many other languages by block scoping constructs (e.g., {}). `JavaBlock` allows you to mark a block of code as having the property that any new objects returned to *Mathematica* during the evaluation are to be treated as temporary, and released when `JavaBlock` finishes.

It is important to note that the preceding sentence said "new objects". `JavaBlock` will not cause every object encountered during the evaluation to be released, only those that are being encountered for the first time. Objects that have already been seen by *Mathematica* will not be affected. This means that you do not have to worry that `JavaBlock` will aggressively release an object that is not truly temporary to that evaluation.

It is not enough simply to call `ReleaseJavaObject` on every object you create with `JavaNew`, because many Java method calls return objects. You may not be interested in these return values, or you may never assign them to a named variable because they may be chained together with other calls (as in `obj@returnsObject[]@foo[]`), but you still need to release them. Using `JavaBlock` is an easy way to be sure that all novel objects are released when a block of code finishes.

`JavaBlock[`*expr*`]` returns whatever *expr* returns.

Many *J/Link Mathematica* programs will have the following structure:

```
MyFunc[args__] :=
     JavaBlock[
          Module[{locals},
               ...
           ]
     ]
```

It is very common to write a function that creates and manipulates a number of `JavaObject` expressions, and then returns one of them, the rest being temporary. To facilitate this, if the return value of a `JavaBlock` is a single `JavaObject`, it will not be released.

```
MyOtherFunc[args__] :=
    JavaBlock[
        Module[{obj},
             ...
            obj = JavaNew["java.awt.Frame"];
            ...
            Return[obj]
   (* OK: obj will not be released when JavaBlock finishes. *)
        ]
    ]
```

New in *J/Link* 2.1 is the `KeepJavaObject` function, which allows you to specify an object or sequence of objects that should not be released when the `JavaBlock` ends. Calling `KeepJavaObject` on a single object or sequence of objects means they will not be released when the first enclosing `JavaBlock` ends. If there is an outer enclosing `JavaBlock`, the objects will be freed when *it* ends, however, so if you want the objects to escape a nested set of `JavaBlock` expressions, you must call `KeepJavaObject` at each level. Alternatively, you can call

KeepJavaObject[*obj*, Manual], where the Manual argument tells *J/Link* that the object should not be released by any enclosing JavaBlock expressions. The only way such object will be released is if you manually call ReleaseJavaObject on it. Here is an example that uses KeepJavaObject to allow you to return a list of two objects without them being released:

```
MyOtherFunc[args__] :=
    Module[{obj1, obj2, obj3},
        JavaBlock[
            obj1 = JavaNew["java.awt.Frame"];
            obj2 = JavaNew["java.awt.Button"];
            obj3 = JavaNew["SomeTemporaryObject"];
            ...
            KeepJavaObject[obj1, obj2];
            {obj1, obj2}
        ]
     ]
```

BeginJavaBlock and EndJavaBlock can be used to provide the same functionality as JavaBlock across more than one evaluation. EndJavaBlock releases all novel Java objects returned to *Mathematica* since the previous matching BeginJavaBlock. These functions are mainly of use during development, when you might want to set a mark in your session, do some work, and then release all novel objects returned to *Mathematica* since that point. BeginJavaBlock and EndJavaBlock can be nested. Every BeginJavaBlock should have a matching EndJavaBlock, although it is not a serious error to forget to call EndJavaBlock, even if you have nested levels of them—you will only fail to release some objects.

LoadedJavaObjects and LoadedJavaClasses

LoadedJavaObjects[] returns a list of all Java objects that are currently referenced in *Mathematica*. This includes all objects explicitly created with JavaNew and all those that were returned to *Mathematica* as the result of a Java method call or field access. It does not include objects that have been released with ReleaseJavaObject or through JavaBlock. LoadedJavaObjects is intended mainly for debugging. It is very useful to call it before and after some function you are working on. If the list grows, your function leaks references, and you need to examine its use of JavaBlock and/or ReleaseJavaObject.

LoadedJavaClasses[] returns a list of JavaClass expressions representing all classes loaded into *Mathematica*. Like LoadedJavaObjects, LoadedJavaClasses is intended mainly for debugging. Note that you do not have to determine if a class has already been loaded before you call LoadJavaClass. If the class has been loaded, LoadJavaClass does nothing but return the appropriate JavaClass expression.

Exceptions

How Exceptions Are Handled

J/Link handles Java exceptions automatically. If an uncaught exception is thrown during any call into Java, you will get a message in *Mathematica*. Here is an example that tries to format a real number as an integer.

```
LoadClass["java.lang.Integer"];
Integer`parseInt["1234.5"]
```

```
Java::excptn :
    A Java exception occurred : java.lang.ArrayIndexOutOfBoundsException.
$Failed
```

If the exception is thrown before the method returns a result to *Mathematica*, as in the example, the result of the call will be `$Failed`. As discussed later in "Manually Returning a Result to *Mathematica*", it is possible to write your own methods that manually send a result to *Mathematica* before they return. In such cases, if an exception is thrown after the result is sent to *Mathematica* but before the method returns, you will get a warning message reporting the exception, but the result of the call will be unaffected.

If the Java code was compiled with debugging information included, the *Mathematica* message you get as a result of an exception will show the full stack trace to the point where the exception occurred, with the exact line numbers in each file.

The JavaThrow Function

In some cases, you may want to cause an exception to be thrown in Java. This can be done with the `JavaThrow` function. `JavaThrow` is new in *J/Link* 2.0 and should be considered experimental. Its behavior might change in future versions.

`JavaThrow[`*exceptionObj*`]`	throw the given exception object in Java

Throwing Java exceptions from *Mathematica.*

You will only want to use `JavaThrow` in *Mathematica* code that is itself called from Java. It is quite common for *J/Link* programs written in *Mathematica* to involve both calls from *Mathematica* into Java and calls from Java back to *Mathematica*. Such "callbacks" to *Mathematica* are used extensively in *Mathematica* programs that create Java user interfaces, as described in

detail later in the section "Creating Windows and Other User Interface Elements". For example, you can associate a *Mathematica* function to be called when the user clicks a Java button. This *Mathematica* function is called directly from Java, and you might want it to behave just like a Java method, including having the ability to throw Java exceptions.

An example of throwing an exception in a callback from a user interface action like clicking a button is not very realistic because there is typically nothing in Java to catch such exceptions; thus they are essentially ignored. A more meaningful example would be a program that involved a mix of Java and *Mathematica* code where, for flexibility and ease of development reasons, you have a *Mathematica* function being called to implement the "guts" of a Java method that can throw an exception. As a concrete example, say you are doing XML processing with Java and *Mathematica* using the SAX (Simple API for XML) API. SAX processing is based on a set of handler methods that are called as certain events occur during parsing of the XML document. Each such method can throw a `SAXException` to indicate an error and halt the parsing. You want to implement these handler methods in *Mathematica* code, and thus you want a way to throw a `SAXException` from *Mathematica*. Here is a hypothetical example of one such handler method, the `startDocument()` method, which is invoked by the SAX engine when document processing starts:

```
startDocument[] :=
 If[! readyToAcceptParsingEvents, JavaThrow[JavaNew["org.xml.sax.SAXException",
    "Mathematica code has not been initialized"]]]
```

After a call to `JavaThrow`, the rest of the *Mathematica* function executes normally, but there is no result returned to Java.

Returning Objects "by Value" and "by Reference"

References and Values

J/Link provides a mapping between certain *Mathematica* expressions and their Java counterparts. What this means is that these *Mathematica* expressions are automatically converted to and from their Java counterparts as they are passed between *Mathematica* and Java. For example, Java integer types (`long`, `short`, `int`, and so on) are converted to *Mathematica* integers and Java real types (`float` and `double`) are converted to *Mathematica* real numbers. Another mapping is that Java objects are converted to `JavaObject` expressions in *Mathematica*. These `JavaObject` expressions are *references* to Java objects—they have no meaning in *Mathematica* except as they are manipulated by *J/Link*. However, some Java objects are things that have meaningful values in *Mathematica*, and these objects are by default converted to values. Examples of such objects are strings and arrays.

You could say, then, that Java objects are by default returned to *Mathematica* "by reference", except for a few special cases. These special cases are strings, arrays, complex numbers (discussed later), `BigDecimal` and `BigInteger` (discussed later), and the "wrapper" classes (e.g., `java.lang.Integer`). You could say that these exceptional cases are returned "by value". The table in "Conversion of Types between Java and *Mathematica*" shows how these special Java object types are mapped into *Mathematica* values.

In summary, every Java object that has a meaningful value representation in *Mathematica* is converted into this value, simply because that is the most useful behavior. There are times, however, when you might want to override this default behavior. Probably the most common reason for doing this is to avoid unnecessary traffic of large expressions over *MathLink*.

`ReturnAsJavaObject[`*expr*`]`	a Java object returned by *expr* will be in the form of a reference
`ByRef[`*expr*`]`	deprecated; replaced by `ReturnAsJavaObject` in *J/Link* 2.0
`JavaObjectToExpression[`*obj*`]`	give the value of the Java object *obj* as a *Mathematica* expression
`Val[`*obj*`]`	deprecated; replaced by `JavaObjectToExpression` in *J/Link* 2.0

"By reference" and "by value" control.

ReturnAsJavaObject

Consider the case where you have a static method in class `MyClass` called `arrayAbs()` that takes an array of doubles and returns a new array where each element is the absolute value of the corresponding element in the argument array. The declaration of this method thus looks like `double[] arrayAbs(double[] a)`. This is how you would call such a method from *Mathematica*.

```
LoadJavaClass["MyClass", StaticsVisible → True];
arrayAbs[{1., -2., 3., 4.}]
```

```
{1., 2., 3., 4.}
```

The example showed how you probably want the method to work: you pass a *Mathematica* list and get back a list. Now assume you have another method named `arraySqrt()` that acts like `arrayAbs()` except that it performs the `sqrt()` function instead of `abs()`.

```
arraySqrt[arrayAbs[{1., -2., 3., 4.}]]
```

```
{1., 1.41421, 1.73205, 2.}
```

In this computation, the original list is sent over *MathLink* to Java and a Java array is created with these values. That array is passed as an argument to `arrayAbs()`, which itself creates and returns another array. This array is then sent back to *Mathematica* via *MathLink* to create a list, which is then promptly sent back to Java as the argument for `arraySqrt()`. You can see that it was a waste of time to send the array data back to *Mathematica*—you had a perfectly good array (the one returned by the `arrayAbs()` method) living on the Java side, ready to be passed to `arraySqrt()`, but instead you sent its contents back to *Mathematica* only to have it immediately come back to Java again as a new array with the same values! For this example, the cost is negligible, but what if the array has 200,000 elements?

What is needed is a way to let the array data remain in Java and return only a reference to the array, not the actual data itself. This can be accomplished with the `ReturnAsJavaObject` function.

```
ReturnAsJavaObject[arrayAbs[{1., -2., 3., 4.}]]
«JavaObject[[D] »
```

Note that the class name of the `JavaObject` is `"[D"`, which, although a bit cryptic, is the actual Java class name of a one-dimensional array of doubles. Here is how the computation looks using `ReturnAsJavaObject`:

```
arraySqrt[ReturnAsJavaObject[arrayAbs[{1., -2., 3., 4.}]]]
{1., 1.41421, 1.73205, 2.}
```

Earlier you saw `arraySqrt()` being called with an argument that was a *Mathematica* list of reals. Here it is being called with a reference to a Java object that is a one-dimensional array of doubles. All methods and fields that take an array can be called from *Mathematica* with either a *Mathematica* list or a reference to a Java array of the appropriate type.

Strings are the other type for which `ReturnAsJavaObject` is useful. Like arrays, strings have the two properties that (1) they are represented in Java as objects but also have a meaningful *Mathematica* value, and (2) they can be large, so it is useful to be able to avoid passing their data back and forth unnecessarily. As an example, say your class `MyClass` has a static method that appends to a string a digit taken from an external device that you are controlling from Java. It takes a string and returns a new one, so its signature is `static String appendDigit (String s)`. You have a *Mathematica* variable named `veryLongString` that holds a long string, and you want to append to this string 100 times. This code will cause the string contents to make 100 round trips between *Mathematica* and Java.

```
Do[veryLongString = appendString[veryLongString], {100}];
```

Using ReturnAsJavaObject lets the strings remain on the Java side, and thus it generates virtually no *MathLink* traffic.

```
Do[veryLongString = ReturnAsJavaObject[appendString[veryLongString]], {100}];
```

This example is somewhat contrived, since repeatedly appending to a growing string is not a very efficient style of programming, but it illustrates the issues.

When the Do loop is executed, veryLongString gets assigned values that are not *Mathematica* strings, but JavaObject expressions that refer to strings residing in Java. That means that appendString () gets called with a *Mathematica* string the very first iteration, but with a JavaObject expression thereafter. As is the case with arrays, any Java method or field that takes a string can be called in *Mathematica* either with a string or a JavaObject expression that refers to one. The veryLongString variable started out holding a string, but at the end of the loop it holds a JavaObject expression.

```
veryLongString
«JavaObject[java.lang.String] »
```

At some point, you probably want an actual *Mathematica* string, not this string object reference. How do you get the value back? You will visit this example again later when the JavaObjectToExpression function is introduced.

In summary, the ReturnAsJavaObject function causes methods and fields that return objects that would normally be converted into *Mathematica* values to return references instead. It is an optimization to avoid unnecessarily passing large amounts of data between *Mathematica* and Java, and as such it will be useful primarily for very large arrays and strings. As with all optimizations, you should not concern yourself with ReturnAsJavaObject unless you have some code that is running at an unacceptable speed, or you know ahead of time that the code you are writing will benefit measurably from it. Objects of most Java classes have no meaningful "by value" representation in *Mathematica*, and they are always returned "by reference". ReturnAsJavaObject will have no effect in these cases.

Finally, note that ReturnAsJavaObject has no effect on methods in which the Java programmer manually sends the result back to *Mathematica* (this topic is discussed later in this User Guide). Manually returning a result bypasses the normal result-handling routines in *J/Link*, so there is no chance for the ReturnAsJavaObject request to be accommodated.

JavaObjectToExpression

In the previous section, you saw how the `ReturnAsJavaObject` function can be used to cause objects normally returned to *Mathematica* by value to be returned by reference. It is necessary to have a function that does the reverse—takes a reference and converts it to its value representation. That function is `JavaObjectToExpression`.

Returning to the earlier `appendString` example, you used `ReturnAsJavaObject` to avoid costly passing of string data back and forth over *MathLink*. The result of this was that the `veryLongString` variable now held a `JavaObject` expression, not a literal *Mathematica* string. `JavaObjectToExpression` can be used to get the value of this string object as a *Mathematica* string.

```
JavaObjectToExpression[veryLongString]
```

```
037118086362644534489492294928989287822791948284089742269122236592851667829700627394053209887
   2893368
```

The majority of Java objects have no meaningful value representation in *Mathematica*. These objects can only be represented in *Mathematica* as `JavaObject` expressions, and using `JavaObjectToExpression` on them has no effect.

The `ReturnAsJavaObject` function is not the only way to get a `JavaObject` expression for an object that is normally returned to *Mathematica* as a value. The `JavaNew` function always returns a reference.

```
JavaNew["java.lang.String", "a string"]
```

```
«JavaObject[java.lang.String] »
```

```
JavaObjectToExpression[%]
```

```
a string
```

The next section introduces the `MakeJavaObject` function, which is easier than using `JavaNew` to construct Java objects out of *Mathematica* strings and arrays.

MakeJavaObject and MakeJavaExpr

Preamble

In addition to `JavaNew`, which calls a class constructor, *J/Link* provides two convenience functions for creating Java objects from *Mathematica* expressions. These functions are

MakeJavaObject and MakeJavaExpr. Do not get them confused, despite their similar names. MakeJavaObject is a commonly used function for constructing objects of some special types. MakeJavaExpr is an advanced function that creates an object of *J/Link*'s Expr class representing an arbitrary *Mathematica* expression.

MakeJavaObject

MakeJavaObject[*val*]	construct an object of the appropriate type to represent the *Mathematica* expression *val* (numbers, strings, lists, and so on)

MakeJavaObject.

When you call a Java method from *Mathematica* that takes, say, a Java String object, you can call it with a *Mathematica* string. The internals of *J/Link* will construct a Java string that has the same characters as the *Mathematica* string, and pass that string to the Java method. Sometimes, however, you want to pass a string to a method that is typed to take Object. You cannot call such a method from *Mathematica* with a string as the argument because although *J/Link* recognizes that a *Mathematica* string corresponds to a Java string, it does not recognize that a *Mathematica* string corresponds to a Java Object. It does this deliberately, for the sake of imposing as much type safety as possible on calls into Java. For this example, assume that the class MyClass has a method with the following signature:

```
void foo(Object obj);
```

Assume also that theObj is an object of this class, created with JavaNew. Try to call foo with a literal string.

```
theObj@foo["this is a string"]
```

```
Java::argxs :
    The method foo was called with an incorrect number or type of arguments.
$Failed
```

It fails for the reason given above. To call a Java method that is typed to take an Object with a string, you must first explicitly create a Java string object with the appropriate value. You can do this using JavaNew.

```
javaStr = JavaNew["java.lang.String", "this is a string"]
```

```
«JavaObject[java.lang.String] »
```

Now it works, because the argument is a JavaObject expression.

```
theObj@foo[javaStr]
```

To avoid having to call JavaNew to create a Java string object, *J/Link* provides the MakeJavaObject function.

```
javaStr = MakeJavaObject["this is a string"];
```

In the case of a string, MakeJavaObject just calls JavaNew for you. Of course, it would not be of much use if it could only construct String objects. The same issue arises with other Java objects that are direct representations of *Mathematica* values. This includes the "wrapper" classes like java.lang.Integer, java.lang.Boolean, and so on, and the array classes. If you want to call a Java method that takes a java.lang.Integer as an argument, you can call it from *Mathematica* with a raw integer. But if you want to pass an integer to a method that is typed to take an Object, you must explicitly create an object of type java.lang.Integer—*J/Link* will not construct one automatically from an integer argument. It is simpler to call MakeJavaObject than JavaNew for this.

```
MakeJavaObject[42]
```

```
«JavaObject[java.lang.Integer] »
```

When given an integer argument, MakeJavaObject always constructs a java.lang.Integer, never a java.lang.Short, java.lang.Long, or other "integer" Java wrapper object. Similarly, if you call MakeJavaObject with a real number, it creates a java.lang.Double, never a java.lang.Float. If you require an object of one of these other types, you will have to call JavaNew explicitly.

MakeJavaObject also works for Boolean values.

```
MakeJavaObject[True]
```

```
«JavaObject[java.lang.Boolean] »
```

If MakeJavaObject were only a shortcut for calling JavaNew, it would not be all that useful. It becomes indispensable, however, for creating objects of an array class. Recall that in Java, arrays are objects and they belong to a class. These classes have cryptic names, but if you know them you can create array objects with JavaNew. When creating array objects, the second argument to JavaNew is a list giving the length in each dimension. Here you create a 2×3 array of ints.

```
intArray2D = JavaNew["[[I", {2, 3}]
«JavaObject[[[I] »
```

JavaNew lets us create array objects, but it does not let us supply initial values for the elements of the array. MakeJavaObject, on the other hand, takes a *Mathematica* list and converts it into a Java array object with the same values.

```
intArray2D = MakeJavaObject[{{1, 2, 3}, {4, 5, 6}}]
«JavaObject[[[I] »
```

Thus, MakeJavaObject is particularly useful for creating array objects, because it lets you supply the initial values for the array elements, and it frees you from having to learn and remember the names of the Java array classes ([[I for a two-dimensional array of ints, [D for a one-dimensional array of doubles, and so on). MakeJavaObject can create arrays up to three dimensions deep of integers, doubles, strings, Booleans, and objects.

The JavaObjectToExpression function is discussed in the section "JavaObjectToExpression", and you can think of MakeJavaObject as being the inverse of JavaObjectToExpression. MakeJavaObject takes a *Mathematica* expression that has a corresponding Java object that can represent its value, and creates that object. It literally "makes it into a Java object". The JavaObjectToExpression function goes the other way—it takes a Java object that has a meaningful *Mathematica* representation and converts it into that expression. It will always be the case that, for any x that MakeJavaObject can operate on,

```
JavaObjectToExpression[MakeJavaObject[x]] === x
```

Remember that MakeJavaObject is not a commonly used function. You do not need to explicitly construct Java objects from *Mathematica* strings, arrays, and so on, just to pass them to Java methods—*J/Link* does this automatically for you. But even though *J/Link* will create objects automatically from certain arguments in most circumstances, it will not do so when an argument is typed as a generic Object. Then you must create a JavaObject yourself, and MakeJavaObject is the easiest way to do this.

The code for the SetInternetProxy function discussed in the section SetInternetProxy provides a concrete example of using MakeJavaObject. To specify proxy information (for users behind firewalls), you need to set some system properties using the Properties class. This class is a subclass of Hashtable, so it has a method with the signature

```
Object put(Object key, Object value);
```

You should always specify keys and values for `Properties` in the form of strings. Thus, you might try this from *Mathematica*.

```
LoadJavaClass["java.lang.System"];
System`getProperties[]@put["proxySet", "true"]
```

```
Java::argx :
  Method named put defined in class java.util.Properties was called with
    an incorrect number or type of arguments. The
    arguments, shown here in a list, were {proxySet, true}.
```

```
$Failed
```

For this to work, you need to use `MakeJavaObject` to create Java `String` objects:

```
System`getProperties[]@put[MakeJavaObject["proxySet"], MakeJavaObject["true"]]
```

MakeJavaExpr

To understand the `MakeJavaExpr` function, you need to understand the motivation for *J/Link*'s `Expr` class, which is discussed in detail in "Motivation for the Expr Class". Basically, an `Expr` is a Java object that can represent an arbitrary *Mathematica* expression. Its main use is as a convenience for Java programmers who want to examine and operate on *Mathematica* expressions in Java. Sometimes it is useful to have a way of creating `Expr` objects in the *Mathematica* language instead of from Java. `MakeJavaExpr` is the function that fills this need.

`MakeJavaExpr[`*expr*`]`	construct an object of *J/Link*'s `Expr` class that represents the *Mathematica* expression

`MakeJavaExpr`.

Note that if you are calling a Java method that is typed to take an `Expr`, then you do not have to call `MakeJavaExpr` to construct an `Expr` object. *J/Link* will automatically convert any expression you supply as the argument to an `Expr` object, as it does with other automatic conversions. Like `MakeJavaObject`, `MakeJavaExpr` is used in cases where you are calling a method that takes a generic `Object`, not an `Expr`, and therefore *J/Link* will not perform any automatic conversion for you. In such cases you need to explicitly create an `Expr` object out of some *Mathematica* expression. One reason you might want to do this is to store a *Mathematica* expression in Java for retrieval later.

Here is a simple example of `MakeJavaExpr`. This demonstrates a few methods from the `Expr` class, which has a number of *Mathematica*-like methods for examining, modifying, and extracting portions of expressions. Of course, this is a highly contrived example—if you wanted to know the length of an expression you would just call *Mathematica*'s `Length[]` function. The `Expr` methods demonstrated here are typically called from Java, not *Mathematica*.

```
e = MakeJavaExpr[1 + 2 x + x^2]
```

```
«JavaObject[com.wolfram.jlink.Expr] »
```

```
e@length[]
```

```
3
```

```
e@part[3]
```

$$x^2$$

```
e@insert[x^3, -1]
```

$$1 + 2\,x + x^2 + x^3$$

Note that `Expr` objects, like *Mathematica* expressions, are immutable. The above call to `insert()` did not modify `e`; instead, it returned a new `Expr`.

```
JavaObjectToExpression[e]
```

$$1 + 2\,x + x^2$$

If you are having trouble understanding why you might want to use `MakeJavaExpr` in a *Mathematica* program, do not worry. It is an advanced function that few programmers will have any use for.

Creating Windows and Other User Interface Elements

Preamble

One of the most useful applications for *J/Link* is to write user interface elements for *Mathematica* programs. Examples of such elements would be a progress bar monitoring the completion of a computation, a window that displays an image or animation, a dialog box that prompts the user for input or helps them compose a proper call of an unfamiliar function, or a mini-application that leads the user through the steps of an analysis. These types of user interfaces are distinct from what you might write for a Java program that uses *Mathematica* in the background in that they "pop up" when the user invokes some *Mathematica* code. They do not replace the notebook front end, they just augment it. In this way, they are like an extension of the palettes and other specialty notebook elements you can create in the front end.

Mathematica with *J/Link* is an extremely powerful and productive environment for creating user interfaces. The complexity of user interface code is ideally suited to the interactive line-at-a-time nature of *J/Link* development. You can literally build, modify, and experiment with your user interface *while it is running*.

Anyone considering writing user interfaces for *Mathematica* programs should also look at *GUIKit* . *GUIKit* is built on top of *J/Link*, and provides an extremely high-level means of creating interfaces. Further discussion of *GUIKit* is beyond the scope of this manual, but be aware that *GUIKit* was specifically designed to provide an easier means of creating user interfaces than writing in "raw" *J/Link*, as described here.

Interactive and Non-Interactive Interfaces

To write *Mathematica* programs that create Java windows you need to understand important distinctions between several types of such user interfaces. These distinctions relate to how they interact with the *Mathematica* kernel.

At the highest level of categorization, there is a distinction between "interactive" and "non-interactive" interfaces. The interactiveness under consideration here is with the *Mathematica* kernel, not with the user. What we are calling non-interactive user interfaces have no need to communicate back to *Mathematica*, although they typically are controlled by *Mathematica*. Such interfaces often accept no user input at all—they are created, manipulated, and destroyed by *Mathematica* code. An example of this type is a window that shows a progress bar (a complete progress bar program is presented in "A Progress Bar"). A progress bar does not return a result to *Mathematica* and it does not need to respond to user actions, at least not by interacting with *Mathematica*. In other words, the window may go away when its close box is clicked (a user action), but this is not relevant to *Mathematica* because it does not return a result or trigger a call back into *Mathematica*. A progress bar is completely driven by a *Mathematica* program. The flow of information is in one direction only.

Such user interfaces typically have lifetimes that are encompassed by a single *Mathematica* program, as is the case with a progress bar. This is not required, however. Hosting an applet in its own window, as described in "Hosting Applets", is an example where the window lives on after the code that created it ends execution. The applet window is only dismissed when the user clicks in its close box. Again, though, the important property is that the applet does not need to interact with *Mathematica*.

This type of user interface, which requires no interaction back with *Mathematica*, poses no special issues that need to be discussed in this section. A program that creates, runs, and destroys such an interface is very much like a non-GUI Java computation that is accomplished with a series of calls into Java. It just happens to produce a visual effect. You can examine the progress bar code in "A Progress Bar" if you want to see a fully fleshed out example.

The more common "interactive" type of user interface needs to communicate back to *Mathematica*. This might be to return a result, like a typical modal input dialog, or to initiate a computation as a consequence of the user clicking a button. To understand the special problem this imposes, it is useful to examine some basic considerations about the kernel's "main loop", whereby it acquires input, evaluates it, and sends off any output.

When the *Mathematica* kernel is being used from the front end, it spends most of its life waiting for input to arrive on the *MathLink* that it uses to communicate with the front end. This *MathLink* is given by `$ParentLink`, and it is `$ParentLink` that has the kernel's "attention". When input arrives on `$ParentLink`, it is evaluated, any results are sent back on the link, and the kernel goes back to waiting for more input on `$ParentLink`. When *J/Link* is being used, the kernel has another *MathLink* open, the one that connects to the Java runtime. When you execute some code that calls into Java, the kernel sends something to Java and then blocks waiting for the return value from Java. During this period when the kernel is waiting for a return value from Java, the Java link has the kernel's attention. It is only during this period of time that the kernel is paying attention to the Java link. A more general way of saying this is that the kernel is only listening for input arriving from Java when it has been specifically instructed to do so. The rest of the time it is listening only to `$ParentLink`, which is typically the notebook front end.

Consider what happens when the user clicks on a button in your Java window and that button tries to execute some code that calls into *Mathematica*. The Java side sends something to *Mathematica* and then waits for the result, but the kernel will never get the request because it is not paying attention to the Java link. It is necessary to use some means to tell the kernel to look for input arriving on the Java link. *J/Link* provides three different ways to manage the kernel's attention to the Java link, and thereby control its readiness to accept requests for evaluations initiated by the Java side.

These three ways can be called "modal", "modeless", and "manual". In modal interaction, characterized by the use of the `DoModal` *Mathematica* function, the kernel is pointed at the Java link until the Java side releases it. The kernel is a complete slave to the Java side, and is unavailable for any other computations. In modeless interaction, characterized by the use of the `ShareKernel` *Mathematica* function, the kernel is kept in a state where it is receptive to evaluation requests arriving from either the notebook front end or Java, evenly sharing its attention between these two programs. Lastly, there is a manual mode, characterized by the use of the `ServiceJava` *Mathematica* function, which in some ways is intermediate between modal and modeless operation. Here, you manually instruct the kernel to allow single requests from Java while in the middle of running a larger program. The next few sections are devoted to further exploration of these types of user interfaces.

Before continuing, it is important to remember that all these issues about how to prepare the kernel for computations arriving from Java are only relevant for computations *initiated* in Java, typically by user actions like clicking a button. Calls from Java to *Mathematica* that are part of a back-and-forth series of calls that involve a call from *Mathematica* into Java are not a problem. Any time *Mathematica* has called into Java, *Mathematica* is actively listening for results arriving from Java. This may sound confusing, but that is mostly because it is only in a much later section that discusses writing your own Java methods to be called from *Mathematica*; such methods can call back to *Mathematica* for computations before they return their result (typical examples are to print something in the notebook window or display a message). These are true *callbacks* into *Mathematica*, and *Mathematica* is always ready to handle them. In contrast, calls to *Mathematica* that occur as the result of a user action in the Java side are, in effect, a surprise to *Mathematica*, and it is not normally in a state where it is ready to accept them.

Modal versus Modeless Operation

A common type of user interface element is like a modal dialog: once it is displayed, the *Mathematica* program hangs waiting for the user to dismiss the window. Typically, this is because the window returns a result to *Mathematica*, so it is not meaningful for *Mathematica* to continue until the window is closed. An example of such a window is a simple input window that asks the user for some value, which it returns to *Mathematica* when the **OK** button is clicked.

It is important to understand our slightly generalized use of the term "modal" to describe these windows. They may not be modal in the traditional sense that they must be dismissed before

anything else can be done in the user interface. Rather, they are modal with respect to the *Mathematica* kernel—the kernel cannot do anything else until they are closed. A Java window that you create might not be modal with respect to other Java windows on the screen (i.e., a dialog might not have the `isModal` property set), but it ties up the kernel's attention until it is dismissed.

Another type of user interface element is like a modeless dialog: after it is displayed, the *Mathematica* program that created it will finish, leaving the window visible and usable while the user continues working in the notebook front end. This sounds a lot like the first type of user interface element described earlier, but these windows are distinguished by the fact that they can initiate interactions with *Mathematica* while they are visible. An example would be a window that lets users load packages into *Mathematica* by selecting them from a scrolling list. You write a *J/Link* program that creates this window, displays it, and returns. The window is left open and usable until the user clicks in its close box. In the meantime, the user is free to continue working in the front end, going back to use this Java window whenever it is convenient.

Such a window is almost like another type of notebook or palette window in the front end. You can have any number of front end or Java windows open at once, and active, meaning that they can be used to initiate computations in *Mathematica*. They are each their own little interface onto the same kernel. What is different about the Java window is that it is much more general than a notebook window, and, importantly, it lives in a different application layer than the front end. This last fact makes the Java window in effect a second front end, rather than an extension of the notebook front end. To accommodate such a second front end, the kernel must be kept in a special state that allows it to handle requests for evaluations arriving from either the notebook front end or Java.

Before presenting examples of how to implement modal and modeless windows, it is necessary to jump ahead a little bit and explain the main mechanism by which Java user interface elements can communicate with *Mathematica*.

Handling Events with *Mathematica* Code: The "MathListener" Classes

User interface elements typically have active components like buttons, scrollbars, menus, and text fields that need to trigger some action when they are clicked. In the Java event model, components fire events in response to user actions, and other components indicate their interest in these events by registering as event listeners. In practice, though, components do not

usually act as event listeners directly. Instead, the programmer writes an adapter class that implements the desired event-listener interface and calls certain methods in the component in response to various events. This avoids having to subclass the responding component just to have it act as an event listener. The only specialty code goes into the adapter class, allowing the components that fire and respond to events to be generic.

As an example, say you are writing a standard Java program and you have a button that you want to use to control the appearance of a text area. Clicking the button should toggle between black text on a white background and white text on a black background. Buttons fire `ActionEvent`s when they are clicked, and a class that wants to receive notifications of clicks must implement the `ActionListener` interface, and register with the button by calling its `addActionListener` method. You would write a class, perhaps called `MyActionAdapter`, that implements `ActionListener`. In its `actionPerformed()` method, which is what will be called when the button is clicked, you would call the appropriate methods to set the foreground and background colors of the text area.

If you have ever used a Java GUI builder that lets you create an application by dropping components on a form and then wiring them together via events, the code that is being generated for you consists in large part of adapter classes that manage the logic of calling certain methods in the target objects when events are fired by the source objects.

What all this is leading up to is simply that the wiring of components in a GUI typically involves writing a lot of Java code in the form of classes that implement various event-listener interfaces. *J/Link* programmers want to write GUIs that use the standard Java event model, and they should not have to write Java code to do it. The solution is simple: *J/Link* provides a complete set of classes that implement the standard event-listener interfaces and whose actions are to call back into *Mathematica* to execute user-defined code. This brings all the event-handling logic down into *Mathematica*, where it can be scripted like every other part of the program.

Not only does this solution preserve the "pure *Mathematica*" property of even complex Java GUIs, it is vastly more flexible than writing a traditional application in Java. When you write in Java, or use a fancy drag-and-drop GUI builder, you hard-code the event logic. You have to decide at compile time what every click, scroll, and keystroke will do. But when you use *J/Link*, you decide how your program is wired together at run time. You can even change the behavior on the fly simply by typing a few lines of code.

J/Link provides implementations of all the standard AWT event-listener classes. These classes are named after the interfaces they implement, with "Math" prepended. Thus, the class that implements `ActionListener` is `MathActionListener`. (Perhaps these classes would be better named `MathXXXAdapter`.) The following table shows a summary of all the `MathListener` classes, the methods they implement, and the arguments they send to your *Mathematica* handler function.

class	*methods*	*arguments to Mathematica handler*
`MathActionListener`	`actionPerformed (ActionEvent e)`	`e, e.getActionCommand () (String)`
`MathAdjustmentListener`	`adjustmentValueChanged (AdjustmentEvent e)`	`e, e.getAdjustmentType (), (Integer) e.getValue () (Integer)`
`MathComponentListener`	`componentHidden (ComponentEvent e) componentShown (ComponentEvent e) componentResized (ComponentEvent e) componentMoved (ComponentEvent e)`	`e`
`MathContainerListener`	`componentAdded (ContainerEvent e) componentRemoved (ContainerEvent e)`	`e`
`MathFocusListener`	`focusGained (FocusEvent e) focusLost (FocusEvent e)`	`e`
`MathItemListener`	`itemStateChanged (ItemEvent e)`	`e, e.getStateChange () (Integer)`
`MathKeyListener`	`keyPressed (KeyEvent e) keyReleased (KeyEvent e) keyTyped (KeyEvent e)`	`e, e.getKeyChar (),(Integer) e.getKeyCode () (Integer)`

MathMouseListener	mouseClicked (MouseEvent e) mouseEntered (MouseEvent e) mouseExited (MouseEvent e) mousePressed (MouseEvent e) mouseReleased (MouseEvent e)	e, e.getX (), (Integer) e.getY (), (Integer) e.getClickCount () (Integer)
MathMouseMotionListener	mouseMoved (MouseEvent e) mouseDragged (MouseEvent e)	e, e.getX (), (Integer) e.getY (), (Integer) e.getClickCount () (Integer)
MathPropertyChangeListener	propertyChanged (PropertyChangeEvent e)	e
MathTextListener	textValueChanged (TextEvent e)	e
MathVetoableChangeListener	vetoableChange (PropertyChangeEvent e)	e (veto the change by returning False from your handler)
MathWindowListener	windowOpened (WindowEvent e) windowClosed (WindowEvent e) windowClosing (WindowEvent e) windowActivated (WindowEvent e) windowDeactivated (WindowEvent e) windowIconified (WindowEvent e) windowDeiconified (WindowEvent e)	e

Listener classes provided with *J/Link*.

As an example of how to read this table, take the MathKeyListener class. MathKeyListener implements the KeyListener interface, which contains the methods keyPressed(), keyReleased(), and keyTyped(). If you register a MathKeyListener object with a component that fires KeyEvents, then these three methods will be called in response to the key events they are named after. When any of these methods are called, they will call into *Mathematica* and exe-

cute a user-defined function, passing it three arguments: the KeyEvent object itself, followed by two integers that are the results of the event object's getKeyChar() and getKeyCode() methods. All the MathListener classes pass your handler function the event object itself, and a few, like this one, pass additional integer arguments that are commonly needed values. This just saves you the overhead of having to call back into Java to get these additional values.

To specify the *Mathematica* function associated with any of the methods of a MathListener object, call the object's setHandler() method. setHandler() takes two strings, the first of which is the name of the event-handler method (e.g., "actionPerformed" or "keyPressed"), and the second of which is the *Mathematica* function that should be called in response. The *Mathematica* function can be a name, as in "myButtonFunction" or a pure function (specified as a string). The reason for supplying the name of the actual Java method in the listener interface is that many of the listeners have multiple methods. setHandler() returns True if the handler was set correctly and False otherwise (for example, if the method you named is not spelled correctly).

obj@setHandler["*methodName*","*funcName*"]	set the *Mathematica* function that will be called when the MathListener object *obj*'s event-handler method methodName() is called.

Assigning the *Mathematica* function that will be called in response to an event notification.

The use of these classes will become clear in the simple examples that follow for modal and modeless windows, and in the more fully worked-out examples in the sections "A Simple Modal Input Dialog" and "A Piano Keyboard".

You are not required to use the *J/Link* MathListener classes for creating calls into *Mathematica* triggered by user actions. They are provided simply as a convenience. You could write your own classes to handle events and put calls into *Mathematica* directly into their code. All the "MathListener" classes in *J/Link* are derived from an abstract base class called, appropriately, MathListener. The code in MathListener takes care of all of the details of interacting with *Mathematica*, and it also provides the setHandler() methods that you use to associate events with *Mathematica* code. Users who want to write their own classes in MathListener style (for example, for one of the Swing-specific event-listener interfaces, which *J/Link* does not provide) are strongly encouraged to make their classes subclasses of MathListener to inherit all this

functionality. You should examine the source code for one of the concrete classes derived from `MathListener` (`MathActionListener` is probably the simplest one) to see how it is written. You can use this as a starting point for your own implementation. If you do not make your class a subclass of `MathListener`, and instead choose instead to write your own event-handler code that calls into *Mathematica*, you *must* read "Writing Your Own Event Handler Code".

