Java is a powerful and enormously popular programming environment, and Mathematica’s built-in J/Link toolkit integrates Java and Mathematica very tightly. This article presents a quick-start introduction to creating Mathematica user interfaces using J/Link. By the end, you will have been introduced to all the major techniques and be ready to create your own interfaces.

Introduction

By some measures, Java is the most popular programming language in the world. It features a vast and ever-growing library of classes for just about anything you would want to do with a computer—everything from image and sound processing to database access to internet protocols to advanced user interfaces. Virtually every new computing technology nowadays shows up first in a Java-language implementation. Now imagine for a moment that all of these Java classes were written in Mathematica instead, so that all the power and functionality they provide was available directly to Mathematica programmers. Would that change the rules about what could be done in Mathematica?

Welcome to J/Link, a product that integrates Mathematica and Java so tightly that it is as if everything in Java was actually written in Mathematica. J/Link lets you load Java classes into Mathematica, create objects of those classes, and call methods on them. You can do this with any Java class, without any modification or preparation whatsoever. And you can do it all from the comfortable, integrated, line-at-a-time development environment provided by Mathematica.

Java is good for many things, and one of its more popular uses is for building platform-independent user interfaces. This makes it an ideal toolkit for a commonly sought capability—creating user interface elements for Mathematica programs. You might want something as simple as a progress bar or a dialog box with a few fields for entering input values, or something as complex as a wizard for leading users through the steps of a long computation. J/Link gives you the full power of Java to build these types of interfaces and more, using just Mathematica code, not Java.
This article presents a quick-start introduction to creating *Mathematica* user interfaces using *J/Link*. It can be read and worked through input by input in a single sitting. By the end, you will have been introduced to all the major techniques and you will be ready to create your own interfaces. *J/Link* includes an extensive User Guide that covers all the information in this article, and much more. You can find it in the Help Browser under the Add-ons category, and you should consult it for more detailed examples and explanations. There are also a number of example programs included with *J/Link* that demonstrate user interface techniques. These can be found in the *<Mathematica directory>/AddOns/JLink/Examples/Part1* directory, and they are discussed in Section 1.3 of the User Guide.

Our focus here is on user interfaces designed to be used in conjunction with the notebook front end. That is, we will be creating *Mathematica* programs that, when run from a typical *Mathematica* session, cause dialog boxes or other windows to “pop up” on the screen for the user to interact with. It is also possible to use *J/Link* to write Java programs that call on *Mathematica* in the background for computational services. That involves going in the other direction—calling the *Mathematica* kernel from Java code. Such programs are the subject of the second half of the *J/Link* User Guide, and we will not be dealing with them in this article.

### J/Link Basics

In this section we will examine the basic concepts of *J/Link* programming. My intent is to cover just enough for readers to become comfortable with the user interface examples developed later. I will make occasional references to specific sections in the *J/Link* User Guide. These references will be given as section numbers, and you can use these section numbers in the Go To field of the Help Browser. If a reference is to Section 1.2.1, for example, you can type “*J/Link 1.2.1*” (without the quotation marks) in the text field at the top and click the Go To button. You must make sure that the Add-ons button is depressed when you do this, as the User Guide is in the Add-ons section of the Help Browser. If you are reading this article as a notebook, then of course you can just click the links and be taken to the appropriate pages in the Help Browser automatically.
What is J/Link?

J/Link is a toolkit that integrates Mathematica and Java. It lets you call Java from Mathematica and Mathematica from Java. It sits on top of MathLink, Wolfram Research’s protocol for communicating with the Mathematica kernel. When you use J/Link to call Mathematica from a Java program, you can think of it as a MathLink Developer Kit for Java. When you use it to call Java from Mathematica, as in this article, it is just a Mathematica package that provides functionality for launching and controlling a Java runtime. You simply load the package and start writing programs that use it. The internals of J/Link are written in a combination of Mathematica, Java, and C. All the source code is provided for those who might be interested, and can be found in <Mathematica/AddOns/JLink/Source and <Mathematica directory>/AddOns/JLink/Kernel (where the Mathematica code resides).

Beginning with Mathematica 4.2, J/Link is a standard part of the Mathematica system, not an add-on that you or your users need to download and install (although this is necessary in Mathematica 4.0 and 4.1). A Java Runtime Environment is also installed with Mathematica 4.2, so J/Link is ready to use right out of the box. This means that you can write J/Link programs for others and know that they will work without any setup on the user’s part. They might be completely unaware that Java is even being used. Those with Mathematica 4.0 and 4.1 need to install J/Link and a Java Runtime Environment on their own, although many people will already have an appropriate Java system on their machines. J/Link includes installation instructions and information on obtaining a Java runtime for those with Mathematica 4.0 and 4.1. The current version of J/Link is 2.0, and that is the version assumed for this article. The home page for J/Link is www.wolfram.com/solutions/mathlink/jlink.

Starting the Java Runtime

The first step in using J/Link is to load the package.

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

Java programs execute in a managed environment called a Java Runtime Environment, or just a “Java runtime.” Before you can use J/Link from Mathematica, you must launch the Java runtime. All calls from Mathematica into Java are directed to this single Java runtime, which runs as a separate process from the Mathematica kernel. MathLink is used to communicate between the kernel and the Java runtime, but you do not have to be concerned with such details.

The function that launches Java is called `InstallJava`, named after the existing Mathematica function `Install`, which launches an external program and prepares it to be used via MathLink.
InstallJava[]

LinkObject[d:\math42\SystemFiles\Java\Windows\bin\javaw, 2, 2]

\(J/Link\) also has functions for shutting down the Java runtime (\texttt{UninstallJava}) and for restarting it (\texttt{ReinstallJava}). Do not use them. Developers need to remember that the Java runtime used by \(J/Link\) is a shared resource. Other \texttt{Mathematica} packages you have loaded might also be using \(J/Link\), and shutting down Java will cause problems for them. Start the Java runtime when you need it and then just leave it running.

If Java is already running when you call \texttt{InstallJava}, the function does nothing. This means that it is safe to call it whenever you want, such as at the start of any programs that need \(J/Link\).

\section*{Creating Objects and Calling Methods}

Now let us begin calling into Java. An obvious first step is to create a Java object. In Java, you use the \texttt{new} operator for this. In \texttt{Mathematica}, you use the \texttt{JavaNew} function. Here we create an object of the \texttt{Date} class.

\begin{verbatim}
  dateObject = JavaNew["java.util.Date"]

  «JavaObject[java.util.Date] »
\end{verbatim}

The first argument to \texttt{JavaNew} is the full name of the class with all the periods. Any arguments to the constructor would be given in a sequence after the class name, but the constructor being invoked here takes no arguments. It creates a \texttt{Date} object that represents the current time. Note that the return value is a funny expression surrounded by angle brackets. The angle brackets remind you that this is a special output form, not the true \texttt{Mathematica} representation of the expression. A \texttt{JavaObject} expression, as this is called, is a \texttt{Mathematica} representation of a Java object that lives in the Java runtime.

How did I know the name of the class and the arguments to the constructor I wanted to invoke? Obviously, I read the Java documentation. To use \(J/Link\) you are not going to have to learn the Java language, but you will have to learn what Java classes are available and what methods they contain. There are many books describing the standard Java class libraries, and for some uses you can get by with just the so-called “JavaDoc” help files. JavaDoc is a hyperlinked web page format for supplying help files for Java classes. A complete JavaDoc reference to