Bringing Java Windows to the Foreground

If you are creating a Java window with a *Mathematica* program, you probably want that window to pop up in front of the notebook the user is working in, so that its presence becomes apparent. You might expect that the `toFront()` method of Java's `Window` class is what you would use for this, but this does not work on the Macintosh, and it works slightly differently on different Java runtimes on Windows. As a result of these differences, it is difficult to write a *Mathematica* program that behaves identically on all platforms and all Java virtual machines with respect to making Java windows visible in front of all other windows the user might see.

As a result of these unfortunate differences, *J/Link* provides a *Mathematica* function, `JavaShow`, which performs the proper steps on all configurations. You should use `JavaShow[`*window*`]` in place of `window@setVisible[True]`, `window@show[]`, or `window@toFront[]`. You will see `JavaShow` used in all the example programs. The argument to `JavaShow` must be a Java object that is an instance of a class that can represent a top-level window. Specifically, it must be of class `java.awt.Window` or a subclass. This includes the AWT `Frame` and `Dialog` windows, and also the Swing classes used for top-level windows (`JFrame`, `JWindow`, and `JDialog`).

`JavaShow[`*windowObj*`]`	make the specified Java window visible and bring it in front of all other windows, including notebook windows

Bringing a Java window to the foreground.

Modal Windows

Here is an example of a simple "modal" window. The window contains a button and a text field. The text field starts out displaying the value 1, and each time the button is clicked the value is incremented. The `com.wolfram.jlink.MathFrame` class is used for the enclosing window. `MathFrame` is a simple extension to `java.awt.Frame` that calls `dispose()` on itself when its close box is clicked (the standard `Frame` class does nothing).

```
frm = JavaNew["com.wolfram.jlink.MathFrame"];

button = JavaNew["java.awt.Button"];
textField = JavaNew["java.awt.TextField"];

frm@setLayout[JavaNew["java.awt.GridLayout"]];
frm@add[button];
frm@add[textField];
frm@pack[];
JavaShow[frm];
```

At this point, you should see a small frame window with a button on the left and a text field on the right. Now label the button and set the starting text for the field.

```
button@setLabel["++"];
textField@setText["1"];
```

Now you want to add behavior to the button that causes it to increment the text field value. Buttons fire `ActionEvents`, so you need an instance of `MathActionListener`.

```
buttonListener = JavaNew["com.wolfram.jlink.MathActionListener"];
```

It must be registered with the button by calling `addActionListener`.

```
button@addActionListener[buttonListener];
```

At this point, if you were to click the **++** button, the `actionPerformed()` method of your `MathActionListener` would be called (do not click the button yet!). You know from the `MathListener` table in the previous subsection that the `actionPerformed()` method will call a user-defined *Mathematica* function with two arguments: the `ActionEvent` object itself and the integer value that results from the event's `getActionCommand()` method.

You have not yet set the user-defined code to be called by the `actionPerformed()` method. That is done for all the `MathListener` classes with the `setHandler()` method. This method takes two strings, the first being the name of the method in the event-listener interface, and the second being the function you want called.

```
buttonListener@setHandler["actionPerformed", "buttonFunc"];
```

Now you need to define `buttonFunc`. It must be written to take two arguments, but in this example you are not interested in either argument.

```
buttonFunc[_, _] :=
 Module[{curText, newVal},
       curText = textField@getText[];
       newVal = ToExpression[curText] + 1;
       textField@setText[ToString[newVal]]
 ]
```

You are still not quite ready to try the button. If you click the button now, the Java user interface thread will hang because it will call into *Mathematica* trying to evaluate `buttonFunc` and wait for the result, but the result will never come because the kernel is not waiting for input to arrive on the Java link. What you need is a way to put the kernel into a state where it is continuously reading from the Java link. This is what makes the window "modal"—the kernel cannot do anything else until the window is closed. The function that implements this modal state is `DoModal`.

`DoModal[]`	put the kernel into a state where its attention is solely directed at the Java link
`EndModal[]`	what the Java program must call to make the `DoModal` function return, ending the modal state

Entering and exiting the modal state.

`DoModal` will not return until the Java program calls back into *Mathematica* to evaluate `EndModal[]`. While `DoModal` is executing, the kernel is ready to handle callbacks from Java—for example, from `MathListener` objects. The way to get the Java side to call `EndModal[]` is typically to use a `MathListener`. For example, if your window had **OK** and **Cancel** buttons, these should dismiss the window, so you would create `MathActionListener` objects and register them with these two buttons. These `MathActionListener` objects would be set to call `EndModal[]` in their `actionPerformed()` methods.

`DoModal` returns whatever the block of code that calls `EndModal[]` returns. You would typically use this return value to determine how the window was closed—for example, whether it was the **OK** or **Cancel** button. You could then take appropriate action. See "A Simple Modal Input Dialog" for an example of using the return value of `DoModal`.

In the present example, the only way to close the window is by clicking its close box. Clicking the close box fires a `windowClosing` event, so you use a `MathWindowListener` to receive notifications.

```
windowListener = JavaNew["com.wolfram.jlink.MathWindowListener"];
frm@addWindowListener[windowListener];
```

Now you assign the *Mathematica* function to be called when the close box is clicked. All you need it to do is call `EndModal[]`, so you can specify a pure function that ignores its arguments and does nothing but execute `EndModal[]`.

```
windowListener@setHandler["windowClosing", "EndModal[]&"];
```

The preceding few lines are a fine example of how to use a MathWindowListener to trigger a call to EndModal[] when a window's close box is clicked. You would use something similar to this, except with a MathActionListener, if you wanted to have an explicit **Close** button. In this example, though, there is an easier way. Mentioned earlier is that the MathFrame class is just a normal AWT Frame except that it calls dispose() on itself when its close box is clicked. Actually it has another useful property—it can also execute EndModal[] when its close box is clicked. Thus, if you use MathFrame as the top-level window class for your interfaces, you will not have to manually create a MathWindowListener to terminate the modal loop every time. To enable this behavior of MathFrame, you need to call its setModal method:

```
(***
 This is even easier than using the MathWindowListener above.
 We won't call it here, though, because we have already arranged
 for EndModal to be called, and bad things will happen if we try
 to call it twice.

frm@setModal[]

***)
```

You must not call setModal if you are not using DoModal. This is because after setModal has been called, the MathFrame will try to call into *Mathematica* when it is closed (to execute EndModal), and *Mathematica* needs to be in a state where it is ready for calls originating in Java. The same issue exists for any MathListener you create yourself.

Now that everything is ready, you can enter the modal state and use the window.

```
DoModal[]
```

When you are done playing with the window, click the close box in the frame, which will trigger a callback into *Mathematica* that calls EndModal[]. DoModal then returns, and the kernel is ready to be used from the front end. DoModal[] returns Null if you click the close box of a MathFrame.

Here is how the entire example looks when packaged into a single program. (The code for `SimpleModal` is also available as SimpleModal.nb in the JLink/Examples/Part1 directory.)

```
SimpleModal[] :=
    JavaBlock[
        Module[{frm, button, textField, windowListener,
                buttonListener, buttonFunc},

            (* Create the GUI components. *)
            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            button = JavaNew["java.awt.Button"];
            textField = JavaNew["java.awt.TextField"];

            (* Configure their properties. *)
            frm@setLayout[JavaNew["java.awt.GridLayout"]];
            frm@add[button];
            frm@add[textField];
            button@setLabel["++"];
            textField@setText["1"];
            frm@pack[];

            (* Create the listener and set its handler function. *)
            buttonListener =
                JavaNew["com.wolfram.jlink.MathActionListener"];
            buttonListener@setHandler["actionPerformed", ToString[buttonFunc]];
            button@addActionListener[buttonListener];

            (* Define buttonFunc. *)
            buttonFunc[_, _] :=
                JavaBlock[
                    Module[{curText, newVal},
                        curText = textField@getText[];
                        newVal = ToExpression[curText] + 1;
                        textField@setText[ToString[newVal]]
                    ]
                ];

            (* Make the window visible and bring it in front of any
               notebook windows. *)
            JavaShow[frm];

            (* Tell the frame to end the modal loop when it is closed. *)
            frm@setModal[];

            (* Enter the modal loop. *)
            DoModal[];
        ]
]
```

Remember that `DoModal` will not return until the Java side calls `EndModal`. You have to be a little careful when you call `DoModal` that you have already established a way for the Java side to trigger a call to `EndModal`. As explained earlier, you will typically have done this by using a `MathFrame` as the frame window and calling its `setModal` method, or by creating and registering a `MathListener` of your own that will call `EndModal` in response to a user action (such as clicking an **OK** or **Cancel** button). Once `DoModal` has begun, the kernel is not responsive to the

front end and thus it is too late to set anything up. If you call DoModal and realize that for some reason you cannot end it from Java, you can abort it from the front end by selecting **Evaluation ▸ Interrupt Evaluation** in the menu, and then in the resulting dialog, clicking the button labeled **Abort**.

There is one subtlety you might notice in the code for SimpleModal that is not directly related to *J/Link*. In the line that calls buttonListener@setHandler, you pass the name of the button function not as the literal string "buttonFunc", but as ToString[buttonFunc]. This is because buttonFunc is a local name in a Module, and thus its real name is not buttonFunc, but something like buttonFunc$42. To make sure you capture its true run-time name, you call ToString on the symbolic name. You could avoid this by simply not making the name buttonFunc local to the Module, but the way you have done it automatically cleans up the definition for buttonFunc when the Module finishes.

MathFrame and MathJFrame

You encountered the MathFrame class in this section, which is a useful top-level window class for *J/Link* programmers because it has three special properties. You have already encountered two of them: it calls dispose() on itself when it is closed, and it has the setModal() method, which gives it easy support for use with DoModal. The third property is that it has an onClose() method that you can use to specify *Mathematica* code that will be executed when the window is closed. The onClose() method is used in the Palette example in "Sharing the Front End: Palette-Type Buttons". *J/Link* also has a MathJFrame class, which is a subclass of the Swing JFrame class, and it also has these three special properties. Programmers who want to create interfaces with Swing components instead of AWT ones can use MathJFrame as their top-level window class.

Modeless Windows: Sharing the Kernel with Java

The previous subsection demonstrated how to write *J/Link* programs that display Java windows and then how to use the DoModal function to cause the kernel to wait until the window is closed. During the time that DoModal is running, the kernel is able to receive and process requests for computations that originate from the Java side. The word "modal" is used in this context to refer to the fact that the kernel is busy servicing the Java link, and thus the notebook front end cannot use the kernel until DoModal returns.

This arrangement works fine for many types of Java windows, and it is required for those that return a result to *Mathematica*, because the kernel cannot sensibly proceed until the window is dismissed. Unfortunately, it is too restrictive for a large class of user interface elements. Consider trying to duplicate the general concept of a front end palette window in Java. You want to have a window of buttons that, when clicked, cause some computation to occur in *Mathematica*. Like a front end palette window, you want this window to be created and remain visible and active indefinitely. It would not be of much use if every time you wanted to click one of the buttons you had first to execute `DoModal[]` (and you would also have to arrange for each button to call `EndModal[]` as part of the computation it triggers). You want to be able to go back and forth between notebook windows in the front end and our Java window without needing manually to switch the kernel into and out of some special state each time.

What is needed is a way for the kernel to automatically pay attention to input arriving from the Java link in addition to the notebook front end link. What you really have here is two front ends vying for the kernel's attention. *J/Link* solves this problem by introducing a simple way in which the kernel can be put into a state where it is simultaneously listening for input on any number of links. The function that accomplishes this is `ShareKernel`.

Important Note: In *Mathematica* 5.1 and later, the kernel is always shared with Java. This means that the functions `ShareKernel` and `UnshareKernel` are not necessary and, in fact, do nothing at all. If you are writing program that only need to run in *Mathematica* 5.1 and later, you never need to call `ShareKernel` or `UnshareKernel` (`ShareFrontEnd` and `UnshareFrontEnd` are still useful, however). If your programs need to work on all versions of *Mathematica*, then you will need to use `ShareKernel` and `UnshareKernel` as described next.

`ShareKernel[]`	begin sharing the kernel with Java
`ShareKernel[`*link*`]`	begin sharing the kernel with *link*
`UnshareKernel[`*id*`]`	unregisters the request for sharing (that is, the call to `ShareKernel`) that returned *id*; kernel sharing will not be turned off unless no other requests are outstanding
`UnshareKernel[`*link*`]`	end sharing of the kernel with *link*
`UnshareKernel[]`	end sharing of the kernel with Java
`KernelSharedQ[]`	`True` if the kernel is currently being shared; `False` otherwise
`SharingLinks[]`	a list of the links currently sharing the kernel

Sharing the kernel.

ShareKernel takes a LinkObject as an argument and initiates sharing of the kernel between that link and the current $ParentLink (typically, the notebook front end). If you call ShareKernel with no arguments, it assumes you mean the link to Java. Most users will call it with no arguments.

```
In[]:= ShareKernel[];

In[]:= 2 + 2
(sharing)Out[]= 4
```

Note that while the kernel is being shared, the input prompt has "(sharing)" prepended to it. The string that is prepended is specified by the SharingPrompt option to ShareKernel.

Sharing is transparent to the user. Other than the changed input prompt, there is nothing to suggest that anything different is going on. Input sent from either the front end or a Java program to the kernel will be evaluated and the result sent back to the program that sent the input. Each link is the kernel's $ParentLink during the time that the kernel is computing input that arrived from that link. In other words, ShareKernel takes care of shuffling the $ParentLink value back and forth between links as input arrives on each.

It is safe to call ShareKernel if the kernel is already being shared. This means that programs you write can call it without your having to worry that a user might already have initiated sharing. When you are finished with the need to share the kernel with Java, you can call UnshareKernel. This restores the kernel to its normal mode of operation, paying attention only to the front end.

```
In[]:= UnshareKernel[]
```

When called with no arguments, UnshareKernel shuts down sharing. This is not a desirable thing in most cases, because it might be that some other Java-based program is running that requires sharing. If you are writing code for others to use, you certainly cannot shut down sharing on your users just because your code is done with it. To solve this problem, ShareKernel returns a token (it is just an integer, but you should not be concerned with its representation) that reflects a request for sharing functionality. In other words, calling ShareKernel registers a request for sharing, turns it on if it is not on already, and returns a token that represents that particular request. When you call UnshareKernel, you pass it the token to "unregister" that particular request for sharing. Only if there are no other outstanding requests will sharing actually be turned off.

A quirk of ShareKernel is that you cannot call ShareKernel and UnshareKernel in the same cell. Doing so will cause the kernel to hang. Of course, there is no reason to ever do this, as kernel sharing is only relevant when it spans multiple evaluations (more precisely, the evaluation of multiple cells). There would be no point to turning sharing on and off within the scope of a single computation.

An example of a nontrivial user interface that uses ShareKernel is presented in "Real-Time Algebra: A Mini-Application".

Sharing the Front End

One goal of *J/Link* was to have Java user interface elements be as close as possible to first-class citizens of the notebook front end environment, in the way that notebooks and palettes are. The ability to share the kernel mimics one important aspect of this citizenship, hiding the fact that the Java runtime is a separate program and the kernel is normally only waiting for input from the front end.

There is one more important thing that palettes can do that would be nice to do from Java, and that is interact with the front end. You can create a palette button that, when clicked, evaluates the code Print["hello"]. You can do this easily with *J/Link* also, but with one big difference: when you click the palette button, hello appears in the active notebook, but when you click the Java button, the "hello" gets sent back to the Java program (which is, after all, the kernel's $ParentLink at that moment). Even if you persuaded the kernel to write the TextPacket that contains "hello" to the front end link instead of the Java link, nothing useful would happen because the front end is not paying attention to the kernel link when the front end is not waiting for the result of a computation. Poking some output at the front end while it is idle simply will not work.

J/Link provides the ShareFrontEnd function as the solution to this problem. ShareFrontEnd[] causes Print output and graphics generated by a Java user-interface element to appear in the front end. It also lets the Java side call *Mathematica* functions that manipulate elements of notebooks and have them work properly in the front end (for example, NotebookRead, NotebookWrite, SelectionEvaluate, and so on). While sharing is on, the front end behaves normally, and you can continue to use it for editing, calculations, or whatever. The sharing is transparent.

`ShareFrontEnd[]`	begin sharing the front end with Java
`UnshareFrontEnd[`*id*`]`	unregisters the request for sharing (that is, the call to `ShareFrontEnd`) that returned *id*; front end sharing will not be turned off unless no other requests are outstanding
`UnshareFrontEnd[]`	end sharing of the front end with Java
`FrontEndSharedQ[]`	`True` if the front end is currently being shared with Java; `False` otherwise

Sharing the notebook front end.

`ShareFrontEnd` currently does not work with a remote kernel; the same machine must be running the kernel and the front end.

`ShareFrontEnd` is as close as you currently can come to having Java user interfaces hosted directly by the notebook front end itself, as if they were special types of notebook windows. This type of tight integration might be possible in the future.

Note that `Print` output, graphics, and messages generated by a *modal* Java window will appear in the front end without needing to call `ShareFrontEnd`. This is because `$ParentLink` remains the front end link during `DoModal` (these "side effect" packets always get sent to `$ParentLink`), and also because the front end is able to handle various packets arriving from the kernel because the front end *is* in the middle of a computation—it is waiting for the result of the code that called `DoModal`. `ShareFrontEnd` is a way to restore a feature that was lost when you gained the ability to create *modeless* interfaces via `ShareKernel`. That is how to think of `ShareFrontEnd`—as a step beyond `ShareKernel` that allows side effect output generated by computations triggered in Java to appear in the notebook front end. `ShareFrontEnd` is particularly useful when developing code that needs to use `ShareKernel`, even if the code does not need the extra functionality of `ShareFrontEnd`. This is because *Mathematica* error messages generated by computations triggered by Java events get lost with `ShareKernel`. The messages will show up in the front end if front end sharing is turned on.

When you are done with the need to share the front end, call `UnshareFrontEnd`. Like the `ShareKernel`/`UnshareKernel` pair of functions, `ShareFrontEnd` returns a token that you should pass to `UnshareFrontEnd` to unregister the request for front end sharing. Only when all calls to `ShareFrontEnd` have been unregistered by calls to `UnshareFrontEnd` will front end sharing be turned off. You can force front end sharing to be shut down immediately by calling

UnshareFrontEnd with no arguments, but although this is convenient when you are developing code of your own, it should never be called in code that is intended for others to use. Just because your code is done with front end sharing does not mean that your users are done with it. Instead, save the token returned from ShareFrontEnd and pass it to UnshareFrontEnd.

ShareFrontEnd requires that the kernel be shared, so it calls ShareKernel internally. Calling UnshareKernel with no arguments forces kernel sharing to stop immediately, and this turns off front end sharing as well. Thus, you can use UnshareKernel[] as a quick shortcut to immediately shut down all sharing.

An example of some simple palette-type buttons that use ShareFrontEnd is presented in "Sharing the Front End: Palette-Type Buttons".

An important use for ShareFrontEnd is to allow a popup Java user interface to display graphics containing typeset expressions. When the kernel is asked to produce a graphic containing typeset expressions, say a plot with PlotLabel -> Sqrt[z], it crunches out PostScript for the plot itself, but when it comes time to produce PostScript for the typeset label, it cannot do this. Instead, it sends a special request back to the front end, asking it for the PostScript representation. Because dealing with typeset expressions is a skill possessed only by the notebook front end, when any other interface is driving the kernel, the interface must be careful to instruct the kernel to not attempt to typeset anything in a graphic (ShareKernel handles this automatically for you). This works fine, but you lose the ability to get pictures of typeset expressions in your Java interface.

ShareFrontEnd does two things to overcome this limitation: it fools the kernel into thinking that the Java runtime is a notebook front end and, therefore, capable of handling the special "convert to PostScript" requests; and it gives Java the ability to make good on this promise by forwarding the requests to the front end. "GraphicsDlg: Graphics and Typeset Output in a Window" describes an example of a Java dialog box that displays typeset expressions using ShareFrontEnd.

Summary of Modal and Modeless Operation

The previous discussion of modal and modeless operation, `ShareKernel`, and `ShareFrontEnd` may have seemed complex. In fact, the principles and uses of these techniques are simple. This will become clear upon seeing some more examples. Many of the example programs in "Example Programs" use `ShareKernel` or `ShareFrontEnd`. The important thing is to understand the capabilities they provide so that you can begin to see how to use them in your own programs.

If you want your user-interface element (typically a window) to tie up the kernel until the user dismisses it, then you will use the `setModal`/`DoModal`/`EndModal` suite. Because the internal workings of the modal state are simpler than the modeless state, you should use this style unless your program needs the features of a modeless window. You will always want to use this type of window if you need to return a result to a running *Mathematica* program, such as if you are creating a dialog box into which the user will enter values and then click **OK**. "A Simple Modal Input Dialog" gives an example of this type of dialog.

If you want your window to remain visible and active while the user returns to work in the front end, you must run your window in a "modeless" fashion. This requires calling `ShareKernel` to put the kernel into a state where it is simultaneously receptive to input arriving from either the notebook front end or Java. At this point the kernel is dividing its attention between two independent and essentially equivalent front ends. One drawback (or feature, depending on your point of view) of this state is that all side effect output like `Print` output, messages, or plots triggered by Java code is sent to Java instead of the front end (and the standard Java `MathListener` classes just throw all this output away). Thus, you could not create a button that prints something in a notebook window when it is clicked, like you can with a palette button in the front end. If you want to give your Java program the ability to interact with the front end the way that notebook and palette windows themselves can, you must instead use `ShareFrontEnd`, which you can think of as an extension to `ShareKernel`.

A very common mistake is to create a Java window, wire up a `MathListener` class that calls back to *Mathematica* on some event, and then trigger the event before you have called `DoModal` or `ShareKernel`. This will cause the Java user interface thread to hang. A symptom that the UI thread is hanging is that the controls in your Java window are visually unresponsive (for example, buttons will not appear to depress when you click them). If you do inadvertently get into this state, you can just call `ShareKernel` to allow the queued-up call(s) from Java to proceed.

"Manual" Interfaces: The ServiceJava Function

In addition to the modal and modeless types of interfaces just discussed, there is another type that in some ways is intermediate. Consider the following scenario. You want to create a *Mathematica* program that puts up a Java window and displays something in it that changes over the course of the program. So far, this sounds like an example of a "non-interactive" interface, which was discussed way back at the beginning of this section, the progress bar example being a classic case. Now, though, you want to add some interactivity to the window, meaning that you want user actions in the window to trigger calls into *Mathematica*. Keeping with the progress bar example, say you want to add an **Abort** button that stops the program. How do you manage to get the kernel's attention directed at the Java side so that Java events can trigger calls to *Mathematica*?

The modal type of interface will not work, because in the modal state the kernel is executing `DoModal`, not your computation—the kernel is doing nothing but paying attention to Java. The modeless type of interface will not work either, because the modeless technique causes the kernel to pay attention to the front end and Java alternately, letting each perform a full computation in turn. There is no sharing within the context of a single computation.

The obvious answer is the there needs to be a function that allows the kernel to service a single computation arriving from Java, if there is one waiting. That function is `ServiceJava`. Calling `ServiceJava` in a program will cause the kernel to accept one request for a computation from the Java side. It performs the computation and then returns control to your program. If there is no request waiting, `ServiceJava` returns immediately.

Here is some pseudocode showing the structure of a program that displays a progress bar with an **Abort** button and periodically calls `ServiceJava` to handle user clicks on that button, stopping the computation if requested.

```
    ... create progress bar ...
    progressBar@addActionListener[
        JavaNew["com.wolfram.jlink.MathActionListener", "(userCancelled =
True)&"]
    ];
    JavaShow[progressBar];
    While[i < 100 && !userCancelled,
        ... compute one iteration ...
        ... update progress bar ...
        ServiceJava[];
        i++
    ];
    ... destroy progress bar ...
```

You might recognize that `ServiceJava` is closely related to `DoModal`, and although this is not the actual implementation, you can think of `DoModal` as being written in terms of `ServiceJava` as follows:

```
(* Not the actual implementation of DoModal, but the principle is correct.
*)
DoModal[] :=
    While[!endModal,
        ServiceJava[]
    ]
```

Seen in this way, `DoModal` is a special case of the use of `ServiceJava`, where *Mathematica* is doing nothing but servicing requests from Java. Sometimes you need something else to be going on in *Mathematica*, but still need to be able to handle requests arriving from Java. That is when you call `ServiceJava` yourself. Like `DoModal`, there is no shifting of `$ParentLink` when `ServiceJava` is called. Thus, side-effect output like graphics, messages, and `Print` output triggered by Java computations appear in the notebook, just as if they were hard-coded into the *Mathematica* program that called `ServiceJava`.

The BouncingBalls example program presented in "BouncingBalls: Drawing in a Window" uses `ServiceJava`.

Using a GUI Builder

The preceding discussion on modal and modeless interfaces featured examples that were created entirely with *Mathematica* code. For complex user interfaces, you might find it more convenient to lay out your windows and wire up events with a drag-and-drop GUI builder like the

ones present in most commercial Java development environments. You are free to write as much or as little of the code for your interface in native Java. If you want events in your GUI to trigger calls into *Mathematica*, then you can use any of the `MathListener` classes from Java code just as they are used from *Mathematica* code. Alternatively, you could write your own Java code that calls into *Mathematica* at appropriate times. See the section "Writing Your Own Installable Java Classes" for information about how to write Java code that calls back into *Mathematica*. "GraphicsDlg: Graphics and Typeset Output in a Window" gives a simple example of a dialog box that was created with a GUI builder and is then invoked and controlled by *Mathematica* code.

Drawing and Displaying Mathematica Images in Java Windows

The MathCanvas and MathGraphicsJPanel classes

J/Link makes it easy to draw into Java windows from *Mathematica*, and also display *Mathematica* graphics and typeset expressions. The `MathCanvas` and `MathGraphicsJPanel` classes are provided for this purpose. You can use these classes in pure Java programs that use the *Mathematica* kernel, as described in "Writing Java Programs that use *Mathematica*", but it is also handy for Java windows that are created and scripted from *Mathematica*. Note that the `MathGraphicsJPanel` class is new in *J/Link* 2.0.

`MathCanvas` is a subclass of the AWT `Canvas` class, and `MathGraphicsJPanel` is a subclass of the Swing `JPanel` class. In terms of their special added *Mathematica* graphics capabilities, they are identical. These classes provide two ways to supply the image to be displayed. The first way is by providing a fragment of *Mathematica* code whose output will be displayed. The output can either be a graphics object, or a nongraphics expression that will be typeset. This makes it trivial to display *Mathematica* graphics or typeset expressions in a Java window. The second way to control the display is to provide a Java `Image` object that will be painted. This `Image` will typically be created by *Mathematica* code, such as code that creates a bitmap out of raw *Mathematica* data, or code that draws something using calls to Java's graphics routines.

Because `MathCanvas` and `MathGraphicsJPanel` are Java classes and can be used from Java programs as well as *Mathematica* programs, there is full JavaDoc format documentation for them in the JLink/Documentation/JavaDoc directory. You can browse that documentation for more details.

Showing *Mathematica* Graphics and Typeset Expressions

Here is a simple example of displaying a window that shows a *Mathematica* plot. This example uses `MathCanvas`, but the relevant parts would look the same if you used `MathGraphicsJPanel`. You will be using this window throughout this section, so do not close it if you are evaluating the code as you read this section.

```
frame = JavaNew["com.wolfram.jlink.MathFrame"];
frame@setLayout[JavaNew["java.awt.BorderLayout"]];
mathCanvas = JavaNew["com.wolfram.jlink.MathCanvas"];
frame@add["Center", mathCanvas];
frame@setSize[400, 400];
frame@layout[];
mathCanvas@setMathCommand["Plot[x, {x,0,1}]"];
JavaShow[frame];
```

As you can see, it is as simple as calling the canvas' `setMathCommand()` method. The argument to `setMathCommand()` is a string giving the code to be evaluated. This code must *return* a graphics expression, not just cause one to be produced. For example, `setMathCommand["Plot[x,{x,0,1}];"]` will not work because the trailing semicolon causes the expression to evaluate to `Null`. The image is automatically rendered at the correct size, and centered in the canvas if the actual image size produced by *Mathematica* does not completely fill the requested area (as is often the case with typeset output).

Calling `setMathCommand()` again resets the image.

```
mathCanvas@setMathCommand["Plot3D[Sin[x Cos[y]], {x,0,2Pi}, {y,0,2Pi}]"];
```

If the plotting command depends on variables in your *Mathematica* session, you can call `recompute()` to cause the graphic to be recomputed and rendered. For example, this displays a slow animation in the window.

```
n = 1.0;
mathCanvas@setMathCommand["Plot3D[Sin[n x Cos[y]], {x,0,2Pi}, {y,0,2Pi}]"];
Do[n += 0.1; mathCanvas@recompute[]; Pause[1], {10}]
```

Because you supply the expression as a string, remember to escape any quote marks inside the string with a backslash.

```
mathCanvas@setMathCommand["Plot[x, {x,0,1}, PlotLabel->\"This is a plot\"]"];
```

A `MathCanvas` can also display typeset expressions. The default behavior of `MathCanvas` is to expect that the expression supplied in `setMathCommand()` will evaluate to a graphics object, which should be rendered. To get it to instead typeset the return value, call the `setImageType()` method, supplying the constant `TYPESET`.

```
mathCanvas@setImageType[MathCanvas`TYPESET];
mathCanvas@setMathCommand["Integrate[Sqrt[x] Sqrt[1+x], x]"];
```

To switch back to displaying graphics, call mathCanvas@setImageType[MathCanvas`GRAPHICS]. The default format for typeset output is StandardForm. To switch to TraditionalForm, use the setUsesTraditionalForm() method. You call recompute() here because changing the output type does not force the image to be redrawn.

```
mathCanvas@setUsesTraditionalForm[True];
mathCanvas@recompute[];
```

Graphics are rendered using *Mathematica*'s Display command, which is fast and does not require the notebook front end to be running. For higher quality, though, particularly for 3D graphics, an alternative method is available that uses the front end for rendering services. You can switch to using this technique by calling the setUsesFE() method.

```
(* First, change back to graphics mode from typeset mode. *)
mathCanvas@setImageType[MathCanvas`GRAPHICS];

mathCanvas@setUsesFE[True];
mathCanvas@setMathCommand["Plot3D[Sin[x Cos[y]], {x,0,2Pi}, {y,0,2Pi}]"];
```

You might want to compare the resulting plot with setUsesFE[True] and setUsesFE[False].

An important point about using the front end for rendering is that when the computation to produce the image is performed, the front end must be in a state where it is receptive to requests for services from the kernel. There are two times when this is the case: either a cell in the front end is currently evaluating (as will be the case when you are calling setMathCommand() or recompute() from a *Mathematica* program), or ShareFrontEnd has been called. Looking at it from the other direction, the only time it will not work is if ShareKernel is in use, but not ShareFrontEnd, and the computation is triggered by an event in Java. The rule is that if you want to involve the front end for rendering, and you want to call setMathCommand() or recompute() from Java in response to a user action in a modeless interface, you need to use ShareFrontEnd; ShareKernel is not enough. Modal and modeless interfaces and ShareFrontEnd are discussed in the section "Creating Windows and Other User Interface Elements".

Drawing Using Java's Graphics Functions

You saw that the setMathCommand() method of the MathCanvas and MathGraphicsJPanel classes lets you supply a *Mathematica* expression whose output is to be displayed. You can also use a MathCanvas or MathGraphicsJPanel to display a Java Image by using the setImage() method instead of setMathCommand().

Now look at a simple example of drawing into a Java window from *Mathematica*. You will continue to use the same window and `MathCanvas` you have been working with. If this program used a `MathGraphicsJPanel` instead, the portions of the code related to drawing would look exactly the same. To draw into the `MathCanvas`, you create an offscreen image of the same dimensions, get a graphics context for drawing onto it, draw, and then use the `setImage()` method of `MathCanvas` to cause the offscreen image to be displayed. Drawing into an offscreen image and then blitting it to the screen is a standard technique for flicker-free drawing.

```
offscreen = mathCanvas@createImage[mathCanvas@getSize[]@width,
                                   mathCanvas@getSize[]@height];
g = offscreen@getGraphics[];
g@drawRect[100, 100, 200, 150];
mathCanvas@setImage[offscreen];
```

Programs that want to draw manually into a Java window from *Mathematica* will generally all have this same structure. It takes just a few more lines of code to turn our `MathCanvas` into a scribble program. Here is the complete program (this code is also provided as the file Scribble.nb in the JLink/Examples/Part1 directory).

```
Scribble[] :=
    JavaBlock[
        Module[{frame, mathCanvas, offscreen, g, mml, pts},
            frame = JavaNew["com.wolfram.jlink.MathFrame"];
            frame@setLayout[JavaNew["java.awt.BorderLayout"]];
            mathCanvas = JavaNew["com.wolfram.jlink.MathCanvas"];
            frame@add["Center", mathCanvas];
            frame@setSize[400, 400];
            frame@layout[];
            JavaShow[frame];
            (* Now create the offscreen image and the graphics context
               for drawing into it.
            *)
            offscreen = mathCanvas@createImage[mathCanvas@getSize[]@width,
                                               mathCanvas@getSize[]@height];
            g = offscreen@getGraphics[];
            (* Now create the MathMouseMotionListener that will do the drawing
               and set its mouseDragged event handler callback.
            *)
            mml = JavaNew["com.wolfram.jlink.MathMouseMotionListener"];
            mml@setHandler["mouseDragged", "mouseDraggedFunc"];
            mathCanvas@addMouseMotionListener[mml];
            mouseDraggedFunc[_, x_, y_, _] :=
                (g@drawLine[pts[[-1, 1]], pts[[-1, 2]], x, y];
                 mathCanvas@setImage[offscreen];
                 mathCanvas@repaintNow[];
                 AppendTo[pts, {x,y}];);
            (* Initialize the pts list and run the program modally. *)
            pts = {{0,0}};
            frame@setModal[];
            DoModal[];
            pts
        ]
    ]
```

Run the program, then click and drag the mouse to draw in the window. Close the window to end the program and the `Scribble` function will return the list of points drawn.

```
pts = Scribble[];
```

If you examine the list of points returned, you will see that they are based on Java's coordinate system, which has (0, 0) in the upper left. If you want to plot the points in a *Mathematica* graphic, you have to invert the *y* values. This is demonstrated in the Scribble.nb example notebook.

There is one new `MathCanvas` method demonstrated in this program, `repaintNow()`. In a computation-intensive program like this, where events are being fired on the user interface thread very quickly, and the handlers for these events take a nontrivial amount of time to execute, Java will sometimes delay repainting the window. The drawing becomes very chunky, with no visual effect for a while and then suddenly all the lines drawn in the last few seconds will appear. Even calling the standard `repaint()` method after every new point will not ensure that the window is updated in a timely manner. To solve this problem, the `repaintNow()` method is provided, which forces an immediate redraw of the canvas. If your program relies on smooth visual feedback from user events that fire rapidly, you should call `repaintNow()` also, even if it does not seem necessary on your system. There can be very significant differences between different platforms and different Java runtimes on the responsiveness of the screen updating mechanism.

The ability to draw in response to events in a `MathCanvas` or `MathGraphicsJPanel` opens up the possibility for some impressive interactive demonstrations, tutorials, and so on. Two of the larger example programs provided draw into a `MathCanvas` from *Mathematica:* BouncingBalls (in the section "BouncingBalls: Drawing in a Window") and Spirograph (in the section "Spirograph").

Bitmaps

You have seen how to draw into a `MathCanvas` or `MathGraphicsJPanel` by using an offscreen image. Another type of image that you can create with *Mathematica* code and display using `setImage()` is a bitmap. In this example you will create an indexed-color bitmap out of *Mathematica* data and display it. You will use an 8-bit color table, meaning that every data point in the image will be treated as an index into a 256-element list of colors. You could use a larger color table if desired.

You closed the frame window in the Scribble example, so you must first create a new frame and canvas for the bitmap.

```
frame = JavaNew["com.wolfram.jlink.MathFrame"];
frame@setLayout[JavaNew["java.awt.BorderLayout"]];
mathCanvas = JavaNew["com.wolfram.jlink.MathCanvas"];
frame@add["Center", mathCanvas];
frame@setSize[450, 450];
frame@layout[];
JavaShow[frame];
```

Here is the color table. It is an array of {r,g,b} triplets, with each color component being in the range 0..255. In this example, colors with low indices are mostly blue, and ones with high indices are mostly red.

```
colors = Table[{i, 0, 255 - i}, {i, 0, 255}];
```

The data is a 400×400 matrix of integers in the range 0..255 (because they are indices into the 256-element color table). In a real application, this data might be read from a file or computed in some more sophisticated way. If the range of numbers in the data did not span 0..255, you would have to scale it into that range, or a larger range if you wanted to use a deeper color table.

```
data = Table[Round[255 (0.5 + Sin[x]Cos[y]/2)],
            {x, Pi/100., 4Pi, Pi/100.}, {y, Pi/100., 4Pi, Pi/100.}];
```

Here you create the Java objects that represent the color model and bitmap. You can read the standard Java documentation on these classes for more information.

```
colorModel = JavaNew["java.awt.image.IndexColorModel", 8, 256,
                              Flatten[colors], 0, False];
bitmap = JavaNew["java.awt.image.MemoryImageSource", 400, 400,
                            colorModel, Flatten[data], 0, 400];
```

Now create an Image out of the bitmap and display it.

```
image = frame@getToolkit[]@createImage[bitmap];
mathCanvas@setImage[image];
```

The Java Console Window

J/Link provides a convenient means to display the Java "console" window. Any output written to the standard System.out and System.err streams will be directed to this window. If you are calling Java code that writes diagnostic information to System.out or System.err, then you can see this output while your program runs. Like most *J/Link* features, the console window can be used easily from either *Mathematica* or Java programs (its use from Java code is described in "Writing Java Programs that use *Mathematica*"). To use it from *Mathematica*, call the ShowJavaConsole function.

ShowJavaConsole[]	display the Java console window and begin capturing output written to System.out and System.err
ShowJavaConsole["*stream*"]	display the Java console window and begin capturing output written to the specified stream, which should be "*stdout*" for System.out or "*stderr*" for System.err
ShowJavaConsole[None]	stop all capturing of output

Showing the console window.

```
ShowJavaConsole[]
```

```
«JavaObject[com.wolfram.jlink.ui.ConsoleWindow] »
```

Capturing of output only begins when you call ShowJavaConsole, so when the window first appears it will not have any content that might have been previously written to System.out or System.err. You will also note that the *J/Link* console window displays version information about the *J/Link* Java component and the Java runtime itself. Calling ShowJavaConsole when the window is already open will cause it to come to the foreground.

To demonstrate, you can write some output from *Mathematica*. If you executed the ShowJavaConsole[] given earlier, then you will see "Hello from Java" printed in the window.

```
LoadJavaClass["java.lang.System"];
System`out@println["Hello from Java"]
```

Although it is convenient to demonstrate writing to the window using *Mathematica* code like this, this is typically done from Java code instead. Actually, there is one common circumstance where it is quite useful to use the Java console window for diagnostic output written from *Mathematica* code. This is the case where you have a "modeless" Java user interface (as described in the section "Creating Windows and Other User Interface Elements") and you have not used the ShareFrontEnd function. Recall that in this circumstance, output from calls to Print in *Mathematica* will not appear in the notebook front end. If you write to System.out instead, as in the example, then you will always be able to see the output. You might want to do this in other circumstances just to avoid cluttering up your notebook with debugging output.

Using JavaBeans

JavaBeans is Java's component architecture. Beans are reusable components that can be manipulated visually in a builder tool. At the code level, a Bean is essentially just a normal Java class that conforms to a particular design pattern with respect to how its methods are named and how it supports events and persistence.

JavaBeans has not been mentioned up to this point because there really is not anything special to be said. Beans are just Java classes, and they can be used and called like any other classes. It is probably the case that many Java classes you use from *Mathematica* will be Beans, whether they advertise themselves to be or not. This is especially true for user interface components.

Beans are typically designed to be used in a visual builder tool, where the programmer is not writing code and calling named methods directly. Instead, a Bean exposes "properties" to the builder tool, which can be examined and set using a property editor window. In a typical simple example, a Bean might have methods named `setColor` and `getColor`, and by virtue of this it would be said to have a property named "color". A property editor would have a line showing the name "color" and an edit field where you could type in a color. It might even have a fancy editor that puts up a color picker window to let you visually select a desired color.

For the purposes of a visual builder tool or other type of automated manipulation, beans try to hide the low-level details of actual method names. If you want to call methods in a Bean class from *Mathematica* code, you call them by name in the usual way, without any consideration of the "Bean-ness" of the class.

Note that it would be quite possible to add *Mathematica* functions to *J/Link* that would provide explicit support for Bean properties. For example, a function `BeanSetProperty` could be written that would take a Bean object, a property name as a string, and the value to set the property to. Then, instead of writing what is currently required:

```
bean@setColor[Color`green]
```

you could write:

```
BeanSetProperty[bean, "color", Color`green]
```

The `BeanSetProperty` function lets you write code that manipulates nebulous things called properties instead of calling specific methods in the Bean class. If you do not see any particular advantage in the `BeanSetProperty` style, then you know why there is no special Bean support along these lines in *J/Link*. The advantages of working with properties versus directly calling methods accrues only when you are using a builder tool and not actually writing code by hand.

If you are interested, here are simplistic implementations of BeanSetProperty and BeanGetProperty:

```
BeanSetProperty[bean_?JavaObjectQ, propName_String, val_] :=
    Module[{methName = "set" <> ToUpperCase[StringTake[propName, 1]] <>
            StringDrop[propName, 1]},
        Through[(bean @@ ToHeldExpression[methName])[val]]
    ]

BeanGetProperty[bean_?JavaObjectQ, propName_String] :=
    Module[{methName = "get" <> ToUpperCase[StringTake[propName, 1]] <>
            StringDrop[propName, 1]},
        Through[(bean @@ ToHeldExpression[methName])[]]
    ]
```

To make use of events that a JavaBean fires, you can use one of the standard MathListener classes, as described in the section "Creating Windows and Other User Interface Elements". JavaBeans often fire PropertyChangeEvents, and you can arrange for *Mathematica* code to be executed in response to these events by using a MathPropertyChangeListener or a MathVetoableChangeListener.

Hosting Applets

J/Link gives you the ability to run most applets in their own window directly from *Mathematica*. Although this may seem immensely useful, given the vast number of applets that have been created, most applets do not export any useful public methods. They are generally standalone pieces of functionality, and thus they benefit little from the scriptability that *J/Link* provides. Still, there are many applets that may be useful to launch from a *Mathematica* program.

Note that this section is not about writing applets that use the *Mathematica* kernel. That topic is covered in "Writing Applets".

AppletViewer["*applet class*"]	runs the named applet class in its own window. The default width and height are 300 pixels
AppletViewer["*applet class*", *params*]	runs the named applet class in its own window, supplying it the given parameters, which is a list of "name=value" specifications like those used in an HTML page

Running applets.

J/Link includes an AppletViewer function for running applets. This function takes care of all the steps of creating the applet instance, providing a frame window to hold it, and starting it running. The first argument to AppletViewer is the fully qualified name of the applet class. The

second argument is an optional list of parameters in "name=value" format, corresponding to the parameters supplied to an applet in an HTML page that hosts it. For example, if the <applet> tag in a web page that hosts an applet looks like this:

```
<applet code="SomeApplet.class" width=400 height=300>
     <param name=foo value=bar>
</applet>
```

you would call `AppletViewer` like this:

```
AppletViewer["SomeApplet", {"width=400", "height=300", "foo=bar"}];
```

You will typically supply at least "WIDTH=" and "HEIGHT=" specifications to control the width and height of the applet window. If you do not specify these parameters, the default width and height are 300 pixels.

An excellent example of an applet that is useful to *Mathematica* users is LiveGraphics3D, written by Martin Kraus. LiveGraphics3D is an interactive viewer for *Mathematica* 3D graphics. It gives you the ability to rotate and zoom images, view them in stereo, and more. If you want to try the following example, you will need to get the LiveGraphics3D materials, available from http://wwwvis.informatik.uni-stuttgart.de/~kraus/LiveGraphics3D/. Make sure you put `live.jar` onto your `CLASSPATH` before trying that example, or use the `AddToClassPath` feature of *J/Link* to make it available.

First, load the `PolyhedronOperations`` package and create the graphic to display. The LiveGraphics3D documentation gives a more general-purpose function for turning a *Mathematica* graphics expression into appropriate input for the LiveGraphics3D applet but, for many examples, using `ToString`, `InputForm`, and `N` is sufficient.

```
<< PolyhedronOperations`
dodec = ToString[InputForm[
    N[Graphics3D[Stellate[Normal[PolyhedronData["Dodecahedron", "Faces"]]]]]]];
```

You specify the image to be displayed via the `INPUT` parameter, which takes a string giving the `InputForm` representation of the graphic.

```
AppletViewer["Live", {"INPUT=" <> dodec, "WIDTH=400", "HEIGHT=400"}];
```

The Live applet has a number of keyboard and mouse controls for manipulating the image. You can read about them in the LiveGraphics3D documentation. Try Alt+S to switch into a stereo view.

When you are done with an applet, just click the window's close box.

If the applet needs to refer to other files, you should be aware that AppletViewer sets the document base to be the directory specified by the "user.dir" Java system property. This will normally be *Mathematica*'s current directory (given by Directory[]) at the time that InstallJava was called.

Most applets expose no public methods useful for controlling from *Mathematica*, so there is nothing to do but start them up with AppletViewer and then let the user close the window when they are finished. The Live applet is an exception—it provides a full set of methods to allow the view point, spin, and so on to be modified by *Mathematica* code. These methods are in the Live class, so to call them you need an instance of the Live class. The way you used AppletViewer earlier does not give us any instance of the applet class. The construction and destruction of the applet instance was hidden within the internals of AppletViewer. You can also call AppletViewer with an instance of an applet class instead of just the class name. This lets you manage the lifetime of the applet instance.

```
applet = JavaNew["Live"];
AppletViewer[applet, {"INPUT=" <> dodec, "WIDTH=400", "HEIGHT=400"}];
```

Now you can call methods on the applet instance. See the LiveGraphics3D documentation for the full set of methods. This scriptability opens up lots of possibilities, such as programming "flyby" views of objects, or creating buttons that jump the image into certain orientations or spins.

```
applet@setMagnification[0.5];
```

When you are done, you call ReleaseJavaObject to release the applet instance. This can be done before or after the applet window is closed.

```
ReleaseJavaObject[applet]
```

Periodical Tasks

The section "Creating Windows and Other User Interface Elements" described the ShareKernel function and how it allows Java and the notebook front end to share the kernel's attention. A side benefit of this functionality is that it becomes easy to provide a means whereby users can schedule arbitrary *Mathematica* programs to run at periodical intervals during a session. Say you have a source that provides continuously updated financial data and you want to have some variables in *Mathematica* constantly reflect the current values. You have written a program that goes out and reads from the source to get the information, but you have to manually run this program all the time while you are working. A better solution would be to set up a periodical task that pulls the data from the source and sets the variables every 15 seconds.

`AddPeriodical[`*expr*`,`*secs*`]`	cause *expr* to be evaluated every *secs* seconds while the kernel is idle
`RemovePeriodical[`*id*`]`	stop scheduling of the periodical represented by *id*
`Periodical[`*id*`]`	return a list `{HoldForm[`*expr*`],` *secs*`}` showing the expression and time interval associated with the periodical represented by *id*
`Periodicals[]`	return a list of the *id* numbers of all currently scheduled periodicals
`SetPeriodicalInterval[`*id*`]`	reset the periodical interval for the periodical task represented by *id*
`$ThisPeriodical`	holds the *id* of the currently executing periodical task

Controlling periodical tasks.

You can set up such a task with the `AddPeriodical` function.

```
id = AddPeriodical[updateFinancialData[], 15];
```

`AddPeriodical` returns an integer ID number that you must use to identify the task—for example, when it comes time to stop scheduling it by calling `RemovePeriodical`. `AddPeriodical` relies on kernel sharing, so it calls `ShareKernel` if it has not already been called. There is no limit on the number of periodicals that can be established.

After scheduling that task, `updateFinancialData[]` will be executed every 15 seconds while the kernel is idle. Note that periodical tasks are run only when the kernel is not busy—they do not interrupt other evaluations. If the kernel is in the middle of another evaluation when the allotted 15 seconds elapses, the task will wait to be executed until immediately after the computation finishes. Any such delayed periodicals are guaranteed to be executed as soon as the kernel finishes with the current computation. They cannot be indefinitely delayed if the user is busy with numerous computations in the front end or in Java. The converse to these facts is also true—if a periodical is executing when the user evaluates a cell in the front end, the evaluation will not be able to start until all periodicals finish, but it is guaranteed to start immediately thereafter.

To remove a single periodical task, use `RemovePeriodical`, supplying the ID number of the periodical as the argument. To remove all periodical tasks, use `RemovePeriodical[Periodicals[]]`. Periodical tasks are all removed if you call `UnshareKernel[]` with no arguments, which turns off all kernel sharing. You would then need to use `AddPeriodical` again to reestablish periodical tasks.

You can reset the scheduling interval for a periodical task by calling `SetPeriodicalInterval`, which is new in *J/Link* 2.0. This line makes the financial data periodical execute every 10 seconds, instead of 15 as shown earlier.

```
SetPeriodicalInterval[id, 10]
```

Sometimes you might want to change the interval for a periodical task or remove it entirely from within the code of the task itself. `$ThisPeriodical` is a variable that holds the ID of the currently executing periodical task. It will only have a value during the execution of a periodical task. You use `$ThisPeriodical` from within your periodical task to obtain its ID so that you can call `RemovePeriodical` or `SetPeriodicalInterval`.

Periodical tasks do not necessarily have anything to do with Java, nor do they need to use Java. Technically, Java does not even need to be running. However, because Java is used by the internals of `ShareKernel` to yield the CPU, if Java is not running then setting a periodical task will cause the kernel to keep the CPU continuously busy. Periodical task functionality is included in *J/Link* because it is a simple extension to `ShareKernel`, and it does have some nice uses in association with Java.

A final note about periodical tasks is that they do not cause output to appear in the front end. Look at this attempt.

```
id = AddPeriodical[Print["hello"], 10];
```

The programmer expects to get `hello` printed in his notebook every 10 seconds, but nothing happens. During the time when periodicals are executed, `$ParentLink` is not assigned to the front end (or Java). Results or side effects like `Print` output, messages, or graphics vanish into the ether.

Before proceeding, clean up the periodical tasks you created.

```
RemovePeriodical[Periodicals[]];
```

Some Special Number Classes

Preamble

There is a set of special number-related classes in Java that *J/Link* maps to their *Mathematica* numeric representation. Like strings and arrays, objects of these number classes have an important property: although they are objects in Java, they have a meaningful "by value"

representation in *Mathematica*, so it is convenient for *J/Link* to automatically convert them to numbers as they are returned from Java to *Mathematica*, and back to objects as they are sent from *Mathematica* to Java.

These classes are the so-called "wrapper" classes that represent primitive types (`Byte`, `Integer`, `Long`, `Double`, and so on), `BigDecimal` and `BigInteger`, and any class used to represent complex numbers. The treatment of these classes is described in this section.

The "Wrapper" Classes: Integer, Float, Boolean, and Others

Java has a set of so-called "wrapper" classes that represent primitive types. These classes are `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `Boolean`. The wrapper classes hold single values of their respective primitive types, and are necessary to allow everything in Java to be represented as a subclass of Object. This lets various utility methods and data structures that deal with objects handle primitive types in a straightforward way. It is also necessary for Java's reflection capabilities.

If you have a Java method that returns one of these objects, it will arrive in *Mathematica* as an integer (for `Byte`, `Character`, `Short`, `Integer`, and `Long`), real number (for `Float` and `Double`), or the symbols `True` or `False` (for `Boolean`). Likewise, a Java method that takes one of these objects as an argument can be called from *Mathematica* with the appropriate raw *Mathematica* value. The same rules hold true for arrays of these objects, which are mapped to lists of values.

In the unlikely event that you want to defeat these automatic "pass by value" semantics, you can use the `ReturnAsJavaObject` and `JavaObjectToExpression` functions, discussed in "References and Values".

Complex Numbers

You have seen that Java number types (e.g., `byte`, `int`, `double`) are returned to *Mathematica* as integers and reals, and integers and reals are converted to the appropriate types when sent as arguments to Java. What about complex numbers? It would be nice to have a Java class representing complex numbers that mapped directly to *Mathematica*'s `Complex` type, so that automatic conversions would occur as they were passed back and forth between *Mathematica* and Java. Java does not have a standard class for complex numbers, so *J/Link* lets you name the class that you want to participate in this mapping.

SetComplexClass["*classname*"]	set the class to be mapped to complex numbers in *Mathematica*
GetComplexClass[]	return the class currently used for complex numbers

Setting the class for complex numbers.

You can use any class you like as long as it has the following properties:

1. A public constructor that takes two doubles (the real and imaginary parts, in that order)
2. Methods that return the real and imaginary parts, having the following signatures:

```
public double re();
public double im();
```

Say that you are doing some computations with complex numbers in Java, and you want to interact with these methods from *Mathematica*. You like to use the complex number class available from `netlib`. This class is named `ORG.netlib.math.complex.Complex` and is available at http://www.netlib.org/java/. You use the `SetComplexClass` function to specify the name of the class:

```
SetComplexClass["ORG.netlib.math.complex.Complex"];
```

Now any method or field that takes an argument of type `ORG.netlib.math.complex.Complex` will accept a *Mathematica* complex number, and any object of class `ORG.netlib.math.complex.Complex` returned from a method or field will automatically be converted into a complex number in *Mathematica*. The same holds true for arrays of complex numbers.

Note that you must call `SetComplexClass` before you load any classes that use complex numbers, not merely before you call any methods of the class.

BigInteger and BigDecimal

Java has standard classes for arbitrary-precision floating-point numbers and arbitrary-precision integers. These classes are `java.math.BigDecimal` and `java.math.BigInteger`, respectively. Because *Mathematica* effortlessly handles such "bignums," *J/Link* maps `BigInteger` to *Mathematica* integers and `BigDecimal` to *Mathematica* reals. What this means is that any Java method or field that takes, say, a `BigInteger` can be called from *Mathematica* by passing an integer. Likewise, any method or field that returns a `BigDecimal` will have the value returned to *Mathematica* as a real number.

Ragged Arrays

Java allows arrays that are deeper than one dimension to be "ragged," or non-rectangular, meaning that they do not have the same length at every position at the same level. For example, `{{1,2,3},{4,5},{6,7,8}}` is a ragged two-dimensional array. *J/Link* allows you to send and receive ragged arrays, but it is not the default behavior. The reason for this is simply efficiency—the *MathLink* library has functions that allow very efficient transfer of rectangular arrays of most primitive types (e.g., `byte`, `int`, `double`, and so on), whereas ragged ones have to be picked apart tediously with a series of individual calls to get every piece. This all happens deep inside *J/Link*, so you do not have to be concerned with the mechanics of array passing, but it has a huge impact on speed. To maximize speed, *J/Link* assumes that arrays of primitive types are rectangular. You can toggle back and forth between allowing and rejecting ragged arrays by calling the `AllowRaggedArrays` function with either `True` or `False`.

`AllowRaggedArrays[True]`	allow ragged (i.e., nonrectangular) arrays to be sent to Java

Ragged array support.

With `AllowRaggedArrays[True]`, sending of arrays deeper than one dimension is greatly slowed. Here is an example of array behavior and how it is affected. Assume the class `Testing` has the following method, which takes a two-dimensional array of ints and simply returns it:

```
public static int[][] intArrayIdentity(int[][] a) {
    return a;
}
```

Look what happens if you call it with a ragged array.

```
LoadClass["Testing"];
Testing`intArrayIdentity[{{1, 2, 3}, {4, 5}}]
```

```
Java::argxs1 :
  The static method Testing`intArrayIdentity was called with an incorrect
    number or type of arguments. The argument was {{1,2,3},{4,5}}.
$Failed
```

An error occurs because the *Mathematica* definition for the `Testing`intArrayIdentity()` function requires that its argument be a two-dimensional rectangular array of integers. The call never even gets out of *Mathematica*.

Here you turn on support for ragged arrays, and the call works. This requires modifications in both the *Mathematica*-side type checking on method arguments and the Java-side array-reading routines.

```
AllowRaggedArrays[True]
Testing`intArrayIdentity[{{1, 2, 3}, {4, 5}}]
{{1, 2, 3}, {4, 5}}
```

It is a good idea to turn off support for ragged arrays as soon as you no longer need it, since it slows arrays down so much.

```
AllowRaggedArrays[False]
```

Implementing a Java Interface with Mathematica Code

You have seen how *J/Link* lets you write programs that use existing Java classes. You have also seen how you can wire up the behavior of a Java user interface via callbacks to *Mathematica* via the `MathListener` classes. You can think of any of these `MathListener` classes, such as `MathActionListener`, as a class that "proxies" its behavior to arbitrary user-defined *Mathematica* code. It is as if you have a Java class that has its implementation written in *Mathematica*. This functionality is extremely useful because it greatly extends the set of programs you can write purely in *Mathematica*, without resorting to writing our own Java classes.

`ImplementJavaInterface["interfaceName",{"methName"->"mathFunc",...}]`
create an instance of a Java class that implements the named Java interface by calling back to *Mathematica* according to the given mappings of Java methods to *Mathematica* functions

Implementing a Java interface entirely in *Mathematica*.

It would be nice to be able to take this behavior and generalize it, so that you could take *any* Java interface and implement its methods via callbacks to *Mathematica* functions, and do it all without having to write any Java code. The `ImplementJavaInterface` function, new in *J/Link* 2.0, lets you do precisely that. This function is easier to understand with a concrete example. Say you are writing a *Mathematica* program that uses *J/Link* to display a Java window with a Swing menu, and you want to script the behavior of the menu in *Mathematica.* The Swing `JMenu` class fires events to registered `MenuListeners`, so what you need is a class that implements `MenuListener` by calling into *Mathematica*. A quick glance at the section on MathListen-

ers reveals that *J/Link* does not provide a `MathMenuListener` class for you. You could choose to write your own implementation of such a class, and in fact this would be very easy, even trivial, since you would make it a subclass of `MathListener` and inherit virtually all the functionality you would need. For the sake of this discussion, assume that you choose not to do that, perhaps because you do not know Java or you do not want to deal with all the extra steps required for that solution. Instead, you can use `ImplementJavaInterface` to create such a Java class with a single line of *Mathematica* code:

```
mathMenuListener =
    ImplementJavaInterface["javax.swing.event.MenuListener",
        {"menuSelected" -> "menuSelectedFunc",
         "menuCanceled" -> "menuCanceledFunc",
         "menuDeselected" -> "menuDeselectedFunc"}
    ];
myMenu@addMenuListener[mathMenuListener];

...

(* Later, define the three Mathematica event-handler functions: *)
menuSelectedFunc[menuEvent_] := ...

menuCanceledFunc[menuEvent_] := ...

menuDeselectedFunc[menuEvent_] := ...
```

The first argument to `ImplementJavaInterface` is the Java interface or list of interfaces you want to implement. The second argument is a list of rules that associate the name of a Java method from one of the interfaces with the name of a *Mathematica* function to call to implement that method. The *Mathematica* function will be called with the same arguments that the Java method takes. What `ImplementJavaInterface` returns is a Java object of a newly created class that implements the named interface(s). You use it just like any `JavaObject` obtained by calling `JavaNew` or through any other means. It is just as if you had written your own Java class that implemented the named interface by calling the associated *Mathematica* functions, and then called `JavaNew` to create an instance of that class.

It is not necessary to associate every method in the interface with a *Mathematica* function. Any Java methods you leave out of your list of mappings will be given a default Java implementation that returns `null`. If this is not an appropriate return value for the method (e.g., if the method returns an `int`) and the method gets called at some point an exception will be thrown. Generally, this exception will propagate to the top of the Java call stack and be ignored, but it is recommended that you implement all the methods in the Java interface.

The ImplementJavaInterface function makes use of the "dynamic proxy" capability introduced in Java 1.3. It will not work in Java versions earlier than 1.3. All Java runtimes bundled with *Mathematica* 4.2 and later are at Version 1.3 or later. If you have *Mathematica* 4.0 or 4.1, the ImplementJavaInterface function is another reason to make sure you have an up-to-date Java runtime for your system.

At first glance, the ImplementJavaInterface function might seem to give us the capability to write arbitrary Java classes in the *Mathematica* language, and to some extent that is true. One important thing you cannot do is extend, or subclass, an existing Java class. You also cannot add methods that do not exist in the interface you are implementing. Event-handler classes are a good example of the type of classes for which this facility is useful. You might think that the MathListener classes are rendered obsolete by ImplementJavaInterface, and it is true that their functionality can be duplicated with it. The MathListener classes are still useful for Java versions earlier than 1.3, but most importantly they are useful for writing pure Java programs that call *Mathematica*. Using a class implemented in *Mathematica* via ImplementJavaInterface in a Java program that calls *Mathematica* would be possible, but quite cumbersome. If you want a dual-purpose class that is as easy to use from *Mathematica* as from Java, you should write your own subclass of MathListener. One *poor* reason for choosing to use ImplementJavaInterface instead of writing a custom Java class is that you are worried about complicating your application by requiring it to include its own Java classes in addition to *Mathematica* code. As explained in "Deploying Applications That Use *J/Link*", it is extremely easy to include supporting Java classes in your application. Your users will not require any extra installation steps nor will they need to modify the Java class path.

Writing Your Own Installable Java Classes

Preamble

The previous sections have shown how to load and use existing Java classes. This gives *Mathematica* programmers immediate access to the entire universe of Java classes. Sometimes, though, existing Java classes are not enough, and you need to write your own.

J/Link essentially obliterates the boundary between Java and *Mathematica*, letting you pass expressions of any type back and forth and use Java objects in *Mathematica* in a meaningful way. This means that when writing your own Java classes to call from *Mathematica*, you usually do not need to do anything special. You write the code in exactly the same way as you would if

you wanted to use the class only from Java. (One important exception to this rule is that because it is comparatively slow to call into Java from *Mathematica*, you *might* need to design your classes in a way that will not require an excessive number of method calls from *Mathematica* to get the job done. This issue is discussed in detail in "Overhead of Calls to Java".)

In some cases, you might want to exert more direct control over the interaction with *Mathematica*. For example, you might want a method to return something different to *Mathematica* than what the method itself returns. Or you might want the method to not just return something, but also trigger a side effect in *Mathematica*—for example, printing something or displaying a message under certain conditions. You can even have an extended "dialog" with *Mathematica* before your method returns, perhaps invoking multiple computations in *Mathematica* and reading their results. You might also want to write a class of the `MathListener` type that calls into *Mathematica* as the result of some event triggered in Java.

If you do not want to do any of these things, then you can happily ignore this section. The whole point of *J/Link* is to make unnecessary the need to be concerned about the interaction with *Mathematica* through *MathLink*. Most programmers who want to write Java classes to be used from *Mathematica* will just write Java classes, period, without thinking about *Mathematica* or *J/Link*. Those programmers who want more control, or want to know more about the possibilities available with *J/Link*, read on.

The issues discussed in this section require some knowledge of *MathLink* programming (or, more precisely, *J/Link* programming using the Java methods that use *MathLink*), which is discussed in detail in "Writing Java Programs that use *Mathematica*". The fact that you meet some of these methods and issues here is a consequence of the false but useful dichotomy, noted in the Introduction, between using *MathLink* to write "installable" functions to be called from *Mathematica* and using *MathLink* to write front ends for *Mathematica*. *MathLink* is always used in the same way, it is just that virtually all of it is handled for you in the installable case. This section is about how to go beyond this default behavior, so you will be making direct *J/Link* calls to read and write to the link. Thus you will encounter concepts, classes, and methods in this section that are not explained until "Writing Java Programs That Use *Mathematica*".

Some of the discussion in this section will compare and contrast the process of writing an installable program in C. This is designed to help experienced *MathLink* programmers understand how *J/Link* works, and also to convince you that *J/Link* is a superior solution to using C, C++, or FORTRAN.

Installable Functions—The Old Way

Writing a so-called "installable" or "template" program in C requires a number of steps. If you have a file foo.c that contains a function `foo`, to call it from *Mathematica* you must first write a template (.tm) file that contains a template entry describing how you want `foo` to be called from *Mathematica*, what types of arguments it takes, and what it returns. You then pass this .tm file through a tool called `mprep`, which writes a file of C code that manages some, possibly all, of the *MathLink*-related aspects of the program. You also need to write a simple `main` routine, which is always the same. You then compile all of these files, resulting in an executable for just one platform.

Two big drawbacks of this method are that you need to write a template entry for every single function you want to call (imagine doing that for a whole function library), and the compiled program is not portable to other platforms. The biggest drawback, however, is that there is no automatic support for anything but the simplest types. If you want to do something as basic as returning a list of integers, you need to write the *MathLink* calls to do that yourself. And forget about object-oriented programming, as there is no way to pass "objects" to *Mathematica*.

Installable Functions in Java

J/Link makes all those steps go away. As you have seen all throughout this tutorial, you can literally call any method in any class, without any preparation.

It is only in cases where the default behavior of calling a method and receiving its result is not enough that you need to write specialty Java code. The rest of this section will examine some of the special techniques that can be used.

Setting Up Definitions in *Mathematica* When Your Class Is Loaded

Template entries in .tm files required by installable *MathLink* programs written in C have two features that might appear to be lost in *J/Link*. The first feature is the ability to specify arbitrary *Mathematica* code to be evaluated when the program is first "installed." This is done by using the `:Evaluate:` line in a template entry. The second feature is the ability to specify the way in which the function is to be called from *Mathematica*, including the name of the *Mathematica* function that maps to the C function, its argument sequence, how those arguments are mapped to the ones provided to the C function, and possibly some processing to be done on them before they are sent. This information is specified in the `:Pattern:` and `:Arguments:` lines of a template entry.

These two features are related to each other, because they both rely on the ability to specify *Mathematica* code that is loaded when an external program is installed. *J/Link* gives you this ability and more, through two special methods called `onLoadClass()` and `onUnloadClass()`. When a class is loaded into *Mathematica*, either directly through `LoadJavaClass` or indirectly by calling `JavaNew`, it is examined to see if it has a method with the following signature:

```
public static void onLoadClass(KernelLink ml);
```

If such a method is present, it will be called after all the method and field definitions for the class are set up in *Mathematica*. Because a class can only be loaded once in a Java session, this method will only be called once in the lifetime of a single Java runtime, although it may be called more than once in the lifetime of a single *Mathematica* kernel (because the user can repeatedly launch and quit the Java runtime). The `KernelLink` that is provided as an argument to this method is of course the link back to *Mathematica*.

A typical use for this feature would be to define the text for an error message issued by one of the methods in the class. Here is an example:

```
public static void onLoadClass(KernelLink ml) throwsMathLinkException {
    ml.evaluate("MyClass::sun = \"The foo() method can only be called on
Sunday.\"");
    ml.discardAnswer();
}
```

Note that this method throws `MathLinkException`. Your `onLoadClass()` method can throw any exceptions you like (a `MathLinkException` would be typical). This will not interfere with the matching of the expected signature for `onLoadClass()`. If an exception is thrown during `onLoadClass`, it will be handled gracefully, meaning that the normal operation of `LoadJavaClass` will not be affected. The only exception to this rule is if your code throws an exception while it is interacting with the link to the kernel, and more specifically, in the period between the time that it sends a computation to the kernel and the time that it begins to read the result. In other words, exceptions you throw will not break the `LoadJavaClass` mechanism, but it is up to you to make sure that you do not screw up the link's state by starting something you do not finish.

Another reason to use `onLoadClass()` would be if you wanted to create a *Mathematica* function for users to call that "wrapped" a static method call, providing it with a preferred name or argument sequence. If you have a class named `MyClass` with the method `public static`

`void myMethod(double[a])`, the definition that will be automatically created for it in *Mathematica* will require that its argument be a list of real numbers or integers. Say you want to add a definition named `MyMethod`, having the traditional *Mathematica* capitalization, and you also want this function automatically to use `N` on its argument so that it will work for anything that will *evaluate* to a list of numbers, such as `{Pi, 2Pi, 3Pi}`. Here is how you would set up such an additional definition:

```
public static void onLoadClass(KernelLink ml) throwsMathLinkException {
    ml.evaluate("MyMethod[x_] := myMethod[N[x]]");
    ml.discardAnswer();
}
```

In other words, if you are not happy with the interface to the class that will automatically be created in *Mathematica*, you can use `onLoadClass()` to set up the desired definitions without changing the Java interface.

The *Mathematica* context that will be current when `onLoadClass()` is called is the context in which all the class' static methods and fields are defined. That is why in the preceding example the definition was made for `MyMethod` and not `MyClass`MyMethod`. This is important since you cannot know the correct context in your Java code because it is determined by the user via the `AllowShortContext` option to `LoadJavaClass`.

It is generally not a good idea to use `onLoadClass()` to send a lot of code to *Mathematica*. This will make the behavior of your class hard for people to understand because the *Mathematica* code is hidden, and also inflexible since you would have to recompile it to make changes to the embedded *Mathematica* code. If you have a lot of code that needs to accompany a Java class, it is better to put that code into a *Mathematica* package file that you or your users load. That is, rather than having users load a class that dumps a lot of code into *Mathematica*, you should have your users load a *Mathematica* package that loads your class. This will provide the greatest flexibility for future changes and maintenance.

Finally, there is no reason why your `onLoadClass()` method needs to restrict itself to making *J/Link* calls. You could perform operations specific to the Java side, for example, writing some debugging information to the Java console window, opening a file for writing, or whatever else you desire.

Similar to the handling of the `onLoadClass()` method, the `onUnloadClass()` method is called when a class is unloaded. Every loaded class is unloaded automatically by `UninstallJava` right before it quits the Java runtime. You can use `onUnloadClass()` to remove definitions created by `onLoadClass()`, or perform any other clean-up you would like. The signature of `onUnloadClass()` must be the following, although it can throw any exceptions:

```
public static void onUnloadClass(KernelLink ml);
```

Note that the meaning of loading and unloading classes here refers to being loaded by *Mathematica* with `LoadJavaClass` either directly or indirectly. It does not refer to the loading and unloading of classes internally by the Java runtime. Class loading by the Java runtime occurs when the class is first used, which may have occurred long before `LoadJavaClass` was called from *Mathematica*.

Manually Returning a Result to *Mathematica*

The default behavior of a Java method called from *Mathematica* is to return to *Mathematica* exactly what the method itself returns. There are times, however, when you want to return something else. For example, you might want to return an integer in some circumstances, and a symbol in others. Or you might want a method to return one thing when it is being called from Java, and return something different to *Mathematica*. In these cases, you will need to manually send a result to *Mathematica* before the method returns.

Say you are writing a file-reading class that you want to call from *Mathematica*. Because you want almost the identical behavior to the standard class `java.io.FileInputStream`, your class will be a subclass of it. The only changes you want to make are to provide some more *Mathematica*-like behavior. One example is that you want the `read` method to return not -1 when it reaches the end of the file, but rather the symbol `EndOfFile`, which is what *Mathematica*'s built-in file-reading functions return.

```
import java.io.*;
import com.wolfram.jlink.*;

public class MyFileReader extends FileInputStream {

    <<constructors, other methods deleted>>

    public int read() {

        int i = super.read();
        if (i == -1) {
            KernelLink link = StdLink.getLink();
            if (link != null) {
                link.beginManual();
                try {
                    link.putSymbol("EndOfFile");
                } catch (MathLinkException e) {}
            }
        }
        return i;
    }
}
```

If the file has reached the end, `i` will be -1, and you want to manually return something to *Mathematica*. The first thing you need to do is get a `KernelLink` object that can be used to communicate with *Mathematica*. This is obtained by calling the static method `StdLink.getLink()`. If you have written installable *MathLink* programs in C, you will recognize the choice of names here. A C program has a global variable named stdlink that holds the link back to *Mathematica*. *J/Link* has a `StdLink` class that has a few methods related to this link object.

The first thing you do is check whether `getLink()` returns `null`. It will never be `null` if the method is being called from *Mathematica*, so you can use this test to determine whether the method is being called from *Mathematica* or as part of a normal Java program. In this way, you can have a method that can be used from Java in the usual way when a *Mathematica* kernel is nowhere in sight. The `getLink()` call works no matter if the method is called directly from *Mathematica*, or indirectly as part of a chain of methods triggered by a call from *Mathematica*.

Once you have verified that a link back to the kernel exists, the first thing to do is inform *J/Link* that you will be sending the result back to *Mathematica* yourself, so it should not try automatically to send the method's return value. This is accomplished by calling the `beginManual()` method on the `KernelLink` object.

You *must* call `beginManual()` before you send any part of a result back to *Mathematica*. If you fail to do this, the link will get out of sync and the next *J/Link* call you make from *Mathematica* will probably hang. It is safe to call `beginManual()` more than once, so you do not have to worry that your method might be called from another method that has already called `beginManual()`.

Returning to the example program, the next thing after `beginManual()` is to make the required "put"-type calls to send the result back to *Mathematica* (in this case, just a single `putSymbol()`). As always, these calls can throw a `MathLinkException`, so you need to wrap them in a `try/catch` block. The `catch` handler is empty, since there really is not anything to do in the unlikely event of a *MathLink* error. The internal *J/Link* code that wraps all method calls will handle the cleanup and recovery from any *MathLink* error that might have occurred calling `putSymbol()`. You do not need to do anything for `MathLinkExceptions` that occur while you are putting a result manually. The method call will return `$Failed` to *Mathematica* automatically.

Installable programs written in C can also manually send results back. This is indicated by using the `Manual` keyword in the function's template entry. Thus for C programs the manual/automatic decision must be made at compile time, whereas with *J/Link* it is a runtime switch. You can have it both ways with *J/Link*—a normal automatic return in some circumstances and a manual return in others, as the preceding example demonstrates.

Requesting Evaluations by *Mathematica*

So far, you have seen only cases where a Java method has a very simple interaction with *Mathematica*. It is called and returns a result, either automatically or manually. There are many circumstances, however, where you might want to have a more complex interaction with *Mathematica*. You might want a message to appear in *Mathematica*, or some `Print` output, or you might want to have *Mathematica* evaluate something and return the answer to you. This is a completely separate issue from what you want to return to *Mathematica* at the *end* of your method—you can request evaluations from the body of a method whether it returns its final result manually or not.

In some sense, when you perform this type of interaction with *Mathematica* you are turning the tables on *Mathematica*, reversing the "master" and "slave" roles for a moment. When *Mathematica* calls into Java, the Java code is acting as the slave, performing a computation and returning control to *Mathematica*. In the middle of a Java method, however, you can call back into *Mathematica*, temporarily turning it into a computational server for the Java side. Thus you would expect to encounter essentially all the same issues that are discussed in "Writing Java Programs That Use *Mathematica*", and you would need to understand the full *J/Link* Java-side API.

The full treatment of the `MathLink` and `KernelLink` interfaces is presented in "Writing Java Programs That Use *Mathematica*". This section discusses a few special methods in `KernelLink` that are specifically for use by "installed" methods. You have already seen one, the `beginManual()` method. Now you will treat the `message()`, `print()`, and `evaluate()` methods.

The task of issuing a *Mathematica* message from a Java method and triggering some `Print` output are so commonly done that the `KernelLink` interface has special methods for these operations. The method `message()` performs all the steps of issuing a *Mathematica* message. It comes in two signatures:

```
public void message(String symtag, String arg);
public void message(String symtag, String[] args);
```

The first form is for when you just have a single string argument to be slotted into the message text, and the second form is for if the message text needs two or more arguments. You can pass `null` as the second argument if the message text needs no arguments.

The `print()` method performs all the steps necessary to invoke *Mathematica*'s `Print` function:

```
public void print(String s);
```

Here is an example method that uses both. Assume that the following messages are defined in *Mathematica* (this could be from loading a package or during this class' `onLoadClass()` method):

```
Foo::arg = "The `1` argument to foo must be greater than or equal to 0."
```

Here is the Java code:

```
public static double foo(double x, double y) {

    KernelLink link = StdLink.getLink();
    if (link != null) {
        link.print("inside foo");
        if (x < 0)
            link.message("Foo::arg", "first");
        if (y < 0)
            link.message("Foo::arg", "second");
    }
    return Math.sqrt(x) * Math.sqrt(y);
}
```

Note that `print()` and `message()` send the required code to *Mathematica* and also read the result from the link (it will always be the symbol `Null`). They do not throw `MathLinkException` so you do not have to wrap them in `try/catch` blocks.

Here is what happens when you call `foo()`:

```
LoadJavaClass["MyClass", StaticsVisible → True];
foo[1.0, -2.0]
```

```
inside foo
```

```
Foo::arg : The second argument to foo must be greater than or equal to 0.
```

```
Indeterminate
```

Note that you automatically get `Indeterminate` returned to *Mathematica* when a floating-point result from Java is NaN ("Not-a-Number").

The methods `print()` and `message()` are convenience functions for two special cases of the more general notion of sending intermediate evaluations to *Mathematica* before your method returns a result. The general means of doing this is to wrap whatever you send to *Mathematica* in `EvaluatePacket`, which is a signal to the kernel that this is not the final result, but rather something that it should evaluate and send the result back to Java. You can explicitly send the `EvaluatePacket` head, or you can use one of the methods in `KernelLink` that use `EvaluatePacket` for you. These methods are:

```
void evaluate (String s) throws MathLinkException;
String evaluateToInputForm (String s, int pageWidth);
String evaluateToOutputForm (String s, int pageWidth);
byte[] evaluateToImage (String s, int width, int height);
byte[] evaluateToTypeset (String s, int pageWidth, boolean useStdForm);
```

These methods are discussed in "Writing Java Programs that use *Mathematica*" (actually, they also come in several more flavors with other argument sequences). Here is a simple example:

```
public static double foo(double x, double y) {

    KernelLink link = StdLink.getLink();
    if (link != null) {
        try {
            link.evaluate("2+2");
            // Wait for, and then read, the answer.
            link.waitForAnswer();
            int sum1 = link.getInteger();

            // evaluateToOutputForm makes the result come back as a
            // string formatted in OutputForm, and all in one step
            // (no waitForAnswer call needed).
            String s = link.evaluateToOutputForm("3+3");
            int sum2 = Integer.parseInt(s);

            // If you want, put the whole evaluation piece by piece,
            // including the EvaluatePacket head.
            link.putFunction("EvaluatePacket");
            link.putFunction("Plus", 2);
            link.put(4);
            link.put(4);
            link.waitForAnswer();
            int sum3 = link.getInteger();
        } catch (MathLinkException e) {
            // The only type of mathlink error we are likely to get
            // is from a "get" function when what we are trying to
            // get is not the type of expression that is waiting. We
            // just clear the error state, throw away the packet we
            // are reading, and let the method finish normally.
            link.clearError();
            link.newPacket();
        }
    }
    return Math.sqrt(x) * Math.sqrt(y);
}
```

Throwing Exceptions

Any exceptions that your method throws will be handled gracefully by *J/Link*, resulting in the printing of a message in *Mathematica* describing the exception. This was discussed in "How Exceptions Are Handled". If you are sending computations to *Mathematica* as described in the previous section, you need to make sure that an exception does not interrupt your code unexpectedly. In other words, if you start a transaction with *Mathematica*, make sure you complete it or you will leave the link out of sync and future calls to Java will probably hang.

Making a Method Interruptible

If you are writing a method that may take a while to complete, you should consider making it interruptible from *Mathematica*. In C *MathLink* programs, a global variable named `MLAbort` is provided for this purpose. In *J/Link* programs, you call the `wasInterrupted()` method in the `KernelLink` interface:

```
public boolean wasInterrupted();
```

Here is an example method that performs a long computation, checking every 100 iterations whether the user tried to abort it (using the **Interrupt Evaluation** or **Abort Evaluation** commands in the **Evaluation** menu).

```
public int foo() {

    KernelLink link = StdLink.getLink();
    for (int i = 0; i < 10000, i++) {
        ... perform one step ...
        if (i % 100 == 0 && link.wasInterrupted())
            return 0;  // Return value will not be seen by Mathematica.
    }
    return 42;
}
```

This method returns 0 if it detects an attempt by the user to abort, but this value will never be seen by *Mathematica*. This is because *J/Link* causes a method or constructor call that is aborted to return `Abort[]`, whether or not you detect the abort in your code. Therefore, if you detect an abort and want to honor the user's request, just return some value right away. When *J/Link* returns `Abort[]`, the user's entire computation is aborted, just as if the `Abort[]` was embedded in *Mathematica* code. This means that you do not have to be concerned with any details of propagating the abort back to *Mathematica*—all you have to do is return prematurely if you detect an abort request, and the rest is handled for you.

J/Link makes no distinction between an interrupt request and an abort request; they each cause `wasInterrupted()` to return `true`. Recall that *Mathematica* has separate commands for interrupting and aborting computations. The "Abort" operation (Alt+. on Windows) causes the entire computation to end as soon as possible and return `$Aborted`. The "Interrupt" operation (Alt+, on Windows) brings up a dialog box with further choices. If this **Interrupt** dialog box is triggered when a Java method is executing, it has a different set of buttons than when normal *Mathematica* code is executing. One of the options is **Send Abort to Linked Program** and another is **Send Interrupt to Linked Program**. Both of these choices have the same effect for Java methods, which is to cause `wasInterrupted()` to return `true` and the call to return `Abort[]` when it completes. The third button is **Kill Linked Program**, which will cause the Java runtime to quit. If you call a Java method that is not interruptible, killing the Java runtime in this way is the only way to make the method call terminate (you can also kill the Java runtime using process control features of your operating system).

Sometimes you might want a Java method to detect an abort and do something other than cause the entire *Mathematica* computation to abort. For example, you might want a loop to stop and return its results up to that point. Note that this is not generally recommended. Users expect a program to abort and return `$Aborted` when they issue an abort request. In some cases, however, especially if the code is not intended for use by a large community, you might find it useful to use an abort as a "message" to communicate some information to your Java code instead of just having the computation aborted. This idea is similar to *Mathematica*'s `CheckAbort` function, which allows you to detect an abort and absorb it so that it does not propagate further and abort the entire computation. To "absorb" the abort in your Java code so that *J/Link* does not return `Abort[]`, simply call the `clearInterrupt()` method:

```
public void clearInterrupt();
```

Here is an example:

```
public int foo() {

    KernelLink link = StdLink.getLink();
    for (int i = 0; i < 10000, i++) {
        ... perform one step ...
        if (i % 100 == 0 && link.wasInterrupted()) {
            link.clearInterrupt();
            return resultSoFar; // This is the value that will be returned
to Mathematica
        }
    }
    ...
    return 42;
}
```

Writing Your Own Event Handler Code

"Handling Events with *Mathematica* Code: The "MathListener" Classes" introduced the topic of triggering calls into *Mathematica* as a response to events fired in Java, such as clicking a button. A set of classes derived from `MathListener` is provided by *J/Link* for this purpose. You are not required to use the provided `MathListener` classes, of course. You can write your own classes to handle events and put calls into *Mathematica* directly into their code. All the event handler classes in *J/Link* are derived from the abstract base class `MathListener`, which takes care of all the details of interacting with *Mathematica*, and also provides the `setHandler()` methods that you use to associate events with *Mathematica* code. Users who want to write their own `MathListener`-style classes (for example, for one of the Swing-specific event listener interfaces, which *J/Link* does not provide) are strongly encouraged to make their classes subclasses of `MathListener` to inherit all this functionality. You should examine the source code for `MathListener`, and also one of the concrete classes derived from it (`MathActionListener` is probably the simplest one) to see how it is written. You can use this as a starting point for your own implementation.

There is a new feature of *J/Link* 2.0 that should be pointed out in this context. This is the `ImplementJavaInterface` *Mathematica* function, which lets you implement any Java interface entirely in *Mathematica* code. `ImplementJavaInterface` is described in more detail in "Implementing a Java Interface with *Mathematica* Code", but a common use for it would be to

create event-handler classes that implement a "Listener"-type interface for which *J/Link* does not have a built-in `MathListener`. This is discussed in more detail in "Implementing a Java Interface with *Mathematica* Code", and if you choose this technique, then you do not have to worry about any of the issues in this section because they are handled for you.

If you are going to write a Java class, and you choose not to derive your class from `MathListener`, there are two very important rules that *must* be adhered to when writing event-handler code that calls into *Mathematica*. To be more precise, these rules apply whenever you are writing code that needs to call into *Mathematica* at a point when *Mathematica* is not currently calling into Java. That may sound confusing, but it is really very simple. "Requesting Evaluations by *Mathematica*" showed how to request evaluations by *Mathematica* from within a Java method. In this case, *Mathematica* has called your Java method, and while *Mathematica* is waiting for the result, your code calls back to perform some computation. This works fine as described in that earlier section, because at the point the code calls back into *Mathematica*, *Mathematica* is in the middle of a call to Java. This is a true "callback"—*Mathematica* has called Java, and during the handling of this call, Java calls back to *Mathematica*. In contrast, consider the case where some Java code executes in response to a button click. When the button click event fires, *Mathematica* is probably not in the middle of a call to Java.

Special considerations are necessary in the latter case because there are two threads in the Java runtime that are using *MathLink*. The first one is created and used by the internals of *J/Link* to handle standard calls into Java originating in *Mathematica* as described throughout this tutorial. The second one is the Java user interface thread (sometimes called the AWT thread), which is the one on which your event handler code will be called. You need to make sure that your use of the link back to the kernel on the user interface thread does not interfere with *J/Link*'s internal thread.

The following code shows an idealized version of the `actionPerformed()` method in the `MathActionListener` class. The actual code in `MathActionListener` is different, because this work is farmed out to the parent class, `MathListener`, but this example shows the correct flow of operations. This is the code that is executed when the associated object's action occurs (like a button click).

```
public void actionPerformed(ActionEvent e) {
    KernelLink ml = StdLink.getLink();
    StdLink.requestTransaction();
    synchronized (ml) {
        try {
            // Send the code to perform the user's requested operation.
            ml.putFunction("EvaluatePacket", 1);
            ... code to put rest of expression to evaluate goes here ...
            ml.endPacket();
            ml.discardAnswer();
        } catch (MathLinkException exc) {
            ...
        }
    }
}
```

The first rule to note in this code is that the complete transaction with *Mathematica*, which includes sending the code to evaluate and completely reading the result, is wrapped in a `synchronized(ml)` block. This is how you ensure that the user interface thread has exclusive access to the link for the entire transaction. The second rule is that the `synchronized(ml)` statement must be preceded by a call to `StdLink.requestTransaction()`. This call will block until the kernel is at a point where it is ready to accommodate evaluations originating in Java. The call must occur before the `synchronized(ml)` block begins, and once you call it you must make sure that you send something to *Mathematica*. In other words, when `requestTransaction()` returns, the kernel will be blocking in an attempt to read from the Java link. The kernel will be stuck in this state until you send it something, so you must protect against a Java exception being thrown after you call `requestTransaction()` but before you send anything. Typically you will do this simply by calling `requestTransaction()` immediately before the `synchronized(ml)` block begins and you start sending something.

It was just said that StdLink.requestTransaction() will block until the kernel is ready to accept evaluations originating in Java. To be specific, it will block until one of the following conditions occurs:

- *Mathematica* executes DoModal
- *Mathematica* executes ServiceJava
- Kernel sharing has been turned on via ShareKernel or ShareFrontEnd, and the kernel is not busy with another computation
- *Mathematica* is already in the middle of a call to Java
- Java is not being used from *Mathematica* (InstallJava has not been called)

These conditions should make sense given the discussion about creating user interface elements in the section "Creating Windows and Other User Interface Elements". DoModal, ShareKernel, and ServiceJava are the three ways in which you direct the kernel's attention to the Java link so that it can detect incoming request for computations.

If you make the common mistake of inadvertently triggering a call to *Mathematica* from Java before you have called DoModal or ShareKernel, the Java user interface thread will hang. This can be easily remedied by calling DoModal, ShareKernel, or ServiceJava afterwards (ServiceJava may need to be called more than once, if more than one event callback is queued up).

If the rule about when it is necessary to use StdLink.requestTransaction() and synchronized(ml) is confusing, you will be happy to learn that it is fine to use these constructs in any code that calls *Mathematica*. In code that does not need them, they are pointless, but harmless, and will not cause the calling thread to block. If you are writing a Java method that needs to call *Mathematica* and there is any chance that it might be called from the user interface thread, add the StdLink.requestTransaction() and synchronized(ml).

Debugging Your Java Classes

You can use your favorite debugger to debug Java code that is called from *Mathematica*. The only issue is that you typically have to launch a Java program inside the debugger to do this. The Java program that you need to launch is the one that is normally launched for you when you call InstallJava. The class that contains *J/Link*'s main() method is com.wolfram.jlink.Install. Thus, the command line to start *J/Link* that is executed internally by InstallJava is typically

```
java -classpath /path/to/JLink.jar com.wolfram.jlink.Install
```

There may be additions or modifications to this depending on the options to `InstallJava`, and also some extra *MathLink*-specific arguments are tacked on at the end. To use a debugger, you just have to launch Java with the appropriate command-line arguments that allow you to establish the link to *Mathematica* manually.

If you use a development environment that has an integrated debugger, then the debugger probably has a setting for the main class to use (the class whose `main()` method will be invoked) and a setting for command-line arguments. For example, in WebGain Visual Café, you can set these values in the **Project** panel of the **Project/Options** dialog. Set the main class to be `com.wolfram.jlink.Install`, and the arguments to be something like this:

```
(On Windows:)
-linkmode listen -linkname foo

(On Unix/Linux:)
-linkmode listen -linkprotocol tcp -linkname 1234
```

Then start the debugging session. You should see the *J/Link* copyright notice printed and then Java will wait for *Mathematica* to connect. To do this, go to your *Mathematica* session, make sure the JLink.m package has been read in, and execute:

```
(* On Windows: *)
ReinstallJava[LinkConnect["foo"]]

(* On Unix: *)
ReinstallJava[LinkConnect["1234", LinkProtocol -> "TCP"]]
```

This works because `ReinstallJava` can take a `LinkObject` as its argument, in which case it will not try to launch Java itself. This allows you to manually establish the *MathLink* connection between Java and *Mathematica*, then feed that link to `ReinstallJava` and let it do the rest of the work of preparing the *Mathematica* and Java sides for interacting with each other.

If you like to use a command-line debugger like jdb, you can do the following:

```
C:\>jdb
Initializing jdb...
> run com.wolfram.jlink.Install -linkmode listen -linkname foo
running ...
main[1] J/Link (tm)
Copyright (C) 1999-2000, Wolfram Research, Inc. All Rights Reserved.
www.wolfram.com
Version 1.1

Current thread "main" died. Execution continuing...
>
```

The message about the main thread dying is normal. Now jdb is ready for commands. First, though, you have to execute in your *Mathematica* session the `LinkConnect` and `ReinstallJava` lines shown earlier. This example was for Windows, so Unix users will have to adjust the `run` line to reflect the proper arguments:

```
> run com.wolfram.jlink.Install -linkmode listen -linkprotocol tcp
-linkname 1234
```

Deploying Applications that use J/Link

This section discusses some issues relevant to developers who are creating add-ons for *Mathematica* that use *J/Link*.

J/Link uses its own custom class loader that allows it to find classes in a set of locations beyond the startup class path. As described in "Dynamically Modifying the Class Path", users can grow this set of extra locations to search for classes by calling the `AddToClassPath` function. One of the motivations for having a custom class loader was to make it easy for application developers to distribute applications that have parts of their implementation in Java. If you structure your application directory properly, your users will be able to install it simply by copying it into any standard location for *Mathematica* applications. *J/Link* will be able to find your Java classes immediately, without users having to perform any classpath-related operations or even restart Java.

If your *Mathematica* application uses *J/Link* and includes its own Java components, you should create a Java subdirectory in your application directory. You can place any jar files that your application needs into this Java subdirectory. If you have loose class files (not bundled into a

jar file), they should go into an appropriately nested subdirectory of the Java directory. "Appropriately nested" means that if your class is in the Java package `com.somecompany.math`, then its class file goes into the com/somecompany/math subdirectory of the Java directory. If the class is not in any package, it can go directly into the Java directory. *J/Link* can also find native libraries and resources your application needs. Native libraries must be in a subdirectory of your Java/Libraries directory that is named after the `$SystemID` of the platform on which it is installed. Here is an example directory structure for an application that uses *J/Link*:

```
MyApp/
    ... other files and directories used by the application ...
    Java/
        MyAppClasses.jar
        MyImage.gif
        Libraries/
            Windows/
                MyNativeLibrary.dll
            PowerMac/
                MyNativeLibrary
            Darwin/
                libMyNativeLibrary.jnilib
            Linux/
                libMyNativeLibrary.so
            ... and so on for other Unix platforms
```

Your application directory must be placed into one of the standard locations for *Mathematica* applications. These locations are listed as follows. In this notation, `$InstallationDirectory`/AddOns/Applications means "The AddOns/Applications subdirectory of the directory whose value is given by the *Mathematica* variable `$InstallationDirectory`."

`$UserAddOnsDirectory`/Applications (Mathematica 4.2 and later only)

`$AddOnsDirectory`/Applications (Mathematica 4.2 and later only)

`$InstallationDirectory`/AddOns/Applications

`$InstallationDirectory`/AddOns/ExtraPackages

Coding Tips

Here are a few tips on producing high-quality applications. These suggestions are guided by mistakes that developers frequently make.

Call `InstallJava` in the body of a function or functions, not when your package is read in. It is best to avoid side effects during the reading of a package. Users expect reading in a package to be fast and to do nothing but load definitions. If you launch Java at this time, and it fails, it could cause a mysterious hang in the loading process. It is better to call `InstallJava` in the code of one or more of your functions. You probably do not need to call `InstallJava` in every single function that uses Java. Most applications have a few "major" functions that users are likely to use almost exclusively, or at least at the start of their session. If your application does not have this property, then provide an initialization function that your users must call first, and call `InstallJava` inside it.

Call `InstallJava` with no arguments. You cannot know what options your users need for Java on their systems, so do not override what they may have set up. It is the user's responsibility to make sure that they call `SetOptions` to customize the options for `InstallJava` as necessary. Typically this would be done in their `init.m` file.

Make sure you use `JavaBlock` and/or `ReleaseJavaObject` to avoid leaking object references. You cannot know how others will use your code, so you need to be careful to avoid cluttering up their sessions with a potentially large number of useless objects. Sometimes you need to create an object that persists beyond the lifetime of a single *Mathematica* function, like a viewer window. In such cases, use a `MathFrame` or `MathJFrame` as your top-level window and use its `onClose()` method to specify *Mathematica* code that releases all outstanding objects and unregisters kernel or front end sharing you may have used. If this is not possible, provide a cleanup function that users can call manually. Use `LoadedJavaObjects` to look at the list of objects referenced in *Mathematica* before and after your functions run; it should not grow in length.

If you use `ShareKernel` or `ShareFrontEnd`, make sure you save the return values from these functions and pass them as arguments to `UnshareKernel` and `UnshareFrontEnd`. Do not call `UnshareFrontEnd` or `UnshareKernel` with no arguments, as this will shut down sharing even if other applications are using it.

Do not assume that the Java runtime will not be restarted during the lifetime of your application. Although users are strongly discouraged to call `UninstallJava` or `ReinstallJava`, it happens. It is unavoidable that some applications will fail if the Java runtime is shut down at an inopportune time (e.g., when they have a Java window displayed), but there are steps you can take to increase the robustness of your application in the face of Java shut-downs and restarts. One step was already given as the first tip listed—call `InstallJava` at the start of your "major" functions. Another step is to avoid caching `JavaClass` or `JavaObject` expressions unnecessarily, as these will become invalid if Java restarts. An example of this is calling `InstallJava` and then `LoadJavaClass` and `JavaNew` several times when your package file is read in, and storing the results in private variables for the lifetime of your package. This is problematic if Java is restarted. Never store `JavaClass` expressions—call `LoadJavaClass` whenever there is any doubt about whether a class has been loaded into the current Java runtime. Calling `LoadJavaClass` is very inexpensive if the class has already been loaded. If you have a `JavaObject` that is very expensive to create and therefore you feel it necessary to cache it over a long period of time in a user's session, consider using the following idiom to test whether it is still valid whenever it is used. The `JavaObjectQ` test will fail if Java has been shut down or restarted since the object was last created, so you can then restart Java and create and store a new instance of the object.

```
SomeFunction[] :=
    Module[{...},
        If[!JavaObjectQ[$myCachedExpensiveJavaObject],
            InstallJava[];
            $myCachedExpensiveJavaObject = JavaNew[...];
        ];
        ... use $myCachedExpensiveJavaObject ...
    ]
```

Do not call `UninstallJava` or `ReinstallJava` in your application. You need to coexist politely with other applications that may be using Java. Do not assume that when your package is done with Java, the user is done with it as well. Only users should ever call `UninstallJava`, and they should probably never call it either. There is no cost to leaving Java running. Likewise, users will rarely call `ReinstallJava` unless they are doing active Java development and need to reload modified versions of their classes.

Example Programs

Introduction

This section will work through some example programs. These examples are intended to demonstrate a wide variety of techniques and subtleties. Discussions include some nuances in the implementations and touch on most of the major issues in *J/Link* programming.

This will take a relatively rigorous approach, and in particular it will be careful to avoid leaking references. As discussed in the section "JavaBlock", `JavaBlock` and `ReleaseJavaObject` are the tools in this fight, but if you find yourself becoming the least bit confused about the subject, just ignore it completely. For many casual, personal uses of *J/Link*, you can forget about memory management issues, and just let Java objects pile up.

J/Link includes a number of notebooks with sample programs, including most of the programs developed in this section. These notebooks can be found in the `<Mathematica dir>`/SystemFiles/Links/JLink/Examples/Part1 directory.

A Beep Function

Here is a very simple example. *Mathematica* does not have a `Beep` function to provide simple alerts. But Java has a `beep()` method and, by virtue of that, *Mathematica* has one too.

```
Beep[] :=
 (
  LoadJavaClass["java.awt.Toolkit"];
  Toolkit`getDefaultToolkit[]@beep[]
 )
```

You will notice a short delay the first time `Beep[]` is executed. This is due to the `LoadJavaClass` call, which only takes measurable time the first time it is called for any given class.

```
Beep[]
```

This is a perfectly good beep function, and many users will not need to go beyond this. If you are writing code for others to use, however, you will probably want to embellish this code a little bit. Here is a more professional version of the same function.

```
BetterBeep[]:=
    JavaBlock[
        InstallJava[];
        LoadJavaClass["java.awt.Toolkit"];
        Toolkit`getDefaultToolkit[]@beep[];
    ]
```

Note that the first thing you do is call InstallJava. It is a good habit to call InstallJava in functions that use *J/Link*, at least if you are writing code for others to use. If InstallJava has already been called, subsequent calls will do nothing and return very quickly. The whole program is wrapped in JavaBlock. As discussed in the section "JavaBlock", JavaBlock automates the process of releasing references to objects returned to *Mathematica*. The getDefaultToolkit() method returns a Toolkit object, so you want to release the JavaObject that gets created in *Mathematica*. The getDefaultToolkit() method returns a reference to the same Toolkit object every time it is called, so even if you do not call JavaBlock, you will only "leak" one object in an entire session. You could also write Beep using an explicit call to ReleaseJavaObject.

```
(* Alternative version *)
BetterBeep2[]:=
    Module[{toolkit},
        InstallJava[];
        LoadJavaClass["java.awt.Toolkit"];
        toolkit = Toolkit`getDefaultToolkit[];
        toolkit@beep[];
        ReleaseJavaObject[toolkit]
    ]
```

The advantage to using JavaBlock is that you do not have to think about what, if any, methods might return objects, and you do not have to assign them to variables.

Formatting Dates

Here is an example of a computation performed in Java. Java provides a number of powerful date- and calendar-oriented classes. Say you want to create a nicely formatted string showing the time and date. In this first step you create a new Java Date object representing the current date and time.

```
date = JavaNew["java.util.Date"]
```

```
«JavaObject[java.util.Date] »
```

Next you load the DateFormat class and create a formatter capable of formatting dates.

```
LoadJavaClass["java.text.DateFormat"];
dateFormatter = DateFormat`getInstance[]
```

```
«JavaObject[java.text.SimpleDateFormat] »
```

Now you call the format() method, passing the Date object as its argument.

```
dateFormatter@format[date]
```

```
10/9/00 4:56 AM
```

There are many different ways in which dates and times can be formatted, including respecting a user's locale. Java also has a useful number-formatting class, an example of which was given in "An Optimization Example".

A Progress Bar

A simple example of a popup user interface for a *Mathematica* program is a progress bar. This is an example of a "non-interactive" user interface, as defined in "Interactive and Non-Interactive Interfaces", because it does not need to call back to *Mathematica* or return a result to *Mathematica*. The implementation uses the Swing user interface classes, because Swing has a built-in class for progress bars. (You cannot run this example unless you have Swing installed. It comes as a standard part of Java 1.2 or later, but you can get it separately for Java 1.1.x. Most Java development tools that are still at Version 1.1.x come with Swing.) The complete code for this example is also provided in the file ProgressBar.nb in the JLink/Examples/Part1 directory.

The code is commented to point out the general structure. There are several classes and methods used in this code that may be unfamiliar to you. Just keep in mind that this is completely standard Java code translated into *Mathematica* using the *J/Link* conventions. It is line-for-line identical to a Java program that does the same thing.

This code is presented as a complete program, but this does not suggest that it should be developed that way. The interactive nature of *J/Link* lets you tinker with Java objects a line at a time, experimenting until you get things just how you want them. Of course, this is how *Mathematica* programs are typically written, and *J/Link* lets you do the same with Java objects and methods.

You can create a function `ShowProgressBar` that prepares and displays a progress bar dialog. The bar will be used to show percentage completion of a computation. You can supply the initial percent completed or use the default value of zero. `ShowProgressBar` returns the `JProgressBar` object because the bar needs to be updated later by calling `setValue()`. Note that because you return the `bar` object from the `JavaBlock`, it is not released like all other new Java objects created within this `JavaBlock`. This is a new behavior of `JavaBlock` in *J/Link* 2.0. If what is returned from a `JavaBlock` is precisely a single Java object (and not, for example, a list of objects), then this object is not released. `JavaBlock` is discussed in the section "JavaBlock".

```
ShowProgressBar[title_String:"Computation Progress",
                    caption_String:"Percent complete:",
                    percent_Integer:0
                  ] :=
    JavaBlock[
        Module[{frame, panel, label, bar},
            InstallJava[];
            bar = JavaNew["javax.swing.JProgressBar"];
            frame = JavaNew["javax.swing.JFrame", title];
            frame@setSize[300, 110];
            frame@setResizable[False];
            frame@setLocation[400, 400];
            panel = JavaNew["javax.swing.JPanel"];
            panel@setLayout[Null];
            frame@getContentPane[]@add[panel];
            label = JavaNew["javax.swing.JLabel", caption];
            label@setBounds[20, 10, 260, 20];
            panel@add[label];
            bar@setBounds[20, 40, 260, 30];
            bar@setMinimum[0];
            bar@setMaximum[100];
            bar@setValue[percent];
            panel@add[bar];
            JavaShow[frame];
            bar
        ]
    ]
```

You also need a function to close the progress dialog and clean up after it. Only two things need to be done. First, the `dispose()` method must be called on the top-level frame window that contains the bar. Second, if you want to avoid leaking object references, you need to call `ReleaseJavaObject` on the bar object because it is the only object reference that escaped the `JavaBlock` in `ShowProgressBar`. You need to call `dispose()` on the `JFrame` object you created in `ShowProgressBar`, but you did not save a reference to it. The `SwingUtilities` class has a handy method `windowForComponent()` that will retrieve this frame, given the bar object.

```
DestroyProgressBar[bar_?JavaObjectQ] :=
    JavaBlock[
        LoadJavaClass["javax.swing.SwingUtilities"];
        SwingUtilities`windowForComponent[bar]@dispose[];
        ReleaseJavaObject[bar]
    ]
```

The bar dialog has a close box in it, so a user can dismiss it prematurely if desired. This would take care of disposing the dialog, but you would still need to release the bar object. `DestroyProgressBar` (and the bar's `setValue()` method) is safe to call whether or not the user closed the dialog.

Here is how you would use the progress bar in a computation. The call to `ShowProgressBar` displays the bar dialog and returns a reference to the bar object. Then, while the computation is running, you periodically call the `setValue()` method to update the bar's appearance. When the computation is done, you call `DestroyProgressBar`.

```
bar = ShowProgressBar[];
n = 0;
While[n <= 5,
    bar@setValue[n/5 * 100];
    Pause[1]; (* This simulates the time-consuming computation. *)
    n++
];
DestroyProgressBar[bar];
```

An easy way to test whether your code leaks object references is to call `LoadedJavaObjects[]` before and after the computation. If the list of objects gets longer, then you have forgotten to use `ReleaseJavaObject` or improperly used `JavaBlock`.

It can take several seconds to load all the Swing classes used in this example. This means that the first time `ShowProgressBar` is called, there will be a significant delay. You could avoid this delay by using `LoadJavaClass` ahead of time to explicitly load the classes that appear in `JavaNew` statements.

The dialog appears onscreen with its upper left at the coordinates (400, 400). It is left as an exercise to the reader to make it centered on the screen. (Hint: the `java.awt.Toolkit` class has a `getScreenSize()` method).

Finally, because the progress bar uses the Swing classes, you can play with the look-and-feel options that Swing provides. Specifically, you can change the theme at runtime. The progress bar window is not very complicated, so it changes very little in going from one look-and-feel theme to another, but this demonstrates how to do it. The effect is much more dramatic for more complex windows.

First, create a new progress bar window.

```
bar = ShowProgressBar[];
```

Now load some classes from which you need to call static methods.

```
LoadJavaClass["javax.swing.UIManager"];
LoadJavaClass["javax.swing.SwingUtilities"];
```

The default look and feel is the "metal" theme. You can change it to the native style look for your platform as follows (it helps to be able to see the window when doing this).

```
JavaBlock[
    UIManager`setLookAndFeel[UIManager`getSystemLookAndFeelClassName[]];
    frame = SwingUtilities`windowForComponent[bar];
    SwingUtilities`updateComponentTreeUI[frame]
]
```

Clean up.

```
DestroyProgressBar[bar]
```

A Simple Modal Input Dialog

You saw one example of a simple modal dialog in "Modal Windows". Presented here is another one—a basic dialog that prompts the user to enter an angle, with a choice of whether it is being specified in degrees or radians. This will demonstrate a dialog that returns a value to a running *Mathematica* program when it is dismissed, much like *Mathematica*'s built-in `Input` function, which requests a string from the user before returning. Dialogs like this one are not "modal" in the traditional sense that they must be closed before other Java windows can be used, but rather they are modal with respect to the kernel, which is kept busy until they are dismissed (that is, until `DoModal[]` returns). The section "Creating Windows and Other User Interface Elements" discusses modal and modeless Java windows in detail.

The code is rather straightforward and warrants little in the way of commentary. In creating the window and the controls within it, it exactly mirrors the Java code you would use if you were writing the program in Java. One technique it demonstrates is determining whether the **OK** or **Cancel** button was clicked to dismiss the dialog. This is done by having the `MathActionListener` objects assigned to the two buttons return different things in addition to calling `EndModal[]`. Recall that `DoModal[]` returns whatever the code that calls `EndModal[]` returns, so here you have the **OK** button execute `(EndModal[]; True)&`, a pure function that ignores its arguments, calls `EndModal[]`, and returns `True`, whereas the **Cancel** button executes `(EndModal[]; False)&`. Thus, `DoModal[]` returns `True` if the **OK** button was clicked, or `False` if the **Cancel** button was clicked. It will return `Null` if the window's close box was clicked (this behavior comes from the `MathFrame` itself).

It may take several seconds to display the dialog the first time `GetAngle[]` is called. This is due to the one-time cost of loading the several large AWT classes required. Subsequent invocations of `GetAngle[]` will be much quicker.

The complete code for this example is also provided in the file ModalInputDialog.nb in the JLink/Examples/Part1 directory.

```
GetAngle[] :=
    JavaBlock[
        Module[{frm, inputField, cbGroup, degBox, radBox,
                label, okButton, cancelButton, wasOKButton, angle},

            InstallJava[]; (* In case the user has not called it already. *)

            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            label = JavaNew["java.awt.Label", "Enter an angle:"];
```

```
            inputField = JavaNew["java.awt.TextField"];
            cbGroup = JavaNew["java.awt.CheckboxGroup"];
            degBox = JavaNew["java.awt.Checkbox", "degrees", cbGroup, True];
            radBox = JavaNew["java.awt.Checkbox", "radians", cbGroup, False];
            okButton = JavaNew["java.awt.Button", "OK"];
            cancelButton = JavaNew["java.awt.Button", "Cancel"];

            frm@setLayout[Null];
            frm@add[label];
            frm@add[inputField];
            frm@add[degBox];
            frm@add[radBox];
            frm@add[okButton];
            frm@add[cancelButton];

            frm@setBounds[200, 200, 200, 160];
            label@setBounds[20, 30, 150, 20];
            inputField@setBounds[20, 70, 60, 28];
            degBox@setBounds[100, 60, 80, 20];
            radBox@setBounds[100, 80, 80, 20];
            okButton@setBounds[40, 120, 50, 20];
            cancelButton@setBounds[100, 120, 50, 20];
            frm@setResizable[False];

            okButton@addActionListener[
                JavaNew["com.wolfram.jlink.MathActionListener",
                        "(EndModal[]; True)&"]
            ];
            cancelButton@addActionListener[
                JavaNew["com.wolfram.jlink.MathActionListener",
                        "(EndModal[]; False)&"]
            ];

            (* Now make the window visible and bring it to the foreground. *)
            JavaShow[frm];

            frm@setModal[];
            wasOKButton = DoModal[];
            (* Even though the window may have been closed, it is perfectly
               OK to extract values from the controls in the window.
            *)
            If[TrueQ[wasOKButton],
                angle = ToExpression[inputField@getText[]];
                If[angle =!= Null && degBox@getState[], angle *= Pi/180],
            (* else *)
                (* We will get here if the Cancel button was clicked
                   (wasOKButton will be False), or the dialog was closed
                   by clicking in its close box (wasOKButton will be Null).
                *)
                angle = $Failed
            ];
            (* If the cancel or OK buttons were clicked, frm is still
               visible, so we dispose it here.
            *)
            frm@dispose[];
            angle
        ]
    ]
```

Now invoke it.

```
GetAngle[]
```

π

A File Chooser Dialog Box

A useful feature for *Mathematica* programs is to be able to produce a file chooser dialog, such as the typical **Open** or **Save** dialog boxes. You could use such a dialog box to prompt a user for an input file or a file into which to write data. This is easily accomplished in a cross-platform way with Java, specifically with the `JFileChooser` class in the standard Swing library. The code for such a dialog box is provided in the file FileChooserDialog.nb in the JLink/Examples/Part1 directory.

Mathematica 4.0 introduced a new "experimental" function called `FileBrowse[]` that displays a file browser in the front end. Although this function is usable, it has several shortcomings compared to the Java technique presented next. One of the limitations is that it requires that the front end be in use. Another is that it is not customizable, so you always get a **Save file as:** dialog box and the concomitant behavior, which is not appropriate for an **Open**-type dialog box. The `JFileChooser` class used here allows very sophisticated customization, including setting the initial directory, masking out files based on their names or properties, controlling the title and text on the various buttons, supplying functions to validate the choice before the dialog box is allowed to be dismissed, allowing for multiple file selection, and allowing directories to be selected instead of files.

Although this example is a short program, the code has some unfortunate complexity (meaning "ugliness") in it related to making this special type of dialog window come to the foreground on all platforms. For this reason, the code is not presented here. Instead, some topics in the program code will be mentioned; you can read the full code and its associated comments in the example file if you are interested in the implementation details.

The `FileChooserDialog` function takes three string arguments. The first is the title of the dialog box (for example, **Select a data file to import**), the second is the text to appear on what is essentially the **OK** button (typically this will be **Open** or **Save**), and the third is the directory in which to start. You can also supply no arguments and get a default Open dialog box that starts in the kernel's current directory.

Although this is a "modal" dialog box, there is no need to use DoModal, because the showDialog() method will not return until the user dismisses the dialog box. Recall that DoModal is a way to force *Mathematica* to stall until the dialog box or other window is dismissed. Here, you get this behavior for free from showDialog(). The other thing that DoModal does is put the kernel into a loop where it is ready to receive input from Java, so you can script some of the functionality of the dialog via callbacks to *Mathematica*. The file chooser dialog box does not need to use *Mathematica* in any way until it returns the selected file, so you have no need for this other aspect that DoModal provides.

A second point of interest is in the name of the constant that showDialog() returns to indicate that the user clicked the **Save** or **Open** button instead of the **Cancel** button. The name of this constant in Java is JFileChooser.APPROVE_OPTION. Java names map to *Mathematica* symbols, so they must be translated if they contain characters that are not legal in *Mathematica* symbols, such as the underscore. Underscores are converted to a "U" when they appear in symbols, so the *Mathematica* name of this constant is JFileChooser`APPROVEUOPTION. See "Underscores in Java Names" for more information.

Sharing the Front End: Palette-Type Buttons

As discussed in the section "Creating Windows and Other User Interface Elements", one of the goals of *J/Link* is to allow Java user interface elements to be as close as possible to first-class members of the notebook front end environment in the way notebook and palette windows are. One of the ways this is accomplished is with the ShareKernel function, which allows Java windows to share the kernel's attention with notebook windows. Such Java windows are referred to as "modeless," not in the traditional sense of allowing other Java windows to remain active, but modeless with respect to the kernel, meaning that the kernel is not kept busy while they are open.

Beyond the ability to have Java windows share the kernel with the front end, it would be nice to allow actions in Java to cause effects in notebook windows, such as printing something, displaying a graph, or any of the notebook-manipulation commands like NotebookApply, NotebookPrint, SelectionEvaluate, SelectionMove, and so on. A good example of this is palette buttons. A palette button can cause the current selection to be replaced by something else and the resulting expression to be evaluated in place.

The ShareFrontEnd function lets actions in Java modeless windows trigger events in a notebook window just like can be done from palette buttons or *Mathematica* code you evaluate manually in a notebook. Remember that you get automatically the ability to interact with the front end when you use a *modal* dialog (i.e., when DoModal is running). When Java is being run in a modal way, the kernel's $ParentLink always points at the front end, so all side effect outputs get sent to the front end automatically. A modal window would not be acceptable for the palette example here because the palette needs to be an unobtrusive enhancement to the *Mathematica* environment—it cannot lock up the kernel while it is alive. ShareKernel allows Java windows to call *Mathematica* without tying up the kernel, and ShareFrontEnd is an extension to ShareKernel (it calls ShareKernel internally) that allows such "modeless" Java windows to interact with the front end. ShareFrontEnd is discussed in more detail in "Sharing the Front End".

In the PrintButton example that follows, a simple palette-type button is developed in Java that prints its label at the current cursor position in the active notebook. Because of current limitations with ShareFrontEnd, this example will not work with a remote kernel; the same machine must be running the kernel and the front end.

```
PrintButton[label_String] :=
    JavaBlock[
        Module[{frm, button, listener, tok},
            InstallJava[];
            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            button = JavaNew["java.awt.Button"];
            frm@add[button];
            frm@pack[];
            button@setLabel[label];
            listener = JavaNew["com.wolfram.jlink.MathActionListener",
                               "printButtonFunc"];
            button@addActionListener[listener];
            tok = ShareFrontEnd[];
            frm@onClose["UnshareFrontEnd[" <> ToString[tok] <> "]"];
            JavaShow[frm]
        ]
    ]

printButtonFunc[event_, _] :=
    JavaBlock[
        NotebookApply[SelectedNotebook[], event@getSource[]@getLabel[]];
        (* We need to explicitly release the event object, since it was
           sent to Mathematica before the JavaBlock was entered. *)
        ReleaseJavaObject[event]
    ]
```

Now invoke the PrintButton function to create and display the palette. Click the button to see the button's label (foo in this example) inserted at the current cursor location. When you are done, click the window's close box.

```
PrintButton["foo"]
```

The code is mostly straightforward. As usual, you use the `MathFrame` class for the frame window because it closes and disposes of itself when its close box is clicked. You create a `MathActionListener` that calls `buttonFunc` and you assign it to the button. From the table in the section Handling Events with *Mathematica* Code: The "MathListener" Classes, you know that `buttonFunc` will be called with two arguments, the first of which is the `ActionEvent` object. From this object you can obtain the button that was clicked and then its label, which you insert at the current cursor location using the standard `NotebookApply` function. One subtlety is that you need to specify `SelectedNotebook[]` as the target for notebook operations like `NotebookApply`, `NotebookWrite`, `NotebookPrint`, and so on, which take a notebook as an argument. Because of implementation details of `ShareFrontEnd`, the notebook given by `EvaluationNotebook[]` is not the correct target (after all, there is no evaluation currently in progress in the front end when the button is clicked).

The important thing to note in `PrintButton` is the use of `ShareFrontEnd` and `UnshareFrontEnd`. As discussed earlier, `ShareFrontEnd` puts Java into a state where it forwards everything other than the result of a computation to the front end, and puts the front end into a state where it is able to receive it. This is why the `Print` output triggered by clicking the Java button, which would normally be sent to Java (and just discarded there), appears in the front end. Front end sharing (and also kernel sharing) should be turned off when they are no longer needed, but if you are writing code for others to use you cannot just blindly shut sharing down—the user could have other Java windows open that need sharing. To handle this issue, `ShareFrontEnd` (and `ShareKernel`) works on a register/unregister principle. Every time you call `ShareFrontEnd`, it returns a token that represents a request for front end sharing. If front end sharing is not on, it will be turned on. When a program no longer needs front end sharing, it should call `UnshareFrontEnd`, passing the token from `ShareFrontEnd` as the argument. Only when all requests for sharing have been unregistered in this way will sharing actually be turned off.

The `onClose()` method of the `MathFrame` class lets you specify *Mathematica* code to be executed when the frame is closed. This code is executed after all event listeners have been notified, so it is a safe place to turn off sharing. In the `onClose()` code, you call `UnshareFrontEnd` with the token returned by `ShareFrontEnd`. Using the `onClose()` method in this way allows us to avoid writing a cleanup function that users would have to call manually after they were finished with the palette. It is not a problem to leave front end sharing turned on, but it is desirable to have your program alter the user's session as little as possible.

Now expand this example to include more buttons that perform different operations. The complete code for this example is provided in the file Palette.nb in the JLink/Examples/Part1 directory.

The first thing you do is separate the code that manages the frame containing the buttons from the code that produces a button. In this way you will have a reusable palette frame that can hold any number of different buttons. The `ShowPalette` function here takes a list of buttons, arranges them vertically in a frame window, calls `ShareFrontEnd`, and displays the frame in front of the user's notebook window.

```
ShowPalette[buttons:{__?JavaObjectQ}] :=
    JavaBlock[
        Module[{frm, tok},
            frm = JavaNew["com.wolfram.jlink.MathFrame"];
            frm@setLayout[JavaNew["java.awt.GridLayout", 0, 1]];
            frm@add[#]& /@ buttons;
            ReleaseJavaObject[buttons];
            frm@pack[];
            tok = ShareFrontEnd[];
            frm@onClose["UnshareFrontEnd[" <> ToString[tok] <> "]"];
            JavaShow[frm];
        ]
    ]
```

Note that you do not return anything from the `ShowPalette` function—specifically, you do not return the frame object itself. This is because you do not need to refer to the frame ever again. It is destroyed automatically when its close box is clicked (remember, this is a feature of the `MathFrame` class). Because you do not need to keep references to any of the Java objects you create, the entire body of `ShowPalette` can be wrapped in `JavaBlock`.

Now create a reusable `PaletteButton` function that creates a button. You have to pass in only two things: the label text you want on the button and the function (as a string) you want to have invoked when the button is clicked. This is sufficient to allow completely arbitrary button behavior, as the entire functionality of the button is tied up in the button function you pass in as the second argument.

```
PaletteButton[label_String, buttonFunc_String] :=
    JavaBlock[
        Module[{button, listener},
            button = JavaNew["java.awt.Button"];
            button@setLabel[label];
            listener = JavaNew["com.wolfram.jlink.MathActionListener", buttonFunc];
            button@addActionListener[listener];
            button
        ]
    ]
```

You will use the PaletteButton function to create four buttons. The first is just the print button just defined, the behavior of which is specified by printButtonFunc.

```
btn1 = PaletteButton["foo", "printButtonFunc"];
```

The second will duplicate the functionality of the buttons in the standard **AlgebraicManipulation** front end palette. These buttons wrap a function (e.g., Expand) around the current selection and evaluate the resulting expression in place. Here is how you create the button and define the button function for that operation.

```
btn2 = PaletteButton["Expand[■]", "applyButtonFunc"];

applyButtonFunc[event_, _] :=
    JavaBlock[
        With[{nb = SelectedNotebook[]},
            NotebookApply[nb, event@getSource[]@getLabel[], All];
            ReleaseJavaObject[event];
            SelectionEvaluate[nb]
        ];
    ]
```

The third button will create a plot. All you have to do is call a plotting function—the work of directing the graphics output to a new cell in the frontmost notebook is handled internally by *J/Lin*k as a result of having front end sharing turned on via ShareFrontEnd.

```
btn3 = PaletteButton["Create Plot", "plotButtonFunc"];

plotButtonFunc[event_, _] :=
    (
        Plot[x, {x, 0, 1}];
        ReleaseJavaObject[event];
    )
```

The final button demonstrates another method for causing text to be inserted at the current cursor location. The first example of this, printButtonFunc, uses NotebookApply. You can also just call Print—as with graphics, Print output is automatically routed to the frontmost notebook window by *J/Link* when front end sharing is on. This quick-and-easy Print method works fine for many situations when you want something to appear in a notebook window, but using NotebookApply is a more rigorous technique. You will see some differences in the effects of these two buttons if you put the insertion point into a StandardForm cell and try them.

```
btn4 = PaletteButton["foo", "printButtonFunc2"];

printButtonFunc2[event_, _] :=
    JavaBlock[
        Print[event@getSource[]@getLabel[]];
        ReleaseJavaObject[event];
    ]
```

Now you are finally ready to create the palette and show it.

```
ShowPalette[{btn1, btn2, btn3, btn4}]
```

In closing, it must be noted that although this example has demonstrated some useful techniques, it is not a particularly valuable way to use `ShareFrontEnd`. In creating a simple palette of buttons, you have done nothing that the front end cannot do all by itself. The real uses you will find for `ShareFrontEnd` will presumably involve aspects that cannot be duplicated within the front end, such as more sophisticated dialog boxes or other user interface elements.

Real-Time Algebra: A Mini-Application

This example will put together everything you have learned about modal and modeless Java user interfaces. You will implement the same "mini-application" (essentially just a dialog box) in both modal and modeless flavors. The application is inspired by the classic *MathLink* example program RealTimeAlgebra, originally written for the NeXT computer by Theodore Gray and then done in HyperCard by Doug Stein and John Bonadies. The original RealTimeAlgebra provides an input window into which the user types an expression that depends on certain parameters, an output window that displays the result of the computation, and some sliders that are used to vary the values of the parameters. The output window updates as the sliders are moved, hence the name RealTimeAlgebra. Our implementation of RealTimeAlgebra will be very simplistic, with only a single slider to modify the value of one parameter.

The complete code for this example is provided in the file RealTimeAlgebra.nb in the JLink/Examples/Part1 directory.

Here is the function that creates and displays the window.

```
CreateWindow[] :=
    Module[{frame, slider, listener},
        InstallJava[];
        (* inText and outText are globals, because we need to refer to
           them by name in the scrollFunc. This also means we must
           create them outside the JavaBlock below.
        *)
        inText = JavaNew["java.awt.TextArea", "Expand[(x+1)^a]", 8, 40];
        outText = JavaNew["java.awt.TextArea", 8, 40];
        (* This frame could be created inside the JavaBlock, because it is returned
           from the JavaBlock and therefore will not be released, but it makes
           our intentions more clear to create it outside.
        *)
        frame = JavaNew["com.wolfram.jlink.MathFrame", "RealTimeAlgebra"];
        JavaBlock[
            frame@setLayout[JavaNew["java.awt.BorderLayout"]];
            (* Note that we can refer to the Scrollbar`HORIZONTAL constant within the JavaNew
               command that first loads the Scrollbar class. Its value will not need to be
               resolved until that class has been loaded and all necessary definitions created.
            *)
            slider = JavaNew["java.awt.Scrollbar", Scrollbar`HORIZONTAL, 0, 1, 0, 20];
            frame@add[slider, ReturnAsJavaObject[BorderLayout`NORTH]];
            frame@add[outText, ReturnAsJavaObject[BorderLayout`CENTER]];
            frame@add[inText, ReturnAsJavaObject[BorderLayout`SOUTH]];
            frame@pack[];
            (* Use a fixed-width font for the output window to preserve
               formatting of multi-line expressions. *)
            outText@setFont[JavaNew["java.awt.Font", "Courier", Font`PLAIN, 12]];
            listener = JavaNew["com.wolfram.jlink.MathAdjustmentListener"];
            listener@setHandler["adjustmentValueChanged", "sliderFunc"];
            slider@addAdjustmentListener[listener];
            frame@setLocation[200, 200];
            JavaShow[frame];
        ];
        frame
    ]

(* This is what will be called in response to moving the slider position: *)

sliderFunc[evt_, type_, scrollPos_] :=
    outText@setText[
        Block[{a = scrollPos}, ToString[ToExpression[inText@getText[]]]]
    ]
```

The `sliderFunc` function is called by the `MathAdjustmentListener` whenever the slider's position changes. It gets the text in the `inputText` box, evaluates it in an environment where `a` has the value of the slider position (the range for this is 0..20, as established in the `JavaNew` call that creates the slider), and puts the resulting string into the `outText` box. It then calls `ReleaseJavaObject` to release the first argument, which is the `AdjustmentEvent` object itself. This is the only object passed in as an argument (the other two arguments are integers). If you are wondering how you determine the argument sequence for `sliderFunc`, you get it from the `MathListener` table in the section Handling Events with *Mathematica* Code: The "MathListener"

Classes. Note that you need to refer by name to the input and output text boxes in `sliderFunc`, so you cannot make their names local variables in the `Module` of `CreateWindow`, and of course they cannot be created inside that function's `JavaBlock`.

There is one interesting thing in the code that deserves a remark. Look at the lines where you add the three components to the frame. What is the `ReturnAsJavaObject` doing there? The method being called here is in the `Frame` class, and has the following signature:

```
void add(Component comp, Object constraints);
```

The second argument, `constraints`, is typed only as `Object`. The value you pass in depends on the layout manager in use, but typically it is a string, as is the case here (`BorderLayout`NORTH`, for example, is just the string "NORTH"). The problem is that *J/Link* creates a definition for this signature of `add` that expects a `JavaObject` for the second argument, and *Mathematica* strings do not satisfy `JavaObjectQ`, although they are converted to Java string objects when sent. This means that you can only pass strings to methods that expect an argument of type `String`. In the rare cases where a Java method is typed to take an `Object` and you want to pass a string from *Mathematica*, you must first create a Java `String` object with the value you want, and pass that object instead of the raw *Mathematica* string. You have encountered this issue several times before, and you have used `MakeJavaObject` as the trick to get the raw string turned into a reference to a Java `String` object. `MakeJavaObject[BorderLayout`NORTH]` would work fine here, but it is instructive to use a different technique (it also saves a call into Java). `BorderLayout`NORTH` calls into Java to get the value of the `BorderLayout.NORTH` static field, but in the process of returning this string object to *Mathematica*, it gets converted to a raw *Mathematica* string. You need the object reference, not the raw string, so you wrap the access in `ReturnAsJavaObject`, which causes the string, which is normally returned by value, to be returned in the form of a reference.

Getting back to the **RealTimeAlgebra** dialog box, you are now ready to run it as a modal window. You write a special modal version that uses `CreateWindow` internally.

```
RealTimeAlgebraModal[] :=
    JavaBlock[
        (* In the modal case, we can wrap the whole thing in JavaBlock
           and be sure that all the objects will get released, including
           the inText and outText ones needed during event handling.
        *)
        Module[{frm},
            frm = CreateWindow[];
            frm@setModal[];
            DoModal[];
        ]
    ]
```

Note that the whole function is wrapped in `JavaBlock`. This is an easy way to make sure that all object references created in *Mathematica* while the dialog is running are treated as temporary and released when `DoModal` finishes. This saves you having to properly use `JavaBlock` and `ReleaseJavaObject` in all the handler functions used for your `MathListener` objects (you will notice that these calls are absent from the `sliderFunc` function).

Now run the dialog. The `RealTimeAlgebraModal` function will not return until you close the **RealTimeAlgebra** window, which is what you mean when you call this a "modal" interface.

```
RealTimeAlgebraModal[]
```

It may take several seconds before the window appears the first time. As always, this is the one-time cost of loading all the necessary classes. Play around by dragging the slider, and try changing the text in the input box, for example, to `N[Pi, 2 a]`.

Recall that while *Mathematica* is evaluating `DoModal[]`, any `Print` output, messages, graphics, or any other output or commands other than the result of computations triggered from Java will be sent to the front end. To see this in action, try putting `Print[a]` in the input text box (you will want to arrange windows on your screen so that you can see the notebook window while you are dragging the slider). Next, try `Plot[Sin[a x], {x, 0, 4 Pi}]`.

Quit RealTimeAlgebra by clicking the window's close box. In addition to closing and disposing of the window, this causes `EndModal[]` to be executed in *Mathematica*, which then causes `DoModal` to return. The disposing of the window is due to using the `MathFrame` class for the window, and executing `EndModal[]` is the result of calling the `setModal()` method of `MathFrame`, as discussed in "Modal Windows".

Now implement RealTimeAlgebra as a modeless window. The `CreateWindow` function can be used unmodified. The only difference is how you make *Mathematica* able to service the computations triggered by dragging the slider. For a modal window, you use `DoModal` to force *Mathematica* to pay attention exclusively to the Java link. The drawback to this is that you cannot use the kernel from the notebook front end until `DoModal` ends. To allow the notebook front end and Java to share the kernel's attention, you use `ShareKernel`.

```
RealTimeAlgebraModeless[] :=
    Module[{frm, token},
        frm = CreateWindow[];

token = ShareKernel[];

(* We use the MathFrame onClose method to specify code to
           be executed when the frame is closed.The use here is
           typical--we clean up the object references that need to
           persist throughout the lifetime of the window (otherwise
           we would leak these references),and we call UnshareKernel
           to unregister this application's request for kernel sharing.
        *)
        frm@onClose[
            "ReleaseJavaObject[inText, outText]; UnshareKernel[" <> ToString[token] <> "];"
        ];

ReleaseJavaObject[frm]
    ]
```

Now run it.

```
RealTimeAlgebraModeless[]
```

`RealTimeAlgebraModeless` returns immediately after the window is displayed, leaving the front end and the **RealTimeAlgebra** window able to use the kernel for computations.

You still need a little bit of polish on the modeless version, however. First, to avoid leaking object references, you must change `sliderFunc`. With the modal version, you did not bother to use `JavaBlock` or `ReleaseJavaObject` in `sliderFunc` because you had `DoModal` wrapped in `JavaBlock`. Every call to `sliderFunc`, or any other `MathListener` handler function, occurs entirely within the scope of `DoModal`, so you can handle all object releasing at this level. With a modeless interface, you no longer have a single function call that spans the lifetime of the window. Thus, you put memory-management functions in our handler functions. Here is the new `sliderFunc`.

```
sliderFunc[evt_, type_, scrollPos_] :=
    JavaBlock[
        outText@setText[

    Block[{a = scrollPos}, ToString[ToExpression[inText@getText[]]]]
        ];
        ReleaseJavaObject[evt]
    ]
```

The `JavaBlock` here is unnecessary because the code it wraps creates no new object references. Out of habit, though, you wrap these handlers in `JavaBlock`. You need to explicitly call `ReleaseJavaObject` on `evt`, which is the `AdjustmentEvent` object, because its reference is created in *Mathematica* before `sliderFunc` is entered, so it will not be released by the `JavaBlock`. The `type` and `scrollPos` arguments are integers, not objects.

Try setting the input text to `Print[a]`. Notice that nothing appears in the front end when you move the slider, in contrast to the modal case. With a modeless interface, the Java link is the kernel's `$ParentLink` during the times when the kernel is servicing a request initiated from the Java side. Thus, the output from `Print` and graphics goes to Java, not the notebook front end. (The Java side ignores this output, in case you are wondering.) To get this output sent to the front end instead, use `ShareFrontEnd`.

```
ShareFrontEnd[];
```

Now if you set the input text to, say, `Print[a]` or `Plot[a x, {x, 0, a}]`, you will see the text and graphics appearing in the front end.

When you are finished, quit `RealTimeAlgebra` by clicking its close box. Then turn off front end sharing that was turned on in the previous input.

```
UnshareFrontEnd[]
```

A modal interface is simpler than a modeless one in terms of how it uses *Mathematica*, and is therefore the preferred method unless you specifically need the modeless attribute. `ShareKernel` and `ShareFrontEnd` are complex functions that put the kernel into an unusual state. They work fine, but do not use them unnecessarily.

GraphicsDlg: Graphics and Typeset Output in a Window

It is useful to be able to display *Mathematica* graphics and typeset expressions in your Java user interface, and this is easy to do using *J/Link*'s `MathCanvas` class. This example demonstrates a simple dialog box that allows the user to type in a *Mathematica* expression and see the output in the form of a picture. If the expression is a plotting or other graphics function, the resulting image is displayed. If the expression is not a graphic, then it is typeset in `TraditionalForm` and displayed as a picture. The example is first presented in modal form and then in modeless form using `ShareKernel` and `ShareFrontEnd`.

This example also demonstrates a trivial example of using Java code that was created by a drag-and-drop GUI builder of the type present in most Java development environments. For layout of simple windows, it is easy enough to do everything from *Mathematica*. This method was chosen for all the examples in this tutorial, writing no Java code and instead scripting the creation and layout of controls in windows with *Mathematica* calls into Java. This has the advantage of not requiring any Java classes to be written and compiled. For more complex windows,

however, you will probably find it much easier to create the controls, arrange them in position, set their properties in a GUI builder, and let it generate Java code for you. You might also want to write some additional Java code by hand.

If you choose this route, the question becomes, "How do I connect the Java code thus generated with *Mathematica*?" Any public fields or methods can be called directly from *Mathematica*, but your GUI builder may not have made public all the ones you need to use. You could make these fields and methods public or add some new public methods that expose them. The latter approach is probably preferable since it does not involve modifying the code that the GUI builder wrote, which could confuse the builder or cause it to overwrite your changes in future modifications.

The complete code for this example is provided in the JLink/Examples/Part1/GraphicsDlg directory. Some of the code is in Java.

This example uses the GUI builder in the WebGain Visual Café Java development environment. The builder was used to create a frame window with three controls. The frame window was made to be a subclass of `MathFrame` because you want to inherit the `setModal()` method. In the top left is an AWT `TextArea` that serves as the input box for the expression. To its right is an **Evaluate** button. Occupying the rest of the window is a `MathCanvas`.

Up to this point, no code has been written by hand at all—everything has been done automatically as components were dropped into the frame and their properties set. All that is left to do is to wire up the button so that when it is clicked the input text is taken and supplied as to the `MathCanvas` via its `setMathCommand()` method. You could write that code in Java, using Visual Café's Interaction Wizard to wire up this event (similar facilities exist in other Java GUI builders). You would have to write some Java code by hand, as the code's logic is more complex than can be handled by graphical tools for creating event handlers.

Rather than doing that, move to *Mathematica* to script the rest of the behavior because it is easier and more flexible. You will need to access the `TextArea`, `Button`, and `MathCanvas` objects from *Mathematica*, but the GUI builder made these nonpublic fields of the frame class. Thus, you need to add three public methods that return these objects to the frame class.

```
public Button getEvalButton()       {return evalButton;}
public TextArea getInputTextArea()  {return inputTextArea;}
public MathCanvas getMathCanvas()   {return mathCanvas;}
```

That is all you need to do to the Java code created by the GUI builder.

The GUI builder created a subclass of `MathFrame` that is named `GraphicsDlg`. It also gave it a `main()` method that does nothing but create an instance of the frame and make it visible. You will not bother with the `main()` method, choosing instead to do those two steps manually, since you need a reference to the frame.

Needed before the code is run is a demonstration of one more feature of *J/Link*—the ability to add directories to the class search path dynamically. You need to load the Java classes for this example, but they are not on the Java class path. With *J/Link*, you can add the directory in which the classes reside to the search path by calling `AddToClassPath`. This will work exactly as written in *Mathematica* 4.2 and later. You will need to modify the path if you have an earlier version of *Mathematica*.

```
classDir = ToFileName[{$TopDirectory, "SystemFiles", "Links", "JLink",
                       "Examples", "Part1", "GraphicsDlg"}];
InstallJava[];
AddToClassPath[classDir];
```

Here is the first implementation of the *Mathematica* code to create and run the graphics dialog. This runs the dialog in a modal loop.

```
DoGraphicsDialogModal[] :=
    JavaBlock[
        Module[{frm, btn, listener},
            InstallJava[];
            (* We named the MathFrame subclass GUI builder created "MyFrame". *)
            frm = JavaNew["GraphicsDlg"];
            (* Here we call one of the accessor methods we had to add
               by hand to the GraphicsDlg class.
            *)
            btn = frm@getEvalButton[];
            listener = JavaNew["com.wolfram.jlink.MathActionListener"];
            listener@setHandler["actionPerformed", "btnFunc"];
            btn@addActionListener[listener];
            JavaShow[frm];
            frm@setModal[];
            DoModal[]
        ]
    ]

btnFunc[event_, _] :=
    JavaBlock[
        Module[{frm, expr, textArea, inputText, mathCanvas},
            frm = event@getSource[]@getParent[];
            (* Here we call two of the accessor methods we had to add
               by hand to the GraphicsDlg class.
            *)
            textArea = frm@getInputTextArea[];
            mathCanvas = frm@getMathCanvas[];

            inputText = textArea@getText[];
            (* We have to evaluate the expression ahead of time to determine
               whether it is a graphics object or not, so we can decide
               whether it display it as a plot or as a typeset result.
            *)
            expr = Block[{$DisplayFunction = Identity}, ToExpression[inputText]];
            If[MatchQ[expr, _Graphics | _Graphics3D | _SurfaceGraphics |
                                DensityGraphics | _ContourGraphics],
                mathCanvas@setImageType[MathCanvas`GRAPHICS],
            (* else *)
                mathCanvas@setImageType[MathCanvas`TYPESET];
                mathCanvas@setUsesTraditionalForm[True]
            ];
            mathCanvas@setMathCommand[ToString[expr, InputForm]];
            ReleaseJavaObject[event]
        ]
    ]
```

As mentioned in the section "Creating Windows and Other User Interface Elements" only the notebook front end can perform the feat of taking a typeset (i.e., "box") expression and creating a graphical representation of it. Thus, the `MathCanvas` can render typeset expressions provided that it has a front end available to farm out the chore of creating the appropriate representation. The front end is used to run this example, but it is really because you are running the Java dialog "modally" that everything works the way it does. All the while the dialog is up, the front end is waiting for a result from a computation (`DoModal[]`), and therefore it is receptive to requests from the kernel for various services. As far as the front end is con-

cerned, the code for DoModal invoked the request for typesetting, even though it was actually triggered by clicking a Java button.

Now run the dialog.

```
DoGraphicsDialogModal[]
```

What if you are not happy with the restriction of running the dialog modally? Now you want to have the dialog remain open and active while not interfering with normal use of the kernel from the front end. As discussed in "Modal Windows" and "Real-Time Algebra: A Mini-Application", you get a lot of useful behavior regarding the front end for free when you run your Java user interface modally. One of these features is that the front end is kept receptive to the various sorts of requests the kernel can send to it (such as for typesetting services). You know you can run a Java user interface in a "modeless" way by using ShareKernel, but then you give up the ability to have the kernel use the front end during computations initiated by actions in Java. Luckily, the ShareFrontEnd function exists to restore these features for modeless windows.

Re-implement the graphics dialog in modeless form.

```
DoGraphicsDialogModeless[] :=
    JavaBlock[
        Module[{frm, btn, listener, tok},
            InstallJava[];
            frm = JavaNew["GraphicsDlg"];
            btn = frm@getEvalButton[];
            listener = JavaNew["com.wolfram.jlink.MathActionListener"];
            listener@setHandler["actionPerformed", "btnFunc"];
            btn@addActionListener[listener];
            tok = ShareFrontEnd[];
            frm@onClose["UnshareFrontEnd[" <> ToString[tok] <> "]"];
            JavaShow[frm]
        ]
     ]
```

The code shown is exactly the same as DoGraphicsDialogModal except for the last few lines. You call ShareFrontEnd here instead of setModal and DoModal. That is the only difference—the rest of the code (including btnFunc) is exactly the same. Notice also that you use the onClose() method of MathCanvas to execute code that unregisters the request for front end sharing when the window is closed.

Run the modeless version. Note how you can continue to perform computations in the front end while the window is active.

```
DoGraphicsDialogModeless[]
```

This new version functions exactly like the modeless version except that it does not leave the front end hanging in the middle of a computation. It is interesting to contrast what happens if you turn off front end sharing (but you need to leave kernel sharing on or the Java dialog will break completely). You can do this by replacing `ShareFrontEnd` and `UnshareFrontEnd` in `DoGraphicsDialogModeless` with `ShareKernel` and `UnshareKernel`. Now if you use the dialog you will find that it fails to render typeset expressions, producing just a blank window, but it still renders graphics normally (unless they have some typeset elements in them, such as a plot label). All the functionality is kept intact except for the ability of the kernel to make use of the front end for typesetting services.

BouncingBalls: Drawing in a Window

This example demonstrates drawing in Java windows using the Java graphics API directly from *Mathematica*. It also demonstrates the use of the `ServiceJava` function to periodically allow event handler callbacks into *Mathematica* from Java. The issues surrounding `ServiceJava` and how it compares to `DoModal` and `ShareKernel` are discussed in greater detail in "Manual" Interfaces: The ServiceJava Function.

The full code is a little too long to include here in its entirety, but it is available in the sample file BouncingBalls.nb in the JLink/Examples/Part1 directory. Here is an excerpt that demonstrates the use of `ServiceJava`.

```
...
mwl = JavaNew["com.wolfram.jlink.MathWindowListener"];
mwl@setHandler["windowClosing", "(keepOn = False)&"];
mathCanvas@addWindowListener[mwl];
keepOn = True;
While[keepOn,
    g@setColor[bkgndColor];
    g@fillRect[0, 0, 300, 300];
    drawBall[g, #]& /@ balls;
    mathCanvas@setImage[offscreen];
    balls = recomputePosition /@ balls;
    ServiceJava[]
];
...
```

A `MathWindowListener` is used to set `keepOn = False` when the window is closed, which will cause the loop to terminate. While the window is up, mouse clicks will cause new balls to be

created, appended to the balls list, and set in motion. This is done with a MathMouseListener (not shown in the code). Thus, *Mathematica* needs to be able to handle calls originating from user actions in Java. As discussed in the section "Creating Windows and Other User Interface Elements", there are three ways to enable *Mathematica* to do this: DoModal (modal interfaces), ShareKernel or ShareFrontEnd (modeless interfaces), and ServiceJava (manual interfaces). A modal loop via DoModal would not be appropriate here because the kernel needs to be computing something at the same time it is servicing calls from Java (it is computing the new positions of the balls and drawing them). ShareKernel would not help because that is a way to give Java access to the kernel *between* computations triggered from the front end, not *during* such computations.

You need to periodically point the kernel's attention at Java to service requests if any are pending, then let the kernel get back to its other work. The function that does this is ServiceJava, and the code above is typical in that it has a loop that calls ServiceJava every time through. The calls from Java that ServiceJava will handle are the ones from mouse clicks to create new balls and when the window is closed.

Spirograph

This example is just a little fun to create an interesting, nontrivial application—an implementation of a simple Spirograph-type drawing program. It is run as a modal window, and it demonstrates drawing into a Java window from *Mathematica,* along with a number of MathListener objects for various event callbacks. It uses the Java Graphics2D API, so it will not run on systems that have only a Java 1.1.x runtime.

The code for this example can be found in the file Spirograph.nb in the JLink/Examples/Part1 directory.

One of the things you will notice is that on a reasonably fast machine, the speed is perfectly acceptable. There is nothing to suggest that the entire functionality of the application is scripted from *Mathematica*. It is very responsive despite the fact that a large number of callbacks to *Mathematica* are triggered. For example, the cursor is changed as you float the mouse over various regions of the window (it changes to a resize cursor in some places), so there is a constant flow of callbacks to *Mathematica* as you move the mouse. This example demonstrates the feasibility of writing a sophisticated application entirely in *Mathematica*.

This application was written in *Mathematica*, but it could also have been written entirely in Java, or a combination of Java and *Mathematica*. An advantage of doing it in *Mathematica* is that you generally can be much more productive. Spirograph would have taken at least twice as long to write in Java. It is invaluable to be able to write and test the program a line at a time, and to debug and modify it while it is running. Even if you intend to eventually port the code to pure Java, it can be very useful to begin writing it in *Mathematica*, just to take advantage of the scripting mode of development.

Modal programs like this are best developed using `ShareFrontEnd`, then made modal only when they are complete. Making it modeless while it is being developed is necessary to be able to build and debug it interactively, because while it is running you can continue to use the front end to modify the code, make new definitions, add debugging statements, and so on. Using `ShareFrontEnd` instead of `ShareKernel` for modeless operation lets *Mathematica* error and warning messages generated by event callbacks, and `Print` statement inserted for debugging, show up in the notebook window. Only when everything is working as desired do you add the `DoModal[]` call to turn it into a modal window.

A Piano Keyboard

With the inclusion of the Java Sound API in Java 1.3 and later, it becomes possible to write Java programs that do sophisticated things with sound, such as playing MIDI instruments. The Piano.nb example in the JLink/Examples/Part1 directory displays a keyboard and lets you play it by clicking the mouse. A popup menu at the top lists the available MIDI instruments. This example was created precisely because it is so far outside the limitations of traditional *Mathemat ica* programming. Using *J/Link*, you can actually write a short and completely portable program, entirely in the *Mathematica* language, that displays a MIDI keyboard and lets you play it! With just a little more work, the code could be modified to record a sequence played and then return it to *Mathematica*, where you could manipulate it by transposing, altering the tempo, and so on.

Writing Java Programs That Use Mathematica

Introduction

The first part of this User Guide describes using *J/Link* to allow you to call from *Mathematica* into Java, thereby extending the *Mathematica* environment to include the functionality in all existing and future Java classes. This part shows you how to use *J/Link* in the opposite direction, as a means to write Java programs that use the *Mathematica* kernel as a computational engine.

J/Link uses *MathLink*, Wolfram Research's protocol for sending data and commands between programs. Many of the concepts and techniques in *J/Link* programming are the same as those for programming with the *MathLink* C-language API. The *J/Link* documentation is not intended to be an encyclopedic compendium of everything you need to know to write Java programs that use *MathLink*. Programmers may have to rely a little on the general documentation of *MathLink* programming. Many of the functions *J/Link* provides have C-language counterparts that are identical or nearly so.

If you have not read "Calling Java from *Mathematica*", you should at least skim it at some point. Your Java "front end" can use the same techniques for calling Java methods from *Mathematica* code and passing Java objects as arguments that programmers use when running the kernel from the notebook front end. This allows you to have a very high-level interface between Java and *Mathematica*. When you are writing *MathLink* programs in C, you have to think about passing and returning simple things like strings and integers. With *J/Link* you can pass Java objects back and forth between Java and *Mathematica*. *J/Link* truly obliterates the boundary between Java and *Mathematica*.

Although Java is quite useful as a web-related development language, such as for writing applets or servlets, it is by no means restricted to this domain, and this User Guide does not address the specifics of these types of programs. Addressed are generic issues in *J/Link* programming, leaving the application of these concepts in specific types of programs up to the reader. In other words, if you are looking for a worked-out example program showing how a suite of servlets running on a web or application server can allow remote Java clients to share a cluster of *Mathematica* kernels, you will not find it here. But *J/Link* is just the sort of tool you will need if you embark on such a project.

This half of the User Guide is organized as follows. "What Is *MathLink*?" is a very brief introduction to *MathLink*. The section "Preamble" introduces the most important *J/Link* interfaces and classes. "Sample Program" presents a simple example program. "Creating Links with MathLink-Factory" shows how to launch *Mathematica* and create links. "The *MathLink* Interface" and "The KernelLink Interface" give a listing of methods in the large and all-important `MathLink` and `KernelLink` interfaces. The methods are grouped by function, and there is some commentary mixed in. This treatment does not replace the actual JavaDoc help files for *J/Link*, found in the JLink/Documentation/JavaDoc directory. The JavaDoc files are the main method-by-method reference for *J/Link*, and they include all the classes and interfaces that programmers will use. The remaining sections of this User Guide present discussions of a number of important topics in *J/Link* programming, including how to handle exceptions and get graphics and typeset output.

When you are reading this text, or programming in Java or *Mathematica*, remember that the entire source code for *J/Link* is provided. If you want to see how anything works (or why it does not), you can always consult the source code directly.

What Is MathLink?

MathLink is a platform-independent protocol for communicating between programs. In more concrete terms, it is a means to send and receive *Mathematica* expressions. *MathLink* is the means by which the notebook front end and kernel communicate with each other. It is also used by a large number of commercial and freeware applications and utilities that link *Mathematica* and other programs or languages.

MathLink is implemented as a library of C-language functions. Using it from another language (such as Java) typically requires writing some type of "glue" code that translates between the data types and calling conventions of that language and C. At the core of *J/Link* is just such a translation layer—a library built using Java's JNI (Java Native Interface) specification.

Throughout this part of the User Guide, the term *MathLink* will be used in two ways—as a generic term for the capability *Mathematica* has to communicate with other programs, and as the name for one specific *J/Link* interface. It will always be written in the special font used for Java names when the interface name is being used.

Overview of the Main J/Link Interfaces and Classes

Preamble

The *J/Link* classes are written in an object-oriented style intended to maximize their extensibility in the future without requiring users' code to change. This requires a clean separation between interface and implementation. This is accomplished by exposing the main link functionality through interfaces, not classes. The names of the concrete classes that implement these interfaces will hardly be mentioned because programmers do not need to know or care what they are. Rather, you will use objects that belong to one of the interface types. You do not need to know what the actual classes are because you will never create an instance directly; instead, you use a "factory method" to create an instance of a link class. This will become clear further on.

MathLink and KernelLink

The two most important link interfaces you need to know about are `MathLink` and `KernelLink`. The `MathLink` interface is essentially a port of the *MathLink* C API into Java. Most of the method names will be familiar to experienced *MathLink* programmers. `KernelLink` extends `MathLink` and adds some important high-level convenience methods that are only meaningful if the other side of the link is a *Mathematica* kernel (for example, the method `waitForAnswer()`, which assumes the other side of the link will respond with a defined series of packets).

The basic idea is that the `MathLink` interface encompasses all the operations that can be performed on a link without making any assumptions about what program is on the other side of the link. `KernelLink` adds the assumption that the other side is a *Mathematica* kernel. In the future, other interfaces could be added that also extend `MathLink` and encapsulate other conven tions for communicating over a link.

`KernelLink` is the most important interface, as most programmers will work exclusively with `KernelLink`. Of course, since `KernelLink` extends `MathLink`, many of the methods you will use on your `KernelLink` objects are declared and documented in the `MathLink` interface.

The most important class that implements `MathLink` is `NativeLink`, so named because it uses native methods to call into Wolfram Research's *MathLink* library. In the future, other classes could be added that do not rely on native methods—for example, one that uses RMI to communicate across a network. As discussed above, most programmers do not need to be concerned about what these classes are, because they will never type a link class name in their code.

MathLinkFactory

MathLinkFactory is the class that you use to create link objects. It contains the static methods createMathLink(), createKernelLink(), and createLoopbackLink(), which take various argument sequences. These are the equivalents of calling MLOpen in a C program. The MathLinkFactory methods are discussed in detail in "Creating Links with MathLinkFactory".

MathLinkException

MathLinkException is the exception class that is thrown by many of the methods in MathLink and KernelLink. The *J/Link* API uses exceptions to indicate errors, rather than function return values like the *MathLink* C API. In C, you write code that checks the return values like this:

```
// C code
if (!MLPutInteger(link, 42)) {
    // was error; print message and clean up.
}
```

In *J/Link*, you wrap *MathLink* calls in a try block and catch MathLinkException.

Expr

The Expr class provides a direct representation of *Mathematica* expressions in Java. Expr has a number of methods that provide information about the structure of the expression and that let you extract components. These methods have names and behaviors that will be familiar to *Mathematica* programmers—for example, length(), part(), numberQ(), vectorQ(), take(), delete(), and so on. When reading from a link, instead of using the low-level MathLink interface methods for discovering the structure and properties of the incoming expression, you can just read an entire expression from the link using getExpr(), and then use Expr methods to inspect it or decompose it. For writing to a link, Expr objects can be used as arguments to some of the most important KernelLink methods. The Expr class is discussed in detail in "Motivation for the Expr Class".

PacketListener

A central component of a standard C *MathLink* program is a packet-reading loop, which typically consists of calling the *MathLink* API functions MLNextPacket and MLNewPacket until a desired packet is encountered. *J/Link* programs will typically not include such a loop—instead, you call the KernelLink methods waitForAnswer() or discardAnswer(), which hide the packet loop

within them. Not only is this a convenience to avoid having to put the same boilerplate code into every program, it is necessary since in some circumstances programmers *cannot* write a correct packet because special packets may arrive that *J/Link* needs to handle internally. It is therefore necessary to hide the details of the packet loop from programmers. In some cases, though, programmers will want to observe and/or operate on the incoming flow of packets. A typical example would be to display `Print` output or messages generated by a computation. These outputs are side effects of a computation and not the "answer", and they are normally discarded by `waitForAnswer()`.

To accommodate this need, `KernelLink` objects fire a `PacketArrivedEvent` for each packet that is encountered while running an internal packet loop. You can register your interest in receiving notifications of these packets by creating a class that implements the `PacketListener` interface and registering an object of this class with the `KernelLink` object. The `PacketListener` interface has only one method, `packetArrived()`, which will be called for each packet. Your `packetArrived()` method can consume or ignore the packet *without affecting the internal packet loop in any way*. Very advanced programmers can optionally indicate that the internal packet loop should not see the packet.

The `PacketListener` interface is discussed in greater detail in "Using the PacketListener Interface".

High-Level User Interface Classes

J/Link includes several classes that are useful for creating programs that have user interfaces. The `MathCanvas` and `MathGraphicsJPanel` classes provide an easy way to display *Mathematica* graphics and typeset expressions. These classes are often used from *Mathematica* code, as described in "The MathCanvas and MathGraphicsJPanel Classes", but they are just as useful in Java programs. They are discussed in "MathCanvas and MathGraphicsJPanel". The various "MathListener" classes ("Handling Events with *Mathematica* Code: The 'MathListener' Classes") can be be used from Java code to trigger evaluations in *Mathematica* when user interface actions occur.

New in *J/Link* 2.0 are the classes in the com.wolfram.jlink.ui package. These classes provide some very high-level user interface elements. There is the `ConsoleWindow` class, which gives you a console output window (this is the class used to implement the *Mathematica* function `ShowJavaConsole`, discussed in "The Java Console Window"). The `InterruptDialog` class gives

you an **Interrupt Evaluation** dialog similar to the one you see in the notebook front end when you choose **Interrupt Evaluation** from the **Evaluation** menu. The `MathSessionPane` class provides an `In/Out` *Mathematica* session window complete with a full set of editing functions including cut/copy/paste/undo/redo, support for graphics, syntax coloring, and customizable font styles. The auxiliary classes `SyntaxTokenizer` and `BracketMatcher` are used by `MathSessionPane` but can also be used separately to provide these services in your own programs. All these classes are discussed in the section "Some Special User Interface Classes: Introduction".

Sample Program

Here is a basic Java program that launches the *Mathematica* kernel, uses it for some computations, and then shuts it down. This program is provided in source code and compiled form in the JLink/Examples/Part2 directory. The usual *MathLink* arguments including the path to the kernel are given on the command line you use to launch the program, and some typical examples are given below. You will have to adjust the *Mathematica* kernel path for your system. If you have your `CLASSPATH` environment variable set to include `JLink.jar`, then you can leave off the `-classpath` specification in these command lines. It is assumed that these commands are executed from the JLink/Examples/Part2 directory.

```
(Windows)
java -classpath .;..\..\JLink.jar SampleProgram -linkmode launch -linkname
"c:\program files\wolfram research\mathematica\6.0\mathkernel.exe"

(Unix)
java -classpath .:../../JLink.jar SampleProgram -linkmode launch -linkname
'math -mathlink'

(Mac OS X from a terminal window)
java -classpath .:../../JLink.jar SampleProgram -linkmode launch -linkname
'"/Applications/Mathematica.app/Contents/MacOS/MathKernel" -mathlink'
```

Here is the code from `SampleProgram.java`. This program demonstrates launching the kernel with `MathLinkFactory.createKernelLink()`, and several different ways to send computations to *Mathematica* and read the result.

```
import com.wolfram.jlink.*;

public class SampleProgram {

    public static void main(String[] argv) {

        KernelLink ml = null;

        try {
            ml = MathLinkFactory.createKernelLink(argv);
        } catch (MathLinkException e) {
            System.out.println("Fatal error opening link: " +
e.getMessage());
            return;
        }

        try {
            // Get rid of the initial InputNamePacket the kernel will send
            // when it is launched.
            ml.discardAnswer();

            ml.evaluate("<<MyPackage.m");
            ml.discardAnswer();

            ml.evaluate("2+2");
            ml.waitForAnswer();

            int result = ml.getInteger();
            System.out.println("2 + 2 = " + result);

            // Here's how to send the same input, but not as a string:
            ml.putFunction("EvaluatePacket", 1);
            ml.putFunction("Plus", 2);
            ml.put(3);
            ml.put(3);
            ml.endPacket();
            ml.waitForAnswer();
            result = ml.getInteger();
```

```
            System.out.println("3 + 3 = " + result);

            // If you want the result back as a string, use
evaluateToInputForm
            // or evaluateToOutputForm. The second arg for either is the
            // requested page width for formatting the string. Pass 0 for
            // PageWidth->Infinity. These methods get the result in one
            // step--no need to call waitForAnswer.
            String strResult = ml.evaluateToOutputForm("4+4", 0);
            System.out.println("4 + 4 = " + strResult);

        } catch (MathLinkException e) {
            System.out.println("MathLinkException occurred: " +
e.getMessage());
        } finally {
            ml.close();
        }
    }
}
```

Creating Links with MathLinkFactory

To isolate clients of the *J/Link* classes from implementation details it is required that clients never explicitly name a link class in their code. This means that programs will never call `new` to create an instance of a link class. Instead, a so-called "factory method" is supplied that creates an appropriate instance for you, based on the arguments you pass in. This factory method takes the place of calling `MLOpen` in a C program.

The method that creates a `KernelLink` is a static method called `createKernelLink()` in the `MathLinkFactory` class:

```
public static KernelLink createKernelLink(String cmdLine) throws
MathLinkException

public static KernelLink createKernelLink(String[] argv) throws
MathLinkException

. . . plus a few more of limited usefulness
```

There are also two functions called `createMathLink()` that take the same arguments but create a `MathLink` instead of a `KernelLink`. Very few programmers will need to use `createMathLink()` because the only reason to do so is if you are connecting to a program other than the *Mathematica* kernel. See the JavaDoc files for a complete listing of the methods.

The second signature of `createKernelLink()` is convenient if you are using the command-line parameters that your program was launched with, which are, of course, provided to your `main()` function as an array of strings. An example of this use can be found in the sample program in the section "Sample Program". Other times it will be convenient to specify the parameters as a single string, for example:

```
KernelLink ml = MathLinkFactory.createKernelLink("-linkmode launch
-linkname 'c:\\program files\\wolfram
research\\mathematica\\6.0\\mathkernel'");
```

Note that the linkname argument is wrapped in single quotation marks (`'`). This is because *MathLink* parses this string as a complete command line, and wrapping it in single quotation marks is an easy way to force it to be seen as just a filename. Also note that it is required to type two backslashes to indicate a Windows directory separator character when you are typing a literal string in your Java code because Java, like C and *Mathematica*, treats the `\` as a meta-character that quotes the character following.

Here are some typical arguments for `createKernelLink()` on various platforms when given as a single string. Note the use of quote characters (' and "):

```
// Typical launch on Windows
KernelLink ml = MathLinkFactory.createKernelLink("-linkmode launch
-linkname 'c:\\program files\\wolfram
research\\mathematica\\mathkernel.exe'");

// Typical launch on Unix
KernelLink ml = MathLinkFactory.createKernelLink("-linkmode launch
-linkname 'math -mathlink'");

// Typical launch on Mac OS X
KernelLink ml = MathLinkFactory.createKernelLink("-linkmode launch
-linkname '\"/Applications/Mathematica.app/Contents/MacOS/MathKernel\"
-mathlink'");

// Typical "listen" link on any platform:
KernelLink ml = MathLinkFactory.createKernelLink("-linkmode listen
-linkname 1234 -linkprotocol tcp");

// Windows can use the default protocol for listen/connect links:
KernelLink ml = MathLinkFactory.createKernelLink("-linkmode listen
-linkname foo");
```

Here are typical arguments for createKernelLink() when given as an array of strings:

```
// Typical launch on Windows:
String[] argv = {"-linkmode", "launch", "-linkname", "c:\\program
files\\wolfram research\\mathematica\\6.0\\mathkernel"};

// Typical launch on UNIX:
String[] argv = {"-linkmode", "launch", "-linkname", "math -mathlink"};

// Typical launch on Mac OS X:
String[] argv = {"-linkmode", "launch", "-linkname",
"\"/Applications/Mathematica.app/Contents/MacOS/MathKernel\" -mathlink"};

// Typical "listen" link on any platform:
String[] argv = {"-linkmode", "listen", "-linkname", "1234", "-
linkprotocol", "tcp"};

// Windows can use the default protocol for listen/connect links:
String[] argv = {"-linkmode", "listen", "-linkname", "foo"};
```

The arguments for createKernelLink() and createMathLink() (e.g., -linkmode, -linkprotocol, and so on) are identical to those used for MLOpen in the *MathLink* C API. Consult the *MathLink* documentation for more information.

The createKernelLink() and createMathLink() methods will always return a link object that is not null or throw a MathLinkException. You do not need to test whether the returned link is null. Because these methods throw a MathLinkException on failure, you need to wrap the call in a try block:

```
KernelLink ml = null;
try {
    ml = MathLinkFactory.createKernelLink("-linkmode launch -linkname
'c:\\program files\\wolfram research\\mathematica\\6.0\\mathkernel'");
} catch (MathLinkException e) {
    // This is equivalent to MLOpen returning NULL in a C program.
    System.out.println(e.getMessage());
    System.exit(1);
}
```

The fact that createKernelLink() succeeds does not mean that the link is connected and functioning properly. There are a lot of things that could be wrong. For example, if you launch a program that knows nothing about *MathLink*, createKernelLink() will still succeed. There is a

difference between creating a link (which involves setting up your side) and connecting one (which verifies that the other side is alive and well).

If a link has not been connected yet, *MathLink* will automatically try to connect it the first time you try to read or write something. Alternatively, you can call the `connect()` method to explicitly connect the link after creating it. If the link cannot be connected, then the attempt to connect, whether made explicitly by you or internally by *MathLink*, will fail or even hang indefinitely. It can hang because the attempt to connect will block until the connection succeeds or until it detects a fatal problem with the link. In some cases, neither will happen—for example, if you mistakenly launch a program that is not *MathLink*-aware. Dealing with blocking in *J/Link* methods is discussed more thoroughly later, but in the case of connecting the link you have an easy solution. The `connect()` method has a second signature that takes a `long` argument specifying the number of milliseconds to wait before abandoning the attempt to connect: `connect(long timeoutMillis)`. You do not need to explicitly call `connect()` on a link—it will be connected for you the first time you try to read something. You can use a call to `connect()` to catch failures at a well-defined place, or if you want to use the automatic time out feature. Here is a code fragment that demonstrates how to implement a time out in `connect()`.

```
KernelLink ml = null;
try {
    ml = MathLinkFactory.createKernelLink("-linkmode launch -linkname
'c:\\program files\\wolfram research\\mathematica\\6.0\\mathkernel'");
} catch (MathLinkException e) {
    System.out.println("Link could not be created: " + e.getMessage());
    return; // Or whatever is appropriate.
}

try {
    connect(10000); // Wait at most 10 seconds
} catch (MathLinkException e) {
    // If the timeout expires, a MathLinkException will be thrown.
    System.out.println("Failure to connect link: " + e.getMessage());
    ml.close();
    return; // Or whatever is appropriate.
}
```

When you are finished with a link, call its `close()` method. Although the finalizer for a link object will close the link, you cannot guarantee that the finalizer will be called in a timely fashion, or even at all, so you should always manually close a link when you are done.

Using Listen and Connect Modes

You can use the listen and connect linkmodes, instead of launch, if you want to connect to an already-running program. Using listen and connect linkmodes in *J/Link* works in the same way as with C *MathLink* programs. See the *MathLink* Tutorial (http://library.wolfram.com/infocenter/TechNotes/174/) or "*MathLink* and External Program Communication" for more information.

Using a Remote Kernel

To attach a remote *Mathematica* kernel to a *J/Link* program, open the link using the listen/connect style. On the remote Unix machine, launch *Mathematica* and have it listen on a link by executing the following on a command line.

```
math -mathlink -linkmode listen -linkname 1234 -linkprotocol tcpip
```

Note the use of the TCPIP *MathLink* protocol. The TCPIP protocol is an improved version of the TCP protocol that is only supported in *Mathematica* 5.0 and later. If you are launching *Mathematica* 4.x, use `tcp` as the protocol name instead of `tcpip` (also in the following line).

Then in your Java program:

```
KernelLink ml = MathLinkFactory.createKernelLink("-linkmode connect
-linkprotocol tcpip -linkname 1234@remotemachinename");
```

The drawback to the listen/connect technique is that you must manually log into the remote machine and launch *Mathematica*. You can have the Java program automatically launch *Mathematica* on the remote machine by using an `rsh` or `ssh` client program. Unix machines have `rsh` and `ssh` built in, and *Mathematica* ships with the `winrsh` client program for Windows. Here is an example of using `winrsh` to launch and connect to *Mathematica* on a remote Unix machine.

```
KernelLink ml = MathLinkFactory.createKernelLink("-linkmode listen
-linkprotocol tcpip -linkname 1234");
Runtime.exec("c:\\program files\\wolfram
research\\mathematica\\6.0\\systemfiles\\frontend\\binaries\\windows\\winrs
h -m -q -h -l YourUsername -'math -mathlink -linkmode connect
-linkprotocol tcpip -linkname 1234@localmachinename'");
```

The MathLink Interface

`MathLink` is the low-level interface that is the root of all link objects in *J/Link*. The methods in `MathLink` correspond roughly to a subset of those in the C-language *MathLink* API. Most programmers will deal instead with objects of type `KernelLink`, a higher-level interface that extends `MathLink` and incorporates the assumption that the program on the other side of the link is a *Mathematica* kernel.

There will not be much said here about most of these methods, as they behave like their C API counterparts in most respects. *The JavaDoc help files are the main method-by-method documentation for all the J/Link classes and interfaces*. They can be found in the JLink/Documentation/JavaDoc directory. This section is provided mainly for those who want to skim a traditional printed listing.

These are all public methods (the `public` has been left off to keep lines short).

Managing Links

```
void close();

void connect() throws MathLinkException;

// Wait at most timeoutMillis for the connect to occur, then throw a
MathLinkException
void connect(long timeoutMillis) throws MathLinkException;

//A synonym for connect. This is the newer name.
void activate() throws MathLinkException;
```

Packet Functions

```
//Does not throw exception because it will often be needed in a catch
block.
void newPacket();

int nextPacket() throws MathLinkException;

void endPacket() throws MathLinkException;
```

Error Handling

```
int error();

boolean clearError();

String errorMessage();

void setError(int err);
```

Link State

```
boolean ready() throws MathLinkException;
```

Putting

Putting expressions on the link is a bit different in Java than C because Java lets you overload functions. Thus, there is no need to have methods with names like the C functions `MLPutInteger` and `MLPutDouble`; it suffices to have a single function named `put()` that has different definitions for each argument type. The only exceptions to this are the few cases where the argument needs to be interpreted in a special way. For example, there are three "put" methods that take a single string argument: `put()` (equivalent to the C-language function `MLPutUCS2String`), `putSymbol()` (equivalent to `MLPutUCS2Symbol`), and `putByteString()` (equivalent to `MLPutByteString`).

For numeric types, there are the following methods (there is no need to provide a `put()` method for `byte`, `char`, and `short` types, as these can be automatically promoted to `int`):

```
void put(int i) throws MathLinkException;

void put(long i) throws MathLinkException;

void put(double d) throws MathLinkException;
```

For strings and symbols:

```
void put(String s) throws MathLinkException;

void putByteString(byte[] b) throws MathLinkException;

void putSymbol(String s) throws MathLinkException;
```

All the *J/Link* methods that put or get strings use Unicode, which is the native format for Java strings.

For Booleans, a Java `true` is sent as the *Mathematica* symbol `True`, and `False` for Java `false`:

```
void put(boolean b) throws MathLinkException;
```

There is also a `put()` method for arbitrary Java objects. In the default implementation, this does not do anything very useful for most objects (what it does is send `obj.toString()`). A handful of objects, however, have a meaningful representation to *Mathematica*. These are arrays, strings, `Expr` objects (discussed elsewhere), and instances of the so-called "wrapper" classes (`Integer`, `Double`, `Character`, and so on), which hold single numeric values. Arrays are sent as lists, strings are sent as *Mathematica* strings, and the wrapper classes are sent as their numeric value. (The last case is for complex numbers, which will be discussed later.)

```
void put(Object obj) throws MathLinkException;
```

There is a special method for arrays that lets you specify the heads of the array in each dimension. The heads are passed as an array of strings. Note that unlike the C counterparts (`MLPutInteger32Array`, `MLPutReal64Array`, and so on), you do not have to specify the depth or dimensions because they can be inferred from the array itself:

```
void put(Object array, String[] heads) throws MathLinkException;
```

For putting *Mathematica* functions:

```
void putFunction(String f, int argCount) throws MathLinkException;
```

For transferring expressions from one link to another (the 'this' link is the destination):

```
void transferExpression(MathLink source) throws MathLinkException;

void transferToEndOfLoopbackLink(LoopbackLink source) throws
MathLinkException;
```

Low-level "textual interface":

```
void putNext(int type) throws MathLinkException;

void putArgCount(int argCount) throws MathLinkException;

void putSize(int size) throws MathLinkException;

int bytesToPut() throws MathLinkException;

void putData(byte[] data) throws MathLinkException;

void putData(byte[] data, int len) throws MathLinkException;
```

Flushing:

```
void flush();
```

Getting

Because you cannot overload methods on the basis of return type, there is no catchall `get()` method for reading from the link, as is the case with the `put()` method. Instead, there are separate methods for each data type. Notice that unlike their counterparts in the C API, these methods return the actual data that was read, not an error code (exceptions are used for errors, as with all the methods).

```
int getInteger() throws MathLinkException;

long getLongInteger() throws MathLinkException;

double getDouble() throws MathLinkException;

String getString() throws MathLinkException;

byte[] getByteString(int missing) throws MathLinkException;

String getSymbol() throws MathLinkException;

boolean getBoolean() throws MathLinkException;
```

Arrays of the nine basic types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `String`), as well as complex numbers, can be read with a set of methods of the form `getXXXArrayN()`, where `XXX` is the data type and `N` specifies the depth of the array. For each type there are two methods like the following examples for `int`. There is no way to get the heads of the array using these functions (it will typically be "`List`" at every level). If you need to get the heads as well, you should use `getExpr()` to read the expression as an `Expr` and then examine it using the `Expr` methods.

```
int[] getIntArray1() throws MathLinkException;

int[][] getIntArray2() throws MathLinkException;

... and others for all the eight primitive types and String and the
complex class
```

Note that you do not have to know exactly how deep the array is to use these functions. If you call, say, `getFloatArray1()`, and what is actually on the link is a matrix of reals, then the data will be flattened into the requested depth (a one-dimensional array in this case). Unfortunately, if you do this you cannot determine what the original depth of the data was. If you call a function that expects an array of depth greater than the actual depth of the array on the link, it will throw a `MathLinkException`.

If you need to read an array of depth greater than 2 (but a maximum of 5) , you can use the `getArray()` method. The `getXXXArrayN()` methods already discussed are just convenience methods that use `getArray()` internally. The `type` argument must be one of `TYPE_BOOLEAN`, `TYPE_BYTE`, `TYPE_CHAR`, `TYPE_SHORT`, `TYPE_INT`, `TYPE_LONG`, `TYPE_FLOAT`, `TYPE_DOUBLE`, `TYPE_STRING`, `TYPE_EXPR`, `TYPE_BIGINTEGER`, `TYPE_BIGDECIMAL`, or `TYPE_COMPLEX`.

```
Object getArray(int type, int depth) throws MathLinkException;

// New in J/Link 2.0:
Object getArray(int type, int depth, String[] heads) throws
MathLinkException;
```

New in *J/Link* 2.0 is `getArray(int type, int depth, String[] heads)`. It reads an array and also tells you the heads at each level. See the JavaDocs for more information.

Unlike the C *MathLink* API, there are no methods for "disowning" strings or arrays because this is not necessary. When you read a string or array off the link, your program gets its own copy of the data, so you can write into it if you desire (although Java strings are immutable).

The `getFunction()` method needs to return two things: the head and the argument count. Thus, there is a special class called `MLFunction` that encapsulates both these pieces of information, and this is what `getFunction()` returns. The `MLFunction` class is documented later.

```
MLFunction getFunction() throws MathLinkException;

// Returns the function's argument count. Throws MathLinkException if the
function
// is not the specified one.
int checkFunction(String f) throws MathLinkException;

//Throws an exception if the incoming function does not have this head and
arg count.
void checkFunctionWithArgCount(String f, int argCount) throws
MathLinkException;
```

These methods support the low-level interface for reading from a link.

```
int getNext() throws MathLinkException;

int getType() throws MathLinkException;

int getArgCount() throws MathLinkException;

int bytesToGet() throws MathLinkException;

byte[] getData(int len) throws MathLinkException;
```

Reading `Expr` objects:

```
public Expr getExpr() throws MathLinkException;

// Gets an expression off the link, then resets the link to the state
// prior to reading the expr. You can "peek" ahead without consuming
anything
// off the link.
public Expr peekExpr() throws MathLinkException;
```

Messages

The messages referred to by the following functions are not *Mathematica* warning messages, but a low-level type of *MathLink* communication used mainly to send interrupt and abort requests. The `getMessage()` and `messageReady()` methods no longer function in *J/Link* 2.0 and later. You must use `setMessageHandler()` if you want to receive messages from *Mathematica.*

```
int getMessage() throws MathLinkException;

void putMessage(int msg) throws MathLinkException;

boolean messageReady() throws MathLinkException;
```

Marks

```
long createMark() throws MathLinkException;

//Next two don't throw, since they are often used in cleanup operations in
catch handlers.

void seekMark(long mark);

void destroyMark(long mark);
```

Complex Class

The `setComplexClass()` method lets you assign the class that will be mapped to complex numbers in *Mathematica*. "Mapped" means that the `put(Object)` method will send a *Mathematica* complex number when you call it with an object of your complex class, and `getComplex()` will return an instance of this class. For further discussion about this subject and the restrictions on the classes that can be used as the complex class, see "Complex Numbers".

```
public boolean setComplexClass(Class cls);

public Class getComplexClass();

public Object getComplex() throws MathLinkException;
```

Yield and Message Handlers

The `setYieldFunction()` and `addMessageHandler()` methods take a class, an object, and a method name as a string. The class is the class that contains the named method, and the object is the object of that class on which to call the method. Pass `null` for the object if it is a static method. The signature of the method you use in `setYieldFunction()` must be V(Z); for `addMessageHandler()` it must be II(V). See "Threads, Blocking, and Yielding" for more information and examples.

```
public boolean setYieldFunction(Class cls, Object obj, String methName);

public boolean addMessageHandler(Class cls, Object obj, String methName);

public boolean removeMessageHandler(String methName);
```

Constants

The *MathLink* class also includes the full set of user-level constants from `MathLink.h`. They have exactly the same names in Java as in C. In addition, there are some *J/Link*-specific constants.

```
static int ILLEGALPKT;
static int CALLPKT;
static int EVALUATEPKT;
static int RETURNPKT;
static int INPUTNAMEPKT;
static int ENTERTEXTPKT;
```

```
static int ENTEREXPRPKT;
static int OUTPUTNAMEPKT;
static int RETURNTEXTPKT;
static int RETURNEXPRPKT;
static int DISPLAYPKT;
static int DISPLAYENDPKT;
static int MESSAGEPKT;
static int TEXTPKT;
static int INPUTPKT;
static int INPUTSTRPKT;
static int MENUPKT;
static int SYNTAXPKT;
static int SUSPENDPKT;
static int RESUMEPKT;
static int BEGINDLGPKT;
static int ENDDLGPKT;
static int FIRSTUSERPKT;
static int LASTUSERPKT;

//These next two are unique to J/Link.
static int FEPKT;
static int EXPRESSIONPKT;

static int MLTERMINATEMESSAGE;
static int MLINTERRUPTMESSAGE;
static int MLABORTMESSAGE;

static int MLTKFUNC;
static int MLTKSTR;
static int MLTKSYM;
static int MLTKREAL;
static int MLTKINT;
static int MLTKERR;

//Constants for use in getArray()
static int TYPE_BOOLEAN;
static int TYPE_BYTE;
static int TYPE_CHAR;
static int TYPE_SHORT;
static int TYPE_INT;
static int TYPE_LONG;
static int TYPE_FLOAT;
```

```
static int TYPE_DOUBLE;
static int TYPE_STRING;
static int TYPE_BIGINTEGER;
static int TYPE_BIGDECIMAL;
static int TYPE_EXPR;
static int TYPE_COMPLEX;
```

The KernelLink Interface

`KernelLink` is the interface that you will probably use for the links in your programs. These are all public methods, as is always the case with a Java interface. This section provides only a brief summary of the `KernelLink` methods; it is intended mainly for those who want to skim a traditional printed listing. *The JavaDoc help files are the main method-by-method documentation for all the J/Link classes and interfaces*. They can be found in the JLink/Documentation/JavaDoc directory.

Evaluate

The `evaluate()` method encapsulates the steps needed to put an expression to *Mathematica* as a string or `Expr` and get the answer back as an expression. Internally, it uses an `EvaluatePacket` for sending the expression. The answer comes back in a `ReturnPacket`, although the `waitForAnswer()` method opens up the `ReturnPacket`—all you have to do is read out its contents. You should always use `waitForAnswer()` or `discardAnswer()` instead of spinning your own packet loop waiting for a `ReturnPacket`. See "The *MathLink* 'Packet Loop'".

```
void evaluate(String s) throws MathLinkException;

void evaluate(Expr e) throws MathLinkException;
```

Waiting for the Result

Call `waitForAnswer()` right after `evaluate()` (or if you manually send calculations wrapped in `EvaluatePacket`). It will read packets off the link until it encounters a `ReturnPacket`, which will hold the result. See "The *MathLink* 'Packet Loop'".

```
void waitForAnswer() throws MathLinkException;
```

The `discardAnswer()` method just throws away all the results from the calculation, so the link will be ready for the next calculation. As you may have guessed, it is nothing more than `waitForAnswer()` followed by `newPacket()`.

```
void discardAnswer() throws MathLinkException;
```

The "evaluateTo" Methods

The next set of methods are extensions of `evaluate()` that perform the put and the reading of the result. You do not call `waitForAnswer()` and then read the result yourself. They also do not throw `MathLinkException`—if there is an error, they clean up for you and return `null`. The "evaluateTo" in their names indicates that the methods perform the entire process themselves. `evaluateToInputForm()` returns a string formatted in `InputForm` at the specified page width. Specify 0 for the page width to get `Infinity`. `evaluateToOutputForm()` is exactly like `evaluateToInputForm()` except that it returns a string formatted in `OutputForm`. `OutputForm` results are more attractive for display to the user, but `InputForm` is required if you want to pass the string back to *Mathematica* to be used in further computations. The `evaluateToImage()` method will return a `byte[]` of GIF data if you give it an expression that returns a graphic, for example, a `Plot` command. Pass 0 for the `dpi`, `int`, and `width` arguments if you want their `Automatic` settings in *Mathematica*. `evaluateToTypeset()` returns a `byte[]` of GIF data of the result of the computation, typeset in either `StandardForm` or `TraditionalForm`. These methods are discussed in detail in evaluateToImage() and evaluateToTypeset()

```
String evaluateToInputForm(String s, int pageWidth);
String evaluateToInputForm(Expr e, int pageWidth);

String evaluateToOutputForm(String s, int pageWidth);
String evaluateToOutputForm(Expr e, int pageWidth);

byte[] evaluateToImage(String s, int width, int height);
byte[] evaluateToImage(Expr e, int width, int height);
byte[] evaluateToImage(String s, int width, int height, int dpi, boolean
useFrontEnd);
byte[] evaluateToImage(Expr e, int width, int height, int dpi, boolean
useFrontEnd);

byte[] evaluateToTypeset(String s, int pageWidth, boolean useStdForm);
byte[] evaluateToTypeset(Expr e, int pageWidth, boolean useStdForm);

// Returns the exception that caused the most recent "evaluateTo" method
to return null
Throwable getLastError();
```

Sending Java Object References

If you want to send Java objects "by reference" to *Mathematica* so that *Mathematica* code can call back into your Java runtime via the "installable Java" facility described in "Calling Java from *Mathematica*", you must first call the `enableObjectReferences()` method. This is described in "Sending Object References to *Mathematica*".

```
void enableObjectReferences() throws MathLinkException;
```

Like the `MathLink` interface, `KernelLink` has a `put()` method that sends objects. The `MathLink` version of this method only sends objects "by value". The `KernelLink` version behaves just like the `MathLink` version for those objects that can be sent by value. In addition, though, it sends all other objects by reference. You must have called `enableObjectReferences()` before calling `put()` on an object that will be sent by reference. See "Sending Object References to *Mathematica*".

```
void put(Object obj) throws MathLinkException;
```

The next methods are for putting and getting objects by reference (you must have called `enableObjectReferences()` to use these methods).

```
public void putReference(Object obj) throws MathLinkException;

Object getObject() throws MathLinkException;

// These two methods from the MathLink interface are enhanced to return
MLTKOBJECT if a Java
// object reference is waiting to be read.
int getNext() throws MathLinkException;
int getType() throws MathLinkException;
```

Interrupting, Aborting, and Abandoning Evaluations

These methods are for aborting and interrupting evaluations. They are discussed in "Aborting and Interrupting Computations".

```
void abortEvaluation();
void interruptEvaluation()
void abandonEvaluation();

void terminateKernel();
```

Support for PacketListeners

These methods support the registering and notification of `PacketListener` objects. They are discussed in "Using the PacketListener Interface".

```
void addPacketListener(PacketListener listener);
void removePacketListener(PacketListener listener);

boolean notifyPacketListeners(int pkt);
```

The handlePacket() Method (Advanced Users Only)

The `handlePacket()` method is for very advanced users who are writing their own packet loop instead of calling `waitForAnswer()`, `discardAnswer()`, or any of the "evaluateTo" methods. See the JavaDocs for more information.

```
void handlePacket(int pkt) throws MathLinkException;
```

Methods Valid Only for "StdLinks"

Finally, there are some methods that are meaningful only in methods that are themselves called from *Mathematica* via the "installable Java" functionality described in "Calling Java from *Mathematica*". These methods are documented in "Writing Your Own Installable Java Classes". You will not use them if you are writing a program that uses *Mathematica* as a computational engine.

```
public void print(String s);
public void message(String symtag, String[] args);
public void message(String symtag, String arg);
public void beginManual();

public boolean wasInterrupted();
public void clearInterrupt();
```

Sending Computations and Reading Results

MathLink Packets

Communication with the *Mathematica* kernel generally takes place in the form of "packets". A *MathLink* packet is just a *Mathematica* function, albeit one from a set that is recognized and treated specially by *MathLink*. When you send something to *Mathematica* to be evaluated, you wrap it in a packet that tells *Mathematica* that this is a request for something to be computed,

and also tells something about how it is to be computed. All output you receive from *Mathematica*, including the result and any other side effect output like messages, `Print` output, and graphics, will also arrive wrapped in a packet. The type of packet tells you about the contents.

A *MathLink* program typically sends a computation to *Mathematica* wrapped in a special packet, and then reads a succession of packets arriving from the kernel until the one containing the result of the computation arrives. Along the way, packets that do not contain the result can be either discarded without bothering to examine them or they can be "opened" and operated on. Such nonresult packets include `TextPacket` expressions containing `Print` output, `MessagePacket` expressions containing *Mathematica* warning messages, `DisplayPacket` expressions containing PostScript, and several other types.

You can look at existing *MathLink* documentation for information on the various packet types for sending things to *Mathematica* and for what *Mathematica* sends back. In particular, you should look at *MathLink* Tutorial (http://library.wolfram.com/infocenter/TechNotes/174/). For most uses, *J/Lin*k hides all the details of packet types and how to send and receive them. You only need to read about packet types if you want to do something beyond what the built-in behavior of *J/Link* provides. This can be useful for many programs.

The *MathLink* "Packet Loop"

In a C *MathLink* program, a typical code fragment for sending a computation to *Mathematica* and throwing away the result might look like this:

```
// C code
MLPutFunction(ml, "EvaluatePacket", 1);
MLPutFunction(ml, "ToExpression", 1);
MLPutString(ml, "Needs[\"MyPackage`\"]");
MLEndPacket(ml);
while (MLNextPacket(ml) != RETURNPKT)
    MLNewPacket(ml);
MLNewPacket();
```

After sending the computation (wrapped in an `EvaluatePacket`), the code enters a `while` loop that reads and discards packets until it encounters the `ReturnPacket`, which will contain the result (which will be the symbol `Null` here). Then it calls `MLNewPacket` once again to discard the `ReturnPacket`.

A *MathLink* program will typically do this same basic operation many times, so *J/Link* hides it within some higher-level methods in the `KernelLink` interface. Here is the *J/Link* equivalent:

```
ml.evaluate("Needs[\"MyPackage`\"]");
ml.discardAnswer();
```

The `discardAnswer()` method discards all packets generated by the computation until it encounters the one containing the result, and then discards that one too. There is a related method, `waitForAnswer()`, that discards everything up until the result is encountered. When `waitForAnswer()` returns, the `ReturnPacket` has been opened and you are ready to read out its contents. You can probably guess that `discardAnswer()` is just `waitForAnswer()` followed by `newPacket()`.

Not only is it a convenience to hide the packet loop within `waitForAnswer()` and `discardAnswer()`, it is necessary in some circumstances, since special packets may arrive that *J/Link* needs to handle internally. Although *J/Link* has `nextPacket()` and `newPacket()` methods, programmers should not write `nextPacket()`/`newPacket()` loops like the one in the last C code fragment. Stick to calling `waitForAnswer()`, `discardAnswer()`, or using the "evaluateTo" methods discussed in the next section. If you really need to know about all the packets that arrive in your program, use the `PacketListener` interface, discussed later.

Sending an Evaluation

J/Link provides three main ways to send an expression to *Mathematica* for evaluation. All three techniques are demonstrated in the sample program in the section "Sample Program".

- If you do not care about the result of the evaluation, or if you want the result to arrive in a form other than a string or image, you can use the `evaluate()` method.
- You can send the expression "manually" like in a traditional C *MathLink* program, by putting the `EvaluatePacket` head followed by the parts of the expression using low-level methods from the `MathLink` interface.
- If you want the result back as a string or image, you can use the "evaluateTo" methods, which provide a very high-level and convenient interface.

The "evaluateTo" methods are recommended for their convenience, if you want the result back in one of the formats that that they provide. These methods are discussed in "The 'evaluateTo' Methods". If the expression you want evaluated is in the form of a string or `Expr` (the `Expr`

class is discussed in "Motivation for the Expr Class"), or can be easily converted into one, then you will want to use the `evaluate()` method. If none of these convenience methods are appropriate, you can put the expression piece by piece similar to a traditional C *MathLink* program. You do this by sending pieces in a structure that mirrors the `FullForm` of the expression. Here is a comparison of using these three techniques for sending the computation `NIntegrate[`$x^2 + y^2$`, {`x`, -1, 1}, {`y`, -1, 1}]`:

```
String strResult = ml.evaluateToInputForm("NIntegrate[x^2 + y^2, {x,-1,1},
{y,-1,1}]");

ml.evaluate("NIntegrate[x^2 + y^2, {x,-1,1}, {y,-1,1}]");
ml.waitForAnswer();
double doubleResult1 = ml.getDouble();

// It is convenient to use indentation to indicate the structure
ml.putFunction("EvaluatePacket", 1);
  ml.putFunction("NIntegrate", 3);
    ml.putFunction("Plus", 2);
      ml.putFunction("Power", 2);
        ml.putSymbol("x");
        ml.put(2);
      ml.putFunction("Power", 2);
        ml.putSymbol("y");
        ml.put(2);
    ml.putFunction("List", 3);
      ml.putSymbol("x");
      ml.put(-1);
      ml.put(1);
    ml.putFunction("List", 3);
      ml.putSymbol("y");
      ml.put(-1);
      ml.put(1);
ml.endPacket();
ml.waitForAnswer();
double doubleResult2 = ml.getDouble();
```

Reading the Result

Before diving into reading expressions from a link, keep in mind that if you just want the result back as a string or an image, then you are better off using one of the "evaluateTo" methods described in the next section. These methods send a computation and return the answer as a string or image, so you do not have to read it off the link yourself. Also, if you are not interested in the result, you will use `discardAnswer()` and thus not have to bother reading it.

J/Link provides a number of methods for reading expressions from a link. Many of these methods are essentially identical to functions in the *MathLink* C API, so to some extent you can learn how to use them by reading standard *MathLink* documentation. You should also consult the *J/Link* JavaDoc files for more information. The reading methods generally begin with "get." Examples are `getInteger()`, `getString()`, `getFunction()`, and `getDoubleArray2()`. There are also two type-testing methods that will tell you the type of the next thing waiting to be read off the link. These methods are `getType()` and `getNext()`.

As stated earlier, one method you will generally *not* call is `nextPacket()`. When `waitForAnswer()` returns, `nextPacket()` has already been called internally on the `ReturnPacket` that holds the answer, so this final packet has already been "opened" and you can start reading its contents right away.

The vast majority of `MathLinkException` ocurrences in *J/Link* programs are caused by trying to read the incoming expression in a manner that is not appropriate for its type. A typical example is calling a *Mathematica* function that you expect to return an integer, but you call it with incorrect arguments and therefore it returns unevaluated. You call `getInteger()` to read an integer, but what is waiting on the link is a function like `foo[badArgument]`. There are several general ways for dealing with problems like this. The first technique is to avoid the exception by using `getNext()` to determine the type of the expression waiting. For example:

```
ml.evaluate("SomeFunction[]");
ml.waitForAnswer();
int result;
int type = ml.getNext();
if (type == MathLink.MLTKINT) {
    result = ml.getInteger();
} else {
    // What you do here is up to you.
    System.out.println("Unexpected result: " + ml.getExpr().toString());
    // Throw away the packet contents.
    ml.newPacket();
}
```

A related technique is to read the result as an `Expr` and examine it using the `Expr` methods. The `Expr` class is discussed in "Motivation for the Expr Class".

```
ml.evaluate("SomeFunction[]");
ml.waitForAnswer();
int result;
Expr e = ml.getExpr();
if (e.integerQ()) {
    result = e.asInt();
} else {
    // What you do here is up to you.
    System.out.println("Unexpected result: " + e.toString());
}
```

A final technique is to just go ahead and read the expression in the form that you expect, but catch and handle any `MathLinkException`. (Remember that the entire code fragment that follows must be wrapped in a `try/catch` block for `MathLinkException` objects, but you are only seeing an inner `try/catch` block for `MathLinkException` objects known to be thrown during the read.)

```
ml.evaluate("SomeFunction[]");
ml.waitForAnswer();
int result;
try {
    result = ml.getInteger();
} catch (MathLinkException e) {
    ml.clearError();
    System.out.println("Unexpected result: " + ml.getExpr().toString());
    ml.newPacket(); // Not strictly necessary because of the getExpr()
above
}
```

Another tip for avoiding bugs in code that reads from a link is to use the `newPacket()` method liberally. **A second very common cause of `MathLinkException` occurences is forgetting to read the entire contents of a packet before going on to the next computation.** The `newPacket()` method causes the currently opened packet to be discarded. Another way of saying this is that it throws away all unread parts of the expression that is currently being read. It is a clean-up method that ensures that there are no remnants left over from the last packet when you go on to the next evaluation. Consider the following code:

```
ml.evaluate("SomeFunction[]");
ml.waitForAnswer();
int result;
try {
    result = ml.getInteger();
} catch (MathLinkException e) {
    ml.clearError();
    System.out.println("Unexpected result");
    // Oops. Forgot to call newPacket() to throw away the contents.
}

// Boom. The next line causes a MathLinkException if the previous
getInteger()
// call failed, because nextPacket() will be called before the previous
packet
// was emptied.
ml.evaluate("AnotherFunction[]");
ml.discardAnswer();
```

This code will cause a `MathLinkException` to be thrown at the indicated point if the previous call to `getInteger()` had failed because the programmer forgot to either finish reading the result or call `newPacket()`. Here is an even simpler example of this error:

```
ml.evaluate("SomeFunction[]");
ml.waitForAnswer();
// Oops. Forgot to read or throw away the result.
// Probably meant to call discardAnswer() instead of
// waitForAnswer().

ml.evaluate("AnotherFunction[]");
ml.discardAnswer();   // MathLinkException here!
```

The "evaluateTo" Methods

J/Link provides another set of convenience methods that hide the packet loop within them. These methods perform the entire procedure of sending something to *Mathematica* and returning the result in one form or another. They have names that begin with "evaluateTo" to indicate that they actually return the result, rather than merely send it, as with the `evaluate()` method.

```
String evaluateToInputForm(String s,int pageWidth);
String evaluateToInputForm(Expr e,int pageWidth);

String evaluateToOutputForm(String s,int pageWidth);
String evaluateToOutputForm(Expr e,int pageWidth);

byte[] evaluateToImage(String s, int width, int height);
byte[] evaluateToImage(Expr e, int width, int height);
byte[] evaluateToImage(String s, int width, int height, int dpi, boolean
useFE);
byte[] evaluateToImage(Expr e, int width, int height, int dpi, boolean
useFE);

byte[] evaluateToTypeset(String s, int width, boolean useStdForm);
byte[] evaluateToTypeset(Expr e, int width, boolean useStdForm);
```

Only `evaluateToInputForm()` and `evaluateToOutputForm()` are discussed in this section, deferring consideration of `evaluateToImage()` and `evaluateToTypeset()` until the section "evaluateToImage() and evaluateToTypeset()", Graphics and Typeset Output. The `evaluateToInputForm()` and `evaluateToOutputForm()` methods encapsulate the very common need of sending some code as a string and getting the result back as a formatted string. They differ only in whether the string is formatted in `InputForm` or `OutputForm`. `OutputForm` is good when you want to display the string to the user, and `InputForm` is good if you need to send the expression back to *Mathematica* or if you need to save it to file or splice it into another expression. These methods take a `pageWidth` argument to specify how many character widths you want the maximum line length to be. Pass in 0 for a page width of infinity.

The `evaluateTo` methods do not throw a `MathLinkException`. Instead, they return `null` to indicate that a problem occurred. This is not very likely unless there is a serious problem, such as if the kernel has unexpectedly quit. In the event that `null` is returned from one of these methods, you can call `getLastError()` to get the `Throwable` object that represents the exception thrown to cause the unexpected result. Generally, it will be a `MathLinkException`, but there are some other rare cases (like an `OutOfMemoryError` if an image was returned that would have been too big to handle).

```
// Give the (caught) exception that prevented a normal return from the
last
// call to an "evaluateTo" method.
Throwable getLastError();
```

All the evaluateTo methods take the input to evaluate in the form of a string or an Expr. Although a full discussion of the Expr class is deferred until "Motivation for the Expr Class", a brief discussion is provided here on how and why you might want to send the input as an Expr. It is often convenient to specify *Mathematica* input as a string, particularly if it is taken directly from a user, such as the contents of a text field. There are times, though, when it is difficult or unwieldy to work with strings. This is particularly true if the expression to evaluate is built up programmatically, or if it is being read off one link to be written onto the link to the kernel. One way to deal with this circumstance is to forgo the convenience of using, say, evaluateToOutputForm() and instead hand-code the entire operation of sending the input so that the answer will come back formatted in OutputForm. You would have to send the EvaluatePacket head and use the ToString function to get the output as a string:

```
// This duplicates the following:
// String output = ml.evaluateToOutputForm("Integrate[5 x^n a^x, x]", 0);

// As an expression, we send ToString[Integrate[5 x^n a^x, x], PageWidth->Infinity]
ml.putFunction("EvaluatePacket", 1);
    ml.putFunction("ToString", 2);
    ml.putFunction("Integrate", 2);
          ml.putFunction("Times", 3);
              ml.put(5);
              ml.putFunction("Power", 2);
                  ml.putSymbol("x");
                  ml.putSymbol("n");
              ml.putFunction("Power", 2);
                  ml.putSymbol("a");
                  ml.putSymbol("x");
          ml.putSymbol("x");
       ml.putFunction("Rule", 2);
          ml.putSymbol("PageWidth");
          ml.putSymbol("x");
ml.endPacket();
ml.waitForAnswer();
String output = ml.getString();
```

This version is considerably more verbose, but most of the code comes from the deliberate decision to send the expression piece-by-piece, not as a single string. There are a few extra lines that ensure that the answer comes back as a properly formatted string and that read the

result from the link. It is no great loss to have to do it all by hand. But what if you wanted to do the equivalent with `evaluateToTypeset()`? Most programmers would have no idea how to perform all the work to get the answer in the desired form. If all the `evaluateTo` methods took only strings, then *J/Link* programmers would have to either compose all their input as strings or figure out the difficult steps that are already handled for them by the internals of the various `evaluateTo` methods.

The solution to this is to allow `Expr` arguments as an alternative to strings. Although `Expr` has a set of constructors, the easiest way to create a complicated one is to build the expression on a loopback link and read it off the link as an `Expr`. You can then pass that `Expr` to the desired `evaluateTo` method:

```
LoopbackLink loop = MathLinkFactory.createLoopbackLink();

// Create the expression EvaluatePacket[Integrate[5 x^n a^x, x]] on the
loopback link
loop.putFunction("Integrate", 2);
    loop.putFunction("Times", 3);
        loop.put(5);
        loop.putFunction("Power", 2);
            loop.putSymbol("x");
            loop.putSymbol("n");
        loop.putFunction("Power", 2);
            loop.putSymbol("a");
            loop.putSymbol("x");
    loop.putSymbol("x");
loop.endPacket();

// Now read the Expr off the loopback link
Expr e = loop.getExpr();
// We are done with the loopback link now.
loop.close();
String result = ml.evaluateToOutputForm(e, 0);
e.dispose();
```

In this way, you can build expressions manually with a series of “put” calls and still have the convenience of using the high-level `evaluateTo` methods.

Using the PacketListener Interface

A central component of a standard C *MathLink* program is a packet-reading loop, which typically consists of calling the *MathLink* API functions `MLNextPacket` and `MLNewPacket` until a desired packet is encountered. *J/Link* programs will typically not include such a loop—instead, you call the `KernelLink` methods `waitForAnswer()`, `discardAnswer()`, or one of the "evaluateTo" methods, which hide the packet loop within them. In some cases, though, programmers will want to observe and/or operate on the incoming flow of packets. A typical example would be to display `Print` output or messages generated by a computation. These outputs are side effects of a computation and not part of the "answer", and they are normally discarded by *J/Link*'s internal packet loop.

To accommodate this need, `KernelLink` objects fire a `PacketArrivedEvent` when the internal packet loop reads a packet (that is, right after `nextPacket()` has been called). You can register your interest in receiving notifications when packets arrive by creating a class that implements the `PacketListener` interface and registering it with the `KernelLink` object. This event notification is done according to the standard Java design pattern for events and event listeners. You create a class that implements `PacketListener`, and then call the `KernelLink` method `addPacketListener()` to register this object to receive notifications.

The `PacketListener` interface contains only one method, `packetArrived()`:

```
public boolean packetArrived(PacketArrivedEvent evt) throws
MathLinkException;
```

Your `PacketListener` object will have its `packetArrived()` method called for every incoming packet. At the time `packetArrived()` is called, the packet has been opened with `nextPacket()`. Your code can begin reading the packet contents. The argument to `packetArrived()` is a `PacketArrivedEvent`, from which you can extract the link and the packet type (see the example that follows).

The really nice thing about your `packetArrived()` implementation is that you can consume or ignore the packet *without affecting the internal packet loop in any way*. You do not need to be concerned about interfering with any other `PacketListener` or *J/Link*'s own internal handling of packets. You can read all, some, or none of the contents of any packet.

The `packetArrived()` method returns a Boolean to indicate whether you want to prevent *J/Link*'s internal code from seeing the packet. This very advanced option lets you completely

override *J/Link*'s own handling of packets. At this time, the internals of *J/Link*'s packet handling are undocumented, so programmers will have no use for the override ability. Your `packetArrived()` method should always return `true`.

Here is a sample `packetArrived()` implementation that looks only for `TextPacket` expressions, printing their contents to the `System.out` stream.

```
public boolean packetArrived(PacketArrivedEvent evt) throws
MathLinkException {
    if (evt.getPktType() == MathLink.TEXTPKT) {
        KernelLink ml = (KernelLink) evt.getSource();
        System.out.println(ml.getString());
    }
    return true;
}
```

This design pattern of using an event listener that gets a callback for every packet received allows your program to be very flexible in its handling of packets. You do not have to significantly change your program to implement different policies, such as ignoring nonresult packets, printing them to the `System.out` stream, writing them to a file, displaying them in a window, and so on. Just slot in different `PacketListener` objects with different behavior, and leave all the program logic unchanged. You can use as many `PacketListener` objects as you want.

The PacketPrinter Class for Debugging

J/Link provides one implementation of the `PacketListener` interface that is designed to simplify debugging *J/Link* programs. The `PacketPrinter` class prints out the contents of each packet on a stream you specify. Here is the constructor:

```
public PacketPrinter(PrintStream strm);
```

Here is a code fragment showing a typical use:

```
PacketListener stdoutPrinter = new PacketPrinter(System.out);
ml.addPacketListener(stdoutPrinter);
...
String result = ml.evaluateToOutputForm("Integrate[x^n a^x, x]", 72);
```

It is especially useful to see *Mathematica* messages that were generated during the computation. Using a `PacketPrinter` to see exactly what *Mathematica* is sending back is an extremely useful debugging technique. It is no exaggeration to say that **the vast majority of problems**

with *J/Link* programs can be identified simply by adding one line of code that creates and installs a `PacketPrinter`. When you are satisfied that your program is behaving as expected, just take out the `addPacketListener()` line. No other code changes are required.

Using EnterTextPacket

As noted earlier, when you send something to *Mathematica* to be evaluated, you wrap it in a packet. *Mathematica* supports three different packets for sending computations, but the two that are most important are `EvaluatePacket` and `EnterTextPacket`. `EvaluatePacket` has been used manually in a few code fragments, and they are used internally by the `evaluate()` and `evaluateTo` methods. When *Mathematica* receives something wrapped in `EvaluatePacket`, it evaluates it and sends the result back in a `ReturnPacket`. Side effects like `Print` output and PostScript for graphics are sent in their own packets prior to the `ReturnPacket`. In contrast, when *Mathematica* receives something in an `EnterTextPacket`, it runs its full "main loop", which includes, among other things, generating `In` and `Out` prompts, applying the `$Pre`, `$PrePrint` and `$Post` functions, and keeping an input and output history. This is how the notebook front end uses the kernel. You might want to look at the more detailed discussion of the properties of these packets in *MathLink* Tutorial, available on *MathSource*.

If you are using the kernel as a computational engine, you probably want to use `EvaluatePacket`. Use `EnterTextPacket` instead when you want to present your users with an interactive "session" where previous outputs can be retrieved by number or `%`. An example is if you are providing functionality similar to the notebook front end, or the kernel's standalone "terminal" interface. The use of `EnterTextPacket` as the wrapper packet for computations is not as well supported in *J/Link*, since it will be used much more rarely. You cannot use the `evaluateTo` methods, since they use `EvaluatePacket`.

The packet sequence you get in return from an `EnterTextPacket` computation will not always have a `ReturnTextPacket` in it. If the computation returns `Null`, or if there is a syntax error, no `ReturnTextPacket` will be sent. The final packet that will always be sent is `InputNamePacket`, containing the input prompt to use for the next computation. This means that the `waitForAnswer()` method must accommodate two situations: for most computations, the answer will be in a `ReturnTextPacket`, but for some computations, there will be no answer at all. Therefore `waitForAnswer()` returns when either a `ReturnTextPacket` or an `InputNamePacket` is encountered. This is why `waitForAnswer()` returns an `int`—this is the

packet type that caused `waitForAnswer()` to return. If your call to `waitForAnswer()` returns `MathLink.RETURNTEXTPKT`, then you can read the answer (it will be a string), and then you call `waitForAnswer()` again to receive the `InputNamePacket` that will come afterward. You can read the prompt string with `getString()` (it will be something like "In[1]:="). If the original `waitForAnswer()` returns `MathLink.INPUTNAMEPKT`, then there was no result to display, and you can just call `getString()` to read the input prompt string. In the first case, where a `ReturnTextPacket` does come, instead of calling `waitForAnswer()` a second time to read off the subsequent `InputNamePacket`, you could simply call `nextPacket()`, because the `InputNamePacket` will always immediately follow the `ReturnTextPacket`. Although it might look a little weird, calling `waitForAnswer()` has the advantage of triggering notification of all registered `PacketListener` objects, which would not happen if you manually read a packet with `nextPacket()`. In other words, it is better to let all packets be read by *J/Link*'s internal loop.

Here is an example:

```
String inputString = getStringFromUser();
ml.putFunction("EnterTextPacket", 1);
ml.put(inputString);
String result = null;
int pkt = ml.waitForAnswer();
if (pkt == MathLink.RETURNTEXTPKT) {
    // Mathematica computation returned a non-Null result, so a
RETURNTEXTPKT
    // was generated. Read its contents (a string).
    result = ml.getString();
    // Now call waitForAnswer() again, which will return after opening the
    // InputNamePacket that will always follow. It is essentially
    // nothing more than a call to nextPacket() in this circumstance:
    ml.waitForAnswer();
}
// At this point, a call to waitForAnswer() has returned
MathLink.INPUTNAMEPKT,
// so we just read out the contents, which is the next input prompt.
String nextPrompt = ml.getString();
```

You will probably want to use a `PacketListener` when you are using `EnterTextPacket`, because you probably want to show your users the full stream of output arriving from *Mathematica*, which might include messages and `Print` output. Your `PacketListener` implementation could write the incoming packets to your input/output session window. In fact, if you have such

a PacketListener, you might want to let it handle *all* output, including the ReturnTextPacket containing the result and the InputNamePacket containing the next prompt. Then you would just call discardAnswer() in your main program and let your PacketListener handle everything.

Handling MathLinkExceptions

Most of the *MathLink* and KernelLink methods throw a MathLinkException if a *MathLink* error occurs. This is in contrast to the *MathLink* C API, where functions return an error code. The methods that do not throw a MathLinkException are generally ones that will often need to be used within a catch block handling a MathLinkException that had already been thrown. If these methods threw their own exceptions, then you would need to nest another try/catch block within the catch block.

A well-formed *J/Link* program will typically not throw a MathLinkException except in the case of fatal *MathLink* errors, such as the kernel unexpectedly quitting. What is meant by "well-formed" is that you do not make any overt mistakes when putting or getting expressions, such as specifying an argument count of three in a putFunction() call but only sending two, or calling nextPacket() before you have finished reading the contents of the current packet. The *J/Link* API helps you avoid such mistakes by providing high-level functions like waitForAnswer() and evaluateToOutputForm() that hide the low-level interaction with the link, but in all but the most trivial *J/Link* programs it is still possible to make such errors. Just remember that the vast majority of MathLinkException objects thrown represent logic errors in the code of the program, not user errors or runtime anomalies. They are just bugs to which the programmer needs to be alerted so that they can be fixed.

In a small, well-formed *J/Link* program, you may be able to put a lot of *J/Link* calls, perhaps even the entire program, within a single try/catch block because there is no need to know exactly what the program was doing when the error occurred—all you are going to do is print a message and exit. The example program in the section "Sample Program" has this structure. Many *J/Link* programs will need to be a little more refined in their treatment of MathLinkException objects than just quitting. No matter what type of program you are writing, it is strongly recommended that while you are developing the program, you use try/catch blocks in a fine-grained way (that is, only wrapping small, meaningful units of code in each try/catch block), and always put code in your catch block that prints a message or alerts you

in some way. Many hours of debugging have been wasted because programmers did not realize a *MathLink* error had occurred, or they incorrectly identified the region of code where it happened.

Here is a sample of how to handle a `MathLinkException` in the case where you want to try to recover. The first thing is to call `clearError()`, as other *MathLink* calls will fail until the error state is cleared. If `clearError()` returns `false` then there is nothing to do but close the link. An example of the type of error that `clearError()` will fix is the very common mistake of calling `nextPacket()` before the current packet has been completely read. After `clearError()` is called, the link is reset to the state it was in before the offending `nextPacket()`. You can then read the rest of the current packet or call `newPacket()` to throw it away. Another example of a class of errors where `clearError()` will work is calling an incorrect "get" method for the type of data waiting on the link—for example, calling `getFunction()` when an integer is waiting. After calling `clearError()`, you can read the integer.

```
try {
    ...
} catch (MathLinkException e) {
    System.err.println(e.toString());
    if (ml.clearError() != true)    {
        System.err.println("MathLinkException was unrecoverable; closing
link.");
        ml.close();
        return;  // Or whatever cleanup is appropriate
    }
    // How you respond after clearError is up to you.
}
```

What you do in your `catch` block after calling `clearError()` will depend on what you were doing when the exception was thrown. About the only useful general guideline provided here is that if you are reading from the link when the exception is thrown, call `newPacket()` to abandon the rest of the packet. At least then you will know that you are ready to read a fresh packet, even if you have lost the contents of the previous packet.

`MathLinkException` has a few useful methods that will tell you about the cause of the exception. The `getErrCode()` method will give you the internal *MathLink* error code, which can be looked up in the *MathLink* documentation. It is probably more useful to get the internal message associated with the error, which is given by `getMessage()`. The `toString()` method gives you all this information, and will be the most useful output for debugging.

```
// Some useful MathLinkException methods.
public int getErrCode();
public String getMessage();
public String toString();
public Throwable getCause();
```

Some MathLinkException exceptions might not be "native" *MathLink* errors, but rather special exceptions thrown by implementations of the various link interfaces. *J/Link* follows the standard "exception chaining" idiom by allowing link implementations to catch these exceptions internally, wrap them in a MathLinkException, and re-throw them. As an example, consider a KernelLink implementation built on top of Java Remote Method Invocation (RMI). Methods called via RMI can throw a RemoteException, so such a link implementation might choose to catch internally every RemoteException and wrap it in a MathLinkException. If it did not do this, and instead all its methods could throw a RemoteException in addition to MathLinkException, all client code that used it would have to be modified. What all this means is that if you catch a MathLinkException, it might be "wrapping" another exception, instead of representing an internal *MathLink* problem. You can use the getCause() method on the MathLinkException instance to retrieve the wrapped exception that was the actual cause of the problem. The getCause() method will return null in the typical case where the MathLinkException is not wrapping another type of exception.

Graphics and Typeset Output

Preamble

Many developers who are writing Java programs that use *Mathematica* will want to produce *Mathematica* graphics and typeset expressions. This is a relatively complex subject, although *J/Link* has some very high-level methods designed to make obtaining and displaying these images very simple. If you want to display *Mathematica* images in a Java window, you can use the MathCanvas or MathGraphicsJPanel components, discussed in the next section. If you want a little more control over the process, or if you want to do something with the image data other than display it (like write it to a file or stream), you should read the section on the evaluateToImage() and evaluateToTypeset() methods.

MathCanvas and MathGraphicsJPanel

The MathCanvas and MathGraphicsJPanel classes were discussed in "The MathCanvas and MathGraphicsJPanel Classes" because they are often used from *Mathematica* programs. They

are just as useful in Java programs. Each is a simple graphical component (a JavaBean, in fact), that can display *Mathematica* graphics and typeset expressions. `MathCanvas` is a subclass of the AWT `Canvas` class, and `MathGraphicsJPanel` is a sublcass of the Swing `JPanel` class. They are conceptually identical and have the same set of extra methods for dealing with *Mathematica* graphics. You use `MathCanvas` when you want an AWT component and `MathGraphicsJPanel` when you want a Swing component.

Programmers who want to see how they work are strongly encouraged to examine the source code. The most important methods from these classes are as follows:

```
public void setMathCommand(String cmd);
public void setImageType(int type);
public void setUsesFE(boolean useFE);
public void setUsesTraditionalForm(boolean useTradForm);

public void setImage(Image im);

public void recompute();

public void repaintNow();
```

For brevity, the discussion that follows will refer only to `MathCanvas`; everything said applies equally to `MathGraphicsJPanel`. Use `setMathCommand()` to specify arbitrary *Mathematica* code that will be evaluated and have its result displayed. If you are using your `MathCanvas` to display *Mathematica* graphics, the result of the computation must be a graphics object (that is, an expression with head `Graphics`, `Graphics3D`, and so on). It is not enough that the command *produces* a graphic—it must *return* a graphic. Thus, `setMathCommand("Plot[x,{x,0,1}]")` will work, but `setMathCommand("Plot[x,{x,0,1}];")` will not because the trailing semicolon causes the expression to evaluate to `Null`. If you are using the `MathCanvas` to display typeset output, then the result of executing the code supplied in `setMathCommand()` can be anything. Its typeset form will be displayed. Within the code that you specify via `setMathCommand()`, quotation marks and other characters that have special meanings inside Java strings must be escaped by preceding them with a backslash, as in `setMathCommand("Plot[x,{x,0,1},PlotLabel->\"A Plot\"]")`.

The `setImageType()` method is what toggles between displaying a graphic and displaying a typeset expression. Call `setImageType(MathCanvas.GRAPHICS)` or `setImageType(MathCanvas.TYPESET)` to toggle between the two modes.

J/Link can create images of *Mathematica* graphics in two ways, either by using only the kernel or by using the kernel along with some extra services from the front end. The front end generally can do a better job, but there are some tradeoffs involved. If you want to use the front end, call `setUsesFE(true)`. When you call `setUsesFE(true)`, the front end may be launched, or an already running copy may be used. The exact behavior depends on what operating system and version of *Mathematica* you have. In *Mathematica* 6.0 and later, all graphics output requires the front end, so the `setUsesFE()` mehod has no effect—it is always `true`.

For typeset output, the default is `StandardForm`. To change to `TraditionalForm` call `setUsesTraditionalForm(true)`. When generating typeset output (that is, if you have called `setImageType(MathCanvas.TYPESET)`), the front end is always involved in generating typeset output, so make sure you understand the issues discussed in Using the Front End as a Service.

When you call `setMathCommand()`, the command is executed immediately and the resulting image is cached and used every time the window is repainted. Sometimes the code in your math command depends on variables that will change. To force the command to be recomputed and the new image displayed, call `recompute()`.

The `repaintNow()` method is like a "smart" version of the `JComponent` method `paintImmediately()`, and you use it in the same circumstances as `paintImmediately()`. It knows about the image that needs to be drawn and it will block until all the pixels are ready. You can use this method to force an immediate redraw when you want the image to be updated instantly in response to some user action like dragging a slider that controls a variable upon which the plot depends. If you call the standard method `repaint()` instead, Java might not get around to repainting the image until many frames have gone by, and the plot will appear to jump from one value to another, rather than being redrawn for every change in the variable's value.

The preceding discussion described how you can easily get display *Mathematica* output in a `MathCanvas` simply by supplying some code to `setMathCommand()`. Another way to get an image displayed in a `MathCanvas` is to create a Java `Image` object yourself and call the `setImage()` method. You might want to do this is if your image is a bitmap created with some *Mathematica* data, or if you have drawn into an offscreen image using the Java graphics API. The `setImage()` method was created mainly for use from *Mathematica* code, and it is somewhat less important for Java programmers because you already have other ways to draw into your own components. It can still be useful in Java programs, though, since it can save you from

having to write your own subclass of an AWT component just to override its `paint()` method, which is the usual technique for drawing your own content in components. When used with the `setImage()` method, a `MathCanvas` is really just a useful AWT component—it has nothing directly to do with *Mathematica*.

The next section presents a sample program that uses a `MathCanvas` to display graphics and typeset output.

A Sample Program That Displays Graphics and Typeset Results

Here is the code for a simple program that presents a window that displays *Mathematica* graphics and typeset output. It is an example of how to use the `MathCanvas` class. The code and compiled class files for this program are available in the JLink/Examples/Part2/GraphicsApp directory. Launch the program with the pathname to the kernel executable as an argument (note the use of the quote marks " and '):

```
(Windows)
java -classpath GraphicsApp.jar;..\..\..\JLink.jar GraphicsApp "c:\program
files\wolfram research\mathematica\6.0\mathkernel"

(Unix)
java -classpath GraphicsApp.jar:../../../JLink.jar GraphicsApp 'math
-mathlink'

(Mac OSX command line)
java -classpath GraphicsApp.jar:../../../JLink.jar GraphicsApp
'"/Applications/Mathematica.app/Contents/MacOS/MathKernel" -mathlink'
```

Here is the code.

```
import com.wolfram.jlink.*;
import java.awt.*;
import java.awt.event.*;

public class GraphicsApp extends Frame {

    static GraphicsApp app;
    static KernelLink ml;

    MathCanvas mathCanvas;
    TextArea inputTextArea;
```

```
    Button evalButton;
    Checkbox useFEButton;
    Checkbox graphicsButton;
    Checkbox typesetButton;

    public static void main(String[] argv) {

        try {
            String[] mlArgs = {"-linkmode", "launch", "-linkname",
argv[0]};
            ml = MathLinkFactory.createKernelLink(mlArgs);
            ml.discardAnswer();
        } catch (MathLinkException e) {
            System.out.println("An error occurred connecting to the
kernel.");
            if (ml != null)
                ml.close();
            return;
        }
        app = new GraphicsApp();
    }

    public GraphicsApp() {

        setLayout(null);
        setTitle("Graphics App");
        mathCanvas = new MathCanvas(ml);
        add(mathCanvas);
        mathCanvas.setBackground(Color.white);
        inputTextArea = new TextArea("", 2, 40,
TextArea.SCROLLBARS_VERTICAL_ONLY);
        add(inputTextArea);
        evalButton = new Button("Evaluate");
        add(evalButton);
        evalButton.addActionListener(new BnAdptr());
        useFEButton = new Checkbox("Use front end", false);
        CheckboxGroup cg = new CheckboxGroup();
        graphicsButton = new Checkbox("Show graphics output", true, cg);
        typesetButton = new Checkbox("Show typeset result", false, cg);
        add(useFEButton);
        add(graphicsButton);
        add(typesetButton);
```

```
        setSize(300, 400);
        setLocation(100,100);
        mathCanvas.setBounds(10, 25, 280, 240);
        inputTextArea.setBounds(10, 270, 210, 60);
        evalButton.setBounds(230, 290, 60, 30);
        graphicsButton.setBounds(20, 340, 160, 20);
        typesetButton.setBounds(20, 365, 160, 20);
        useFEButton.setBounds(180, 340, 100, 20);

        addWindowListener(new WnAdptr());
        setBackground(Color.lightGray);
        setResizable(false);

        // Although this code would automatically be called in
        // evaluateToImage or evaluateToTypeset, it can cause the
        // front end window to come in front of this Java window.
        // Thus, it is best to get it out of the way at the start
        // and call toFront to put this window back in front.
        // KernelLink.PACKAGE_CONTEXT is just "JLink`", but it is
        // preferable to use this symbolic constant instead of
        // hard-coding the package context.
        ml.evaluateToInputForm("Needs[\"" + KernelLink.PACKAGE_CONTEXT +
"\"]", 0);
        ml.evaluateToInputForm("ConnectToFrontEnd[]", 0);

        setVisible(true);
        toFront();
    }


    class BnAdptr implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            mathCanvas.setImageType(
                graphicsButton.getState() ? MathCanvas.GRAPHICS :
MathCanvas.TYPESET);
            mathCanvas.setUsesFE(useFEButton.getState());
            mathCanvas.setMathCommand(inputTextArea.getText());
        }
    }

    class WnAdptr extends WindowAdapter {
```

```
            public void windowClosing(WindowEvent event) {
                if (ml != null) {
                    // Because we used the front end, it is important
                    // to call CloseFrontEnd[] before closing the link.
                    // Counterintuitively, this is not because we want
                    // to force the front end to quit, but because we
                    // _don't_ want to do this if the user has begun
                    // working in the front end session we started.
                    // CloseFrontEnd knows how to politely disengage
                    // from the front end if necessary. The need for
                    // this will go away in future releases of
                    // Mathematica.
                    ml.evaluateToInputForm("CloseFrontEnd[]", 0);
                    ml.close();
                }
                dispose();
                System.exit(0);
            }
        }
    }
```

evaluateToImage() and evaluateToTypeset()

If the `MathCanvas` or `MathGraphicsJPanel` classes described in the preceding two sections are not suitable for your needs, you can manually produce images of *Mathematica* graphics and typeset expressions using the `evaluateToImage()` and `evaluateToTypeset()` methods in the `KernelLink` interface.

There are multiple signatures for each. For `evaluateToImage()`, one set takes a simpler argument list and uses default values for the less commonly used arguments. Here are graphics and typesetting methods from the KernelLink interface:

```
byte[] evaluateToImage(String s, int width, int height);
byte[] evaluateToImage(Expr e, int width, int height);
byte[] evaluateToImage(String s, int width, int height,
                       int dpi, boolean useFrontEnd);
byte[] evaluateToImage(Expr e, int width, int height,
                       int dpi, boolean useFrontEnd);

byte[] evaluateToTypeset(String s, int pageWidth, boolean useStdForm);
byte[] evaluateToTypeset(Expr e, int pageWidth, boolean useStdForm);
```

The `evaluateToImage()` method takes the input as a string or `Expr`, and a width and height of the resulting graphic in pixels. The extended versions let you specify a dots-per-inch value, and whether to use the notebook front end or not (as discussed later). The short versions use the values of 0 for the dpi and `false` for whether to use the front end. Specifying 0 for dpi causes *Mathematica* to use its default value. The image will be sized to fit within a box of width x height, without changing its aspect ratio. In other words, the image might not have exactly these dimensions, but it will never be larger in either dimension and it will never be stretched in one dimension to make it fit better. Pass 0 for the width and height to get their `Automatic` values. If the input does not evaluate to a graphics expression, then `null` is returned. It is not enough that the computation causes a plot to be generated—the *return value* of the computation must have head `Graphics` (or `Graphics3D`, etc.). If the `useFrontEnd` argument is true, `evaluateToImage()` will launch the notebook front end if it is not already running. Note that the `useFrontEnd` argument is irrelevant when using *Mathematica* 5.1 and later—the front end is always used for graphics.

The `evaluateToTypeset()` method takes the input as a string or `Expr`, a page width to wrap the output to before it is typeset, and a flag specifying whether to use `StandardForm` or `TraditionalForm`. The units for the page width is pixels (use 0 for a page width of infinity). The `evaluateToTypeset()` method requires the services of the notebook front end, which will be launched if it is not already running.

The result of both of these methods is a byte array of GIF data. The GIF format is wellsuited to most *Mathematica* graphics, but for some 3D graphics the color usage is not ideal. If you want to change to using JPEG format, you can set `$DefaultImageFormat` to `"JPEG"` in the kernel:

```
// Specifies JPEG format for subsequent calls to evaluateToImage()
// and evaluateToTypeset().
ml.evaluateToOutputForm("$DefaultImageFormat = \"JPEG\"", 0);
```

These methods are like `evaluateToInputForm()` and `evaluateToOutputForm()` in that they perform the computation and return the result in a single step. Together, all these methods are referred to as the "evaluateTo" methods. They all return `null` in the unlikely event that a `MathLinkException` occurred.

The `MathCanvas` and `MathGraphicsJPanel` classes use these methods internally, so their source code is a good place to look for examples of calling the methods. The `MathCanvas` code demonstrates how to take the byte array of GIF or JPEG data and turn it into a Java `Image` for display.

The following `Typesetter` sample program is another example. It takes a *Mathematica* expression supplied on the command line, calls `evaluateToTypeset()`, and writes the image data out to a GIF file. You would invoke it from the command line like this:

```
(Windows)
java Typesetter "c:\program files\wolfram
research\mathematica\6.0\mathkernel" "Sqrt[z]" test.gif

(Unix)
java Typesetter 'math -mathlink' "Sqrt[z]" test.gif

(Mac OSX command line)
java Typesetter '"/Applications/Mathematica.app/Contents/MacOS/MathKernel"
-mathlink' "Sqrt[z]" test.gif
```

The first argument is the command line to launch the *Mathematica* kernel, the second argument is the expression to typeset, and the third argument is the filename to create. This program is not intended to be particularly useful—it is just a simple demonstration.

```
import com.wolfram.jlink.*;
import java.io.*;

public class Typesetter {

    public static void main(String[] argv) throws MathLinkException {

        KernelLink ml;
        try {
            String[] mlArgs = {"-linkmode", "launch", "-linkname",
argv[0]};
            ml = MathLinkFactory.createKernelLink(mlArgs);
            ml.discardAnswer();
        } catch (MathLinkException e) {
            System.err.println("FATAL ERROR: link creation failed.");
            return;
        }
        byte[] gifData = ml.evaluateToTypeset(argv[1], 0, false);
        try {
            FileOutputStream s = new FileOutputStream(new File(argv[2]));
            s.write(gifData);
            s.close();
        } catch (IOException e) {}
        // ALWAYS execute CloseFrontEnd[] before killing the kernel if you
used
        // evaluateToTypeset(), or evaluateToImage() with the useFE
parameter
        // set to true:
        ml.evaluateToOutputForm("CloseFrontEnd[]", 0);
        ml.close();
    }
}
```

It is very important to note that you execute `CloseFrontEnd[]` before closing the link to the kernel. This is essential to prevent the front end from quitting in circumstances where it should not—specifically, if an already-running copy was used and the user has open documents.

Aborting and Interrupting Computations

J/Link provides two ways in which you can interrupt or abort computations. The first technique uses the low-level `putMessage()` function to send the desired *MathLink* message. The second and preferred technique is to use a new set of `KernelLink` methods introduced in *J/Link* 2.0. These are listed as follows:

```
void abortEvaluation();
void interruptEvaluation()
void abandonEvaluation();
void terminateKernel();
```

The `abortEvaluation()` method will send an abort request to *Mathematica*, identical to what happens in the notebook front end when you select **Evaluation ▸ Abort Evaluation**. *Mathematica* responds to this command by terminating the current evaluation and returning the symbol `$Aborted`. Be aware that sometimes the kernel is in a state where it cannot respond immediately to interrupts or aborts.

The `interruptEvaluation()` method will send an interrupt request to *Mathematica*, identical to what happens in the notebook front end when you select **Evaluation ▸ Interrupt Evaluation**. *Mathematica* responds to this command by interrupting the current evaluation and sending back a special packet that contains choices for what to do next. The choices can depend on what the kernel is doing at the moment, but in most cases they include aborting, continuing, or entering a dialog. It is not likely that you will want to have to deal with this list of choices on your own, so you might choose instead to call `abortEvaluation()` and just stop the computation. If you are developing an interactive front end, however, you might decide that you want your users to see the same types of choices that the notebook front end provides. If this is the case, then you can use the new `InterruptDialog` class, which provides a dialog box very similar to the front end's **Interrupt Evaluation** dialog. The `InterruptDialog` class is discussed in a later section.

The `abandonEvaluation()` method does exactly what its name suggests—it causes any command that is currently waiting for something to arrive on the link to back out immediately and throw a `MathLinkException`. This `MathLinkException` is recoverable (meaning that `clearError()` will return `true`), so in theory you could call `waitForAnswer()` again later and get the result when it arrives. In practice, however, you should generally not use this method unless you plan to close the link. You should think of `abandonEvaluation()` method is an "emergency exit" function that lets your program back out of waiting for a result no matter what state the kernel is in. Remember that the `abortEvaluation()` method simply sends an abort request to *Mathematica*, and thus it requires some cooperation from the kernel; there is no guarantee that the current evaluation will abort in a timely manner, if ever. If you call `close()` right after `abandonEvaluation()`, the kernel will typically not die, because it is still busy with a computation. You should call `terminateKernel()` before `close()` to ensure that the kernel shuts down. A code fragment that follows demonstrates this.

The terminateKernel() method will send a terminate request to *Mathematica*. It does this by sending the low-level *MathLink* message MLTERMINATEMESSAGE. This is the strongest step you can take to tell the kernel to shut down, short of killing the kernel process with operating system commands. In "normal" operation of the kernel, when you call close() on the link, the kernel will quit. In some cases, however, generally only if the kernel is currently busy computing, it will not quit. In such cases you can generally force the kernel to quit immediately by calling terminateKernel(). You should always call close() immediately afterward. In a server environment, where a Java program that starts and stops *Mathematica* kernels needs to run unattended for a very long period of time with the highest reliability possible, you might consider always calling terminateKernel() before close(), if there is any chance that close() needs to be called while the kernel is still busy. In some rare circumstances (generally only if something has gone wrong with *Mathematica*), even calling terminateKernel() will not force the kernel to quit, and you might need to use facilities of your operating system (perhaps invoked via Java's Runtime.exec() method) to kill the kernel process.

If you want to be able to abort, interrupt, or abandon computations, your program will need to have at least two threads. The thread on which the computation is called will probably look like all the sample programs you have seen. You would call one of the "evaluateTo" methods, or perhaps evaluate() followed by waitForAnswer(). This thread will block, waiting for the result. On a separate thread, such as the user interface thread, you could periodically check for some event, like a time out period elapsing. Or, you could use an event listener to be notified when the Esc key was pressed. Whichever way you want to detect the abort request, all you need to do is call putMessage(MathLink.MLABORTMESSAGE). If the kernel receives the message before it finishes, and it is doing something that can be aborted, the computation will end and return the symbol $Aborted. You typically will not need to do anything special in the computation thread. You wait for the answer as usual; it might come back as $Aborted instead of the final result, that is all. Here are some typical code fragments that demonstrate aborting a computation:

```
// On thread 1
ml.evaluate("Do[2+2, {20000000}]");
ml.waitForAnswer();
// If user aborted, the result will be the symbol $Aborted.

// On thread 2
if (userPressedEscKey() || timeoutElapsed())
    ml.abortEvaluation();
```

Here is some code that demonstrates how to abandon a computation and force an immediate shutdown of the kernel:

```
// On thread 1
try {
    ml.evaluate("While[True]");
    ml.discardAnswer();
} catch (MathLinkException e) {
    // We will get here when abandonEvaluation() is called on the other
thread.
    System.err.println("MathLinkException occurred: " + e.toString());
    if (!ml.clearError()) {
        // clearError() will always fail when abandonEvaluation() was
called.
        ml.terminateKernel();
        ml.close();
    }
}

// On thread 2
if (timeoutElapsedAndReallyNeedToShutdownKernel())
    ml.abandonEvaluation();
```

The discussion so far has focused on the high-level interface for interrupting and aborting computations. The alternative is to use the low-level method `putMessage()` and pass one of the constants `MathLink.MLINTERRUPTMESSAGE`, `MathLink.MLABORTMESSAGE`, or `MathLink.MLTERMINATEMESSAGE`. There is no reason to do this, however, as `interruptEvaluation()`, `abortEvaluation()`, and `terminateKernel()` are just one-line methods that put the appropriate message. The "messages" referred to in the `MathLink` method `putMessage()` are not related to the familiar *Mathematica* error and warning messages. Instead, they are a special type of communication between two *MathLink* programs. This communication takes place on a different channel from the normal flow of expressions, which is why you can call `putMessage()` while the kernel is in the middle of a computation and not reading from the link.

There are several other `MathLink` methods with "message" in their names. These are `messageReady()`, `getMessage()`, `addMessageHandler()`, and `removeMessageHandler()`. These methods are only useful if you want to be able to detect messages the kernel sends to you. *J/Link* programmers will rarely want to do this, so they are not discussed in detail. Please note that `messageReady()` and `getMessage()` **no longer function in *J/Link* 2.0 and later**.

If you want to be able to receive messages from *Mathematica,* you must use `addMessageHandler()` and `removeMessageHandler()`. There is more information in the JavaDocs for these methods.

Using Marks

MathLink allows you to set a "mark" in a link, so that you can read more data and then seek back to the mark, restoring the link to the state it was in before you read the data. Thus, marks let you read data off a link and not have the data consumed, so you can read it again later. There are three mark-related methods in the `MathLink` interface:

```
// In the MathLink interface:
long createMark() throws MathLinkException;
void seekMark(long mark);
void destroyMark(long mark);
```

One common reason to use a mark is if you want to examine an incoming expression and branch to different code depending on some property of the expression. You want the code that actually handles the expression to see the entire expression, but you will need to read at least a little bit of the expression to decide how it must be handled (perhaps just calling `getFunction()` to see the head). Here is a code fragment demonstrating this technique:

```
String head = null;
long mark = ml.createMark();
try {
    head = ml.getFunction().name;
    ml.seekMark(mark);
} finally {
    ml.destroyMark(mark);
}
if (head.equals("foo"))
    handleFoo(ml);
else if (head.equals("bar"))
    handleBar(ml);
```

Because you seek back to the mark after calling `getFunction()`, the link will be reset to the beginning of the expression when the `handleFoo()` and `handleBar()` methods are entered. Note the use of a `try/finally` block to ensure that the mark is always destroyed, whether or not an exception of any kind is thrown after it is created. You should always use marks in this way. Right after calling `createMark()`, start a `try` block whose `finally` clause calls `destroy`

`Mark()`. It is important that no other code intervenes between `createMark()` and the `try` block, especially *MathLink* calls (which can throw `MathLinkException`). If a mark is created and not destroyed, a memory leak will result because incoming data will pile up on the link, never to be freed.

Another common use for marks is to allow you to read an expression one way, and if a `MathLinkException` is thrown, go back and try reading it a different way. For example, you might be expecting a list of real numbers to be waiting on the link. You can set a mark and then call `getDoubleArray1()`. If the data on the link cannot be coerced to a list of reals, `getDoubleArray1()` will throw a `MathLinkException`. You can then seek back to the mark and try a different method of reading the data.

```
double[] data = null;
long mark = ml.createMark();
ty {
    data = ml.getDoubleArray1();
} catch (MathLinkException e) {
    ml.clearError();
    ml.seekMark(mark);
    // Here, try a different way of reading the data:
    switch (ml.getNext()) {
        ...
    }
} finally {
    ml.destroyMark(mark);
}
```

Much of the functionality of marks is subsumed by the `Expr` class, described in "Motivation for the Expr Class". `Expr` objects allow you to easily examine an expression over and over in different ways, and with the `peekExpr()` method you can look at the upcoming expression without consuming it off the link.

Using Loopback Links

In addition to the MathLink and KernelLink interfaces, there is one other link interface: LoopbackLink. Loopback links are a feature of *MathLink* that allow a program to conveniently store *Mathematica* expressions. Say you want to read an expression off a link, keep it around for awhile, and then write it back onto the same or a different link. How would you do this? If you read it with the standard reading functions (getFunction(), getInteger(), and so on), you will have broken the expression down into its atomic components, of which there might be very many. Then you will have to reconstruct it later with the corresponding series of "put" methods. What you really need is a temporary place to transfer the expression in its entirety, where it can be read later or transferred again to a different link. A loopback link serves this purpose.

Before proceeding to examine loopback links, please note that *J/Link*'s Expr class is used for the same sorts of things that a loopback link is used for. Expr objects use loopback links internally, and are a much richer extension of the functionality that loopback links provide. You should consider using Expr objects instead of loopback links in your programs.

If a MathLink is like a pipe, then a loopback link is a pipe that bends around to point back at you. You manage both ends of the link, writing into one "end" and reading out the other, in FIFO order. To create a loopback link in *J/Link*, use the MathLinkFactory method createLoopbackLink():

```
// In class MathLinkFactory:
public static LoopbackLink createLoopbackLink() throws MathLinkException;
```

The LoopbackLink interface extends the MathLink interface, so all the MathLink methods can be used on loopback links. LoopbackLink adds no methods beyond those in the MathLink interface. Why have a separate interface then? It can be useful to have a separate type for this kind of *MathLink*, because it has different behavior than a normal one-sided *MathLink*. Furthermore, there is one method in the MathLink interface (transferToEndOfLoopbackLink()) that requires, as an argument, a loopback link. Thus, it provides a small measure of type safety within *J/Link* and your own programs to have a separate LoopbackLink type.

You will probably use the MathLink method transferExpression(), or its variant transferToEndOfLoopbackLink(), in conjunction with loopback links. You will need transferExpression() either to move an expression from another link onto a loopback link or to move an expression you have manually placed on a loopback link onto another link. Here are the declarations of these two methods:

```
// In the MathLink interface
void transferExpression(MathLink source) throws MathLinkException;
void transferToEndOfLoopbackLink(LoopbackLink source) throws
MathLinkException;
```

Note that the source link is the argument and the destination is the `this` link. The `transferExpression()` method reads one expression from the source link and puts it on the destination link, and the `transferToEndOfLoopbackLink()` method moves all the expressions on the source link (which must be a `LoopbackLink`) to the destination link.

Already mentioned is a common case where loopback links are convenient—temporary storage of an expression for later writing to a different link. This is done more simply using an `Expr` object, however ("Motivation for the Expr Class"). Another use for loopback links is to allow you to begin sending an expression before you know how long it will be. Recall that the `putFunction()` method requires you to specify the number of arguments (i.e., the length). There are times, though, when you do not know ahead of time how long the expression will be. Consider the following code fragment. You need to send a list of random numbers to *Mathematica*, the length of which depends on a test whose outcome cannot be known at compile time. You can create a loopback link and push the numbers onto it as they are generated, counting them as you go. When the loop finishes, you know how many were generated, so you call `putFunction()` and then just "pour out" the contents of the loopback link onto the destination link. In this example, it would be easy to store the accumulating numbers in a Java array or `Vector` rather than a loopback link. But if you were sending complicated expressions it might not be so easy to store them in native Java structures. It is often easier just to write them on a link as you go, and leave the storage issues up to the internals of *MathLink*.

```
// Here we demonstrate sending an expression (a list of reals)
// whose length is unknown at the start.
try {
    ...
    LoopbackLink loop = MathLinkFactory.createLoopbackLink();
    int count = 0;
    while (someTest) {
        loop.put(Math.random());
        count++;
    }
    ml.putFunction("List", count);
    ml.transferToEndOfLoopbackLink(loop);
    loop.close();
    ...
} catch (MathLinkException e) {}
```

Using Expr Objects

Motivation for the Expr Class

The `Expr` class provides a direct representation of *Mathematica* expressions in Java. You can guess that this will be useful, since everything in *Mathematica* is an expression and *MathLink* is all about communicating *Mathematica* expressions between programs.

You have several ways of handling *Mathematica* expressions in a *MathLink* program. First, you can send and/or receive them as strings. This is often convenient, particularly if you are taking input typed by a user, or displaying results to the user. Many of the `KernelLink` methods can take input as a string and return the result as a string. A second way of handling *Mathematica* expressions is to put them on the link or read them off the link a piece at a time with a series of "put" or "get" calls. A third way is to store them on a loopback link and shuttle them around between links. Each of these methods has advantages and disadvantages.

Loopback links were described in the previous section, but it is worthwhile to summarize them here, as it provides some of the background to understanding the motivation for the `Expr` class. Basically, a loopback link provides a means to store a *Mathematica* expression without having tediously to read it off the link, disassembling it into its component atoms in the process. Loopback links, then, let you *store* expressions for later reading or just dumping onto another link. If you eventually want to read and examine the expression, however, you are still stuck with the difficult task of dissecting an arbitrary expression off a link with the correct sequence of "get" calls. This is where the `Expr` class comes in. Like a loopback link, an `Expr` object stores an arbitrary *Mathematica* expression. The `Expr` class goes further, though, and provides a set of methods for examining the structure of the expression, extracting parts of it, and building new ones. The names and operation of these methods will be familiar to *Mathematica* programmers: `head()`, `length()`, `dimensions()`, `part()`, `stringQ()`, `vectorQ()`, `matrixQ()`, `insert()`, `delete()`, and many others.

The advantage of an `Expr` over a loopback link, then, is that you are not restricted to using the low-level *MathLink* API for examining an expression. Consider the task of receiving an arbitrary expression from *Mathematica* and determining if its element at position [[2, 3]] (in *Mathematica* notation) is a vector (a list with no sublists). This can be done with an `Expr` object as follows:

```
ml.evaluate("some code");
ml.waitForAnswer();
Expr e = ml.getExpr();
Expr part23 = e.part(new int[] {2, 3});
boolean isVector = part23.vectorQ();
```

This task would be much more difficult with the *MathLink* API. The `Expr` class provides a minimal *Mathematica*-like functional interface for examining and dissecting expressions.

Methods in the *MathLink* Interface for Reading and Writing Exprs

There are three methods in the `MathLink` interface for dealing with `Expr` objects. This is in addition to the numerous methods in the `Expr` class itself, which deal with composing and decomposing `Expr` objects. The `getExpr()` and `peekExpr()` methods read an expression off a link, but `peekExpr()` resets the link to the beginning of the expression—it "peeks" ahead at the upcoming expression without consuming it. This is quite useful for debugging. The `put()` method will send an `Expr` object as its corresponding *Mathematica* expression.

```
// In the MathLink interface:
Expr getExpr() throws MathLinkException;
Expr peekExpr() throws MathLinkException;
void put(Object obj) throws MathLinkException;
```

Exprs as Replacements for Loopback Links

One way to use `Expr` is as a simple replacement for a loopback link. You can use the `MathLink` method `getExpr()` to read any type of expression off a link and store it in the resulting `Expr` object. To write the expression onto a link, use the `put()` method. Compare the following two code fragments:

```
// Old way, using a loopback link
    LoopbackLink loop = MathLinkFactory.createLoopbackLink();
    // Read expr off of link and store it on loopback
    loop.transferExpression(ml);
    ...
    // Later, write the expr back on the link
    ml.transferExpression(loop);
    loop.close();

// New way, using an Expr
    Expr e = ml.getExpr();
    ...
    // Later, write the expression back on the link
    ml.put(e);
    e.dispose();
```

Note the call to `dispose()` at the end. The `dispose()` method tells an `Expr` object to release certain resources that it might be using internally. You should generally use `dispose()` on an `Expr` when you are finished with it. The `dispose()` method is discussed in more detail in "Disposing of Exprs".

Exprs as a Means to Get String Representations of Expressions

A particularly useful method in the `Expr` class is `toString()`, which produces a string representation of the expression similar to `InputForm` (without involving the kernel, of course). This is particularly handy for debugging purposes, when you want a quick way to see what is arriving on the link. In "The PacketPrinter Class for Debugging" it was mentioned that *J/Link* has a class `PacketPrinter` that implements the `PacketListener` interface and can be used to easily print out the contents of packets as they arrive in your program, without modifying your program. Following is the `packetArrived()` method of that class, which uses an `Expr` object and its `toString()` method to get the printable text representation of an arbitrary expression.

```
public boolean packetArrived(PacketArrivedEvent evt) throws
MathLinkException {
    KernelLink ml = (KernelLink) evt.getSource();
    Expr e = ml.getExpr();
    strm.println("Packet type was " + pktToName(evt.getPktType()) +
                 ". Contents follows.");
    strm.println(e.toString());
    e.dispose();
    return true;
}
```

Whether you use the PacketPrinter class or not, this technique is useful to see what expressions are being passed around. This is often used in conjunction with the MathLink peekExpr() method, which reads an expression off the link but then resets the link so that the expression is not consumed. In this way, you can look at expressions arriving on links without interfering with the rest of the link-reading code in your program. The PacketPrinter code shown does not use peekExpr(), but it has the same effect since the resetting of the link is handled elsewhere.

Exprs as Arguments to KernelLink Methods

The KernelLink methods evaluate(), evaluateToInputForm(), evaluateToOutputForm(), evaluateToImage(), and evaluateToTypeset() take the *Mathematica* expression to evaluate as either a string or an Expr. "The 'evaluateTo' Methods" discusses why and how you would use an Expr object to provide the input instead of a string. This examines one trivial example comparing how you would send 2+2 to *Mathematica* as both a string and as an Expr. In the Expr case you build the expression on a loopback link and then read the Expr off this link. For all but the simplest expressions, this is generally easier than trying to use the Expr constructors.

```
// Send input as a string:
    String result = MathLink.evaluateToOutputForm("2+2", 0);

// Send input as an Expr:
    LoopbackLink loop = MathLinkFactory.createLoopbackLink();
    // Create the expression 2+2 on the loopback link
    loop.putFunction("Plus", 2);
        loop.put(2);
        loop.put(2);
    loop.endPacket();
    // Now read the Expr off the loopback link
    Expr e = loop.getExpr();
    // We are done with the loopback link now.
    loop.close();
    String result = ml.evaluateToOutputForm(e, 0);
    e.dispose();
```

Examining and Manipulating Exprs

Like expressions in *Mathematica*, `Expr` objects are *immutable*, meaning that they cannot be modified once they have been created. Operations that might appear to modify an `Expr`, like the `insert()` method, actually copy the original , modify this copy, and then return a new immutable object. One consequence of being immutable is that the `Expr` class is *thread-safe*—multiple threads can operate on the same `Expr` without worrying about synchronization.

The `Expr` class provides a minimal *Mathematica*-like API for examination and manipulation. The functions are generally named after their *Mathematica* counterparts, and they operate in the same way. This section will only provide a brief review of the `Expr` API. Consult the JavaDocs (found in the JLink/Documentation/JavaDoc directory) for more information about these methods.

Here are some methods for learning about the structure of an `Expr`:

```
public Expr head();
public Expr[] args();
public int length();
public int[] dimensions();
```

There are a number of methods whose names end in "Q", following the same naming pattern as in *Mathematica* for functions that return `true` or `false`. This is not the complete list:

```
// A sampling of the "Q" methods
public boolean atomQ();
public boolean stringQ();
public boolean integerQ();
public boolean numberQ();
public boolean trueQ();
public boolean listQ();
public boolean vectorQ();
public boolean matrixQ();
```

There are several methods for taking apart and building up an `Expr`. Like in *Mathematica*, part numbers and indices are 1-based. You can also supply negative numbers to count backward from the end. Many `Expr` methods throw an `IllegalArgumentException` if they are called with invalid input, such as a part index larger than the length of the `Expr`. These exceptions parallel the *Mathematica* error messages you would get if you made the same error in *Mathematica* code.

```
public Expr part(int index);
public Expr part(int[] indices);
public Expr take(int n);
public Expr delete(int n);
public Expr insert(Expr e, int n);
```

Here is some very simple code that demonstrates a few `Expr` operations.

```
ml.evaluate("Expand[(x + y)^4]");
ml.waitForAnswer();
Expr e1 = ml.getExpr();

System.out.println("e1 is:  " + e1.toString());
System.out.println("the length is:  " + e1.length());
System.out.println("the head is:  " + e1.head().toString());
System.out.println("part [[2]] is:  " + e1.part(2));
System.out.println("part [[-1]] is:  " + e1.part(-1));
System.out.println("part [[2, 2]] is:  " + e1.part(new int[]{2, 2}));
System.out.println("drop the last element:  " + e1.delete(-1).toString());
System.out.println("e1 is unmodified:  " + e1.toString());
Expr e2 = e1.insert(new Expr(new double[] {1.0, 2.0, 3.0}), 1);
System.out.println("e2 is:  " + e2.toString());
```

That code prints the following:

```
e1 is:
Plus[Power[x,3],Times[3,Power[x,2],y],Times[3,x,Power[y,2]],Power[y,3]]
the length is:  4
the head is:  Plus
part [[2]] is:  Times[3,Power[x,2],y]
part [[-1]] is:  Power[y,3]
part [[2, 2]] is:  Power[x,2]
drop the last element:
Plus[Power[x,3],Times[3,Power[x,2],y],Times[3,x,Power[y,2]]]
e1 is unmodified:
Plus[Power[x,3],Times[3,Power[x,2],y],Times[3,x,Power[y,2]],Power[y,3]]
e2 is:
Plus[{1.0,2.0,3.0},Power[x,3],Times[3,Power[x,2],y],Times[3,x,Power[y,2]],P
ower[y,3]]
```

Disposing of Exprs

You have seen the `dispose()` method used frequently in this discussion of the `Expr` class. An `Expr` object might make use of a loopback *MathLink* internally, and any time a Java class holds such a non-Java resource it is important to provide programmers with a `dispose()` method that causes the resource be released. Although the finalizer for the `Expr` class will call `dispose()`, you cannot rely on the finalizer ever being called. Although it is good style to always call `dispose()` on an `Expr` when you are finished using it, you should know that many of the operations you can perform on an `Expr` will cause it to be "unwound" off its internal loopback link and cause that link to be closed. After this happens, the `dispose()` method is unnecessary. Calling the `toString()` method is an example of an operation that makes `dispose()` unnecessary, and in fact virtually any operation that queries the structure of an `Expr` or extracts a part will have this effect. This is useful to know since it allows shorthand code like this:

```
System.out.println("the result was " + ml.getExpr().toString());
```

instead of the more verbose:

```
Expr e = ml.getExpr();
System.out.println("the result was " + e.toString());
e.dispose();
```

You should get in the habit of calling `dispose()` explicitly on `Expr` objects. In cases where it is inconvenient to store an `Expr` in a named variable, and you know that the `Expr` does not need to be disposed, then you can skip calling it.

Because extracting any piece of an existing expression will make `dispose()` unnecessary, you do not have to worry about calling `dispose()` on `Expr` objects that are obtained as parts of another one:

```
Expr e = ml.getExpr();
// The moment that head() or part() are called on e below, you know that
neither
// e, e2, nor e3 need to be disposed.
Expr e2 = e.head();
Expr e3 = e.part(1);
```

You cannot reliably use an `Expr` object after `dispose()` has been called on it. You have already seen that `dispose()` is often unnecessary because many `Expr` objects have already had their

internal loopback links closed. For such an Expr, dispose() will have no effect at all and there would be no problem continuing to use the Expr after dispose() had been called. That being said, it is *horrible* style to ever try to use an Expr after calling dispose(). A call to dispose() should always be an unambiguous indicator that you have no further use for the given Expr or any part of it.

Threads, Blocking, and Yielding

The classes that implement the MathLink and KernelLink interfaces are not thread-safe. This means that if you write a *J/Link* program in which one link object is used by more than one thread, you need to pay careful attention to concurrency issues. The relevant methods in the link implementation classes are synchronized, so at the individual method level there is no chance that two threads can try to use the link at the same time. However, this is not enough to guarantee thread safety because interactions with the link typically involve an entire transaction, encompassing a multistep write and read of the result. This entire transaction must be guarded. This is done by using synchronized blocks to ensure that the threads do not interfere with each other's use of the link.

The "evaluateTo" methods are synchronized, and they encapsulate an entire transaction within one call, so if you use only these methods you will have no concerns. On the other hand, if you use evaluate() and waitForAnswer(), or any other technique that splits up a single transaction across multiple method calls, you should wrap the transaction in a synchronized block, as follows:

```
synchronized (ml) {
    ml.evaluate("2+2");
    ml.waitForAnswer();
    int result = ml.getInteger();
}
```

Synchronization is only an issue if you have multiple threads using the same link.

J/Link functions that read from a link will block until data arrives on that link. For example, when you call evaluateToOutputForm(), it will not return until *Mathematica* has computed and returned the result. This might be a problem if the thread on which evaluateToOutputForm() was called needs to stay active—for example, if it is the AWT thread, which processes user interface events.

How to handle blocking is a general programming problem, and there are a number of solutions. The Java environment is multithreaded, and thus an obvious solution is simply to make *J/Link* calls on a thread that does not need to be continuously responsive to other events in the system.

MathLink has the notion of a "yield function," which is a function you can designate to be called from the internals of *MathLink* while *MathLink* is blocking, waiting for input to arrive from the other side. A primary use for yield functions was to solve the blocking problem on operating systems that did not have threads, or for programming languages that did not have portable threading libraries. The way this would typically work is that your single-threaded program would install a yield function that ran its main event loop, so that the program could process user interface events even while it was waiting for *MathLink* data.

With Java, this motivation goes away. Rather than using a yield function to allow your program's only thread to still handle events while blocking, you simply start a separate thread from the user interface thread and let it happily block inside *J/Link* calls. Despite the limited usefulness of yield functions in Java programs, *J/Link* provides the ability to use them anyway.

```
// From the MathLink interface
public boolean setYieldFunction(Class cls, Object obj, String methName);
```

The `setYieldFunction()` method in the `MathLink` interface takes three arguments that identify the function to be called. These arguments are designed to accommodate static and nonstatic methods, so only two of the three need to be specified. For a static method, supply the method's `Class` and its name, leaving the second argument `null`. For a nonstatic method, supply the object on which you want the method called and the method's name, leaving the `Class` argument `null`. The function must be public, take no arguments, and return a `boolean`, for example:

```
public boolean myYielder();
```

The function you specify will be called periodically while *J/Link* is blocking in a call that tries to read from the link. The return value is used to indicate whether *J/Link* should back out of the read call and return right away. Backing out of a read call will cause a `MathLinkException` to be thrown by the method that is reading from the link. This `MathLinkException` is recoverable (meaning that `clearError()` will return `true`), so you could call `waitForAnswer()` again later and get the result when it arrives if you want. Return `false` from the yield function to indicate

that no action should be taken (thus `false` is the normal return value for a yield function), and return `true` to indicate that *J/Link* should back out of the reading call. To turn off the yield function, call `setYieldFunction(null, null, null)`.

Very few *J/Link* programmers will have any need for yield functions. They are a solution to a problem that is better handled in Java by using multiple threads. About the only reasonable motivation for using a yield function is to be able to back out of a computation that is taking too long and either resists attempts to abort it, or you know you want to close the link anyway. This can also be done by calling `abandonEvaluation()` on a separate thread. The `abandonEvaluation()` method is described in "Aborting and Interrupting Computations". Note that `abandonEvaluation()` uses a yield function internally, so calling it will wipe out any yield function you might have installed on your own.

Sending Object References to Mathematica

The first part of this User Guide describes how to use *J/Link* to allow *Mathematica* code to launch a Java runtime, load Java classes and directly execute Java methods. What this means for you, the reader of this tutorial, who is probably writing his or her own program to launch and communicate with the *Mathematica* kernel, is that you can have a very high-level interaction with *Mathematica*. You can send your own objects to *Mathematica* and use them in *Mathematica* code, but you have to take a special step to enable this type of interaction.

Consider what happens if you have written a Java front end to the *Mathematica* kernel and a user of your program calls a *Mathematica* function that uses the "installable Java" features of *J/Link* and thus calls `InstallJava` in *Mathematica*. `InstallJava` launches a separate Java runtime and proceeds to direct all *J/Link* traffic to that Java runtime. The kernel is blissfully unconcerned whether the front end that is driving it is the notebook front end or your Java program—it does the same thing in each case. This is fine and it is what many *J/Link* programmers will want. You do not have to worry about doing anything special if some *Mathematica* code happens to invoke the "installable Java" features of *J/Link*, because a separate Java runtime will be used.

But what if *you* want to make use of the ability that *J/Link* gives *Mathematica* code to interact with Java objects? You might want to send Java object references to *Mathematica* and operate on them with *Mathematica* code. *Mathematica* "operates" on Java objects by calling into Java, so any callbacks for such objects must be directed to *your* Java runtime. A further detail of

J/Link is that it only supports one active Java runtime for all installable Java uses. What this all adds up to is that if you want to pass references to your own objects into *Mathematica*, then you must call `InstallJava` and specify the link to *your* Java runtime, and you must do this before any function is executed that itself calls `InstallJava`. Actually, a number of steps need to be taken to enable *J/Link* callbacks into your Java environment, so *J/Link* includes a special method in the `KernelLink` interface, `enableObjectReferences()`, that takes care of everything for you.

```
public void enableObjectReferences() throws MathLinkException;

// For sending object references:
public void put(Object obj) throws MathLinkException;
public void putReference(Object obj) throws MathLinkException;
```

After calling `enableObjectReferences()`, you can use the `KernelLink` interface's `put()` or `putReference()` methods to send Java objects to *Mathematica*, and they will arrive as `JavaObject` expressions that can be used in *Mathematica* code as described throughout "Calling Java from *Mathematica*". Recall that the difference between the `put()` and `putReference()` methods is that `put()` sends objects that have meaningful "value" representations in *Mathematica* (like arrays and strings) by value, and all others by reference. The `putReference()` method sends everything as a reference. If you want to use `enableObjectReferences()`, call it early on in your program, before you call `putReference()`. It requires that the JLink.m file be present in the expected location, which means that *J/Link* must be installed in the standard way on the machine that is running the kernel.

Once you have called `enableObjectReferences()`, not only can you send Java objects to *Mathematica*, you can also read Java object references that *Mathematica* sends back to Java. The `getObject()` method is used for this purpose. If a valid `JavaObject` expression is waiting on the link, `getObject()` will return the object that it refers to.

```
public Object getObject() throws MathLinkException;
```

If you call `enableObjectReferences()` in your program, it is imperative that you do not try to write your own packet loop. Instead, you must use the `KernelLink` methods that encapsulate the reading and handling of packets until a result is received. These methods are `waitForAnswer()`, `discardAnswer()`, `evaluateToInputForm()`, `evaluateToOutputForm()`, `evaluateToImage()`, and `evaluateToTypeset()`. If you want to see all the incoming packets yourself, use a `PacketListener` object in conjunction with one of these methods. This is discussed in "Using the PacketListener Interface".

It is worthwhile to examine in more detail the question of why you would want to use `enableObjectReferences()`. Traditionally, *MathLink* programmers have worked with the C API, which limits the types of data that can be passed back and forth between C and *Mathematica* to *Mathematica* expressions. Since *Mathematica* expressions are not generally meaningful in a C program, this translates basically to numbers, strings, and arrays of these things. The native structures that are meaningful in your C or C++ program (structs, objects, functions, and so on) are not meaningful in *Mathematica*. As a result, programmers tend to use a simplistic one-way communication with *Mathematica*, decomposing the native data structures and objects into simple components like numbers and strings. Program logic and behavior is coded entirely in C or C++, with *Mathematica* used solely for mathematical computations.

In contrast, *J/Link* allows Java and *Mathematica* code to collaborate in a high-level way. You can easily code algorithms and other program behavior in *Mathematica* if it is easier for you. As an example, say you are writing a Java servlet that needs to use the *Mathematica* kernel in some way. Your servlet's `doGet()` method will be called with `HttpServletRequest` and `HttpServletResponse` objects as arguments. One approach would be to extract the information you need out of these objects, package it up in some way for *Mathematica*, and send the desired computation for evaluation. But another approach would be simply to **send the `HttpServletRequest` and `HttpServletResponse` objects themselves** to *Mathematica*. You can then use the features and syntax described in "Calling Java from *Mathematica*" to code the behavior of the servlet in *Mathematica*, rather than in Java. Of course, these are just two extremes of a continuum. At one end you have the servlet behavior hard-coded into a compiled Java class file, and you make use of *Mathematica* in a limited way, using a very narrow pipeline (narrow in the logical sense, passing only simple things like numbers, strings, or arrays). At the other end of the continuum you have a completely generic servlet that does nothing but forward all the work into *Mathematica*. The behavior of the servlet is written completely in *Mathematica*. You can use this approach even if you do not need *Mathematica* as a mathematics engine—you might just find it easier to develop and debug your servlet logic in the *Mathematica* language. You can draw the line between Java and *Mathematica* anywhere you like along the continuum, doing whatever amount of work you prefer in each language.

In case you are wondering what such a generic servlet might look like, here is the `doGet()` method:

```
// ml.enableObjectReferences() must have been called prior, for example
// in the servlet's init method.
public void doGet(HttpServletRequest req, HttpServletResponse res)
                    throws ServletException, IOException {
    try {
        ml.putFunction("EvaluatePacket", 1);
        ml.putFunction("DoGet", 2);
        // We could also use plain 'put' here, as these objects would be
put
        // by reference anyway.
        ml.putReference(req);
        ml.putReference(res);
        ml.endPacket();
        ml.discardAnswer();
    } catch (MathLinkException e) {}
}
```

This would be accompanied by a *Mathematica* function `DoGet` that takes the two Java object arguments and implements the servlet behavior. The syntax is explained in "Calling Java from *Mathematica*":

```
doGet[req_, resp_] :=
     JavaBlock[
        Module[{outStream},
            outStream = resp@getOutputStream[];
            outStream@print["<HTML> <BODY>"];
            outStream@print["Hello World"];
            outStream@print["</BODY> </HTML>"];
        ]
    ]
]
```

Some Special User Interface Classes

Introduction

J/Link has several classes that provide some very high-level user interface components for your Java programs. They are discussed individually in the next subsections. These classes are in the new com.wolfram.jlink.ui package, so do not forget to import that package if you want to use the classes.

ConsoleWindow

The `ConsoleWindow` class gives you a top-level frame window that displays output printed to the `System.out` and/or `System.err` streams. It has no input facilities. This is the class used to implement the *Mathematica* function `ShowJavaConsole`, discussed in "The Java Console Window". This class is quite useful for debugging Java programs that do not have a convenient place for console output. An example is a servlet—rather than digging around in your servlet container's log files after every run, you can just display a `ConsoleWindow` and see debugging output as it happens.

This class is a singleton, meaning that there is only ever one instance in existence. It has no public constructors. You call the static `getInstance()` method to acquire the sole `ConsoleWindow` object. Here is a code fragment that demonstrates how to use `ConsoleWindow`. You can find more information on this class in its JavaDoc page.

```
// Don't forget to import it (a different package than the rest of
J/Link):
// import com.wolfram.jlink.ui.ConsoleWindow;

ConsoleWindow cw = ConsoleWindow.getInstance();
cw.setLocation(100, 100);
cw.setSize(450, 400);
cw.show();
// Sepcify that we want to capture System.out and System.err.
cw.setCapture(ConsoleWindow.STDOUT | ConsoleWindow.STDERR);
System.out.println("hello world from stdout");
System.err.println("hello world from stderr");
```

MathSessionPane

The `MathSessionPane` class provides an `In/Out` *Mathematica* session window complete with a full set of editing functions including cut/copy/paste/undo/redo, support for graphics, syntax coloring, and customizable font styles. It is a bit like the *Mathematica* kernel's "terminal" interface, but much more sophisticated. You can easily drop it into any Java program that needs a full command-line interface to *Mathematica*. The class is a Java Bean and will work nicely in a GUI builder environment. It has a very large number of properties that allow its look and behavior to be customized.

The best way to familiarize yourself with the features of MathSessionPane is to run the SimpleFrontEnd example program, found in the JLink/Examples/Part2/SimpleFrontEnd directory. SimpleFrontEnd is little more than a frame and menu bar that host a MathSessionPane. Essentially all the features you see are built into MathSessionPane, including the keyboard commands and the properties settable via the **Options** menu.To run this example, go to the SimpleFrontEnd directory and execute the following command line:

```
(Windows)
java -classpath SimpleFrontEnd.jar;..\..\..\JLink.jar SimpleFrontEnd

(Linux, Unix, Mac OS X):
java -classpath SimpleFrontEnd.jar:../../../JLink.jar SimpleFrontEnd
```

The application window will appear and you will be prompted to enter a path to a kernel to launch. Once *Mathematica* is running, try various computations, including plots. Experiment with the numerous settings and commands on the menus. One feature of MathSessionPane not exposed via the SimpleFrontEnd menu bar is a highly customizable syntax coloring capability. The default behavior is to color built-in *Mathematica* symbols, but you can get as fancy as you like, such as specifying that symbols from a certain list should always appear in red, and symbols from a certain package should always appear in blue.

The methods and properties of MathSessionPane are described in greater detail in the JavaDocs, which are found in the JLink/Documentation/JavaDoc directory.

BracketMatcher and SyntaxTokenizer

The auxiliary classes BracketMatcher and SyntaxTokenizer are used by MathSessionPane but can also be used separately to provide these services in your own programs. An example of the sort of program that would find these classes useful is a text-editor component that needs to have special features for *Mathematica* programmers.

These classes are described in greater detail in their JavaDoc pages. The JavaDocs for J/Link are found in the JLink/Documentation/JavaDoc directory. You can also look to see how they are used in the source code for the MathSessionPane class (MathSessionPane.java).

The BracketMatcher class locates matching bracket pairs (any of (), {}, [], and (**)) in *Mathematica* code. It ignores brackets within strings and within *Mathematica* comments, and it can accommodate nested comments. It searches in the typical way—expanding the current selec-

tion left and right to find the first enclosing matching brackets. To see its behavior in action, simply run the SimpleFrontEnd sample program discussed in the previous section on MathSessionPane and experiment with its bracket-matching feature.

SyntaxTokenizer is a utility class that can break up *Mathematica* code into 4 syntax classes: strings, comments, symbols, and normal (meaning everything else). You can use it to implement syntax coloring or a code analysis tool that can extract all comments or symbols from a file of *Mathematica* code.

InterruptDialog

The InterruptDialog class gives you an **Interrupt Evaluation** dialog box similar to the one you see in the notebook front end when you choose **Interrupt Evaluation** from the **Evaluation** menu. The dialog box that appears has choices for aborting, quitting the kernel, and so on, depending on what the kernel is doing at the time.

The InterruptDialog constructor takes a Dialog or Frame instance that will be the parent window of the dialog box. What you supply for this argument will typically be the main top-level window in your application. InterruptDialog implements the PacketListener interface, and you use it like any other PacketListener:

```
// Don't forget to import it (a different package than the rest of
J/Link):
// import com.wolfram.jlink.ui.InterruptDialog;

ml.addPacketListener(new InterruptDialog(myParentFrame));
```

After the line of code is executed, whenever you interrupt a computation (by sending an MLINTERRUPTMESSAGE or, more commonly, by calling the KernelLink interruptEvaluation() method), a modal dialog box will appear with choices for how to proceed.

The SimpleFrontEnd sample program discussed in the section "MathSessionPane" makes use of an InterruptDialog. To see it in action, launch that sample program and execute the following *Mathematica* statement:

```
While[True]
```

Then select **Interrupt Evaluation** from the **Evaluation** menu. The **Interrupt Evaluation** dialog box will appear and you can click the **Abort Command Being Evaluated** button to stop the computation. To use an InterruptDialog in your own program, your user interface must

provide a means for users to send an interrupt request, such as an Interrupt button or special key combination. In response to this action, your program would call the `KernelLink interruptEvaluation()` method.

That a behavior as complex as a complete **Interrupt Evaluation** dialog box can be plugged into a Java program with only a single line of code is a testament to the versatility of the `PacketListener` interface, described in "Using the PacketListener Interface". The `InterruptDialog` class works by monitoring the incoming flow of packets from the kernel and detecting the special type of `MenuPacket` that the kernel sends after an interrupt request. Anytime you have some application logic that needs to know about packets that arrive from *Mathematica*, you should implement it as a `PacketListener`.

Writing Applets

This User Guide has presented a lot of information about how to use *J/Link* to enable *MathLink* functionality in Java programs, whether those Java programs are applications, JavaBeans, servlets, applets, or anything else. If you want to write an applet that makes use of a local *Mathematica* kernel, you have some special considerations because you will need to escape the Java security "sandbox" within which the browser runs applets.

The only thing that *J/Link* needs special browser security permission for is to load the *J/Link* native library. The only reason the native library is required, or even exists at all, is to perform the translation between Java calls in the `NativeLink` class and Wolfram Research's platform-dependent *MathLink* library. `NativeLink` is the class that implements the `MathLink` interface in terms of native methods. Currently, every time you call `MathLinkFactory.createMathLink()` or `MathLinkFactory.createKernelLink()`, an instance of the `NativeLink` class is created, so the *J/Link* native library must be loaded. In other words, the only thing in *J/Link* that needs the native library is the `NativeLink` class, but currently all `MathLink` or `KernelLink` objects use a `NativeLink` object. You cannot do anything with *J/Link* without requiring the native library to be loaded.

Different browsers have different requirements for allowing applets to load native libraries. In many cases, the applet must be "signed", and the browser must have certain settings enabled. Note that letting Java applets launch local kernels is an extreme breach of security, since *Mathematica* can read sensitive files, delete files, and so on. It is probably not a very good idea in general for users to allow applets to blast such an enormous hole in their browser's security

sandbox. A better choice is to have Java applets use a kernel residing on the server. In this scenario, the browser's Java runtime does not need to load any local native libraries, so there are no security issues to overcome. This requires significant support on both the client and server side. This support is not part of *J/Link* itself, but it is a good example of the sort of programs *J/Link* can be used to create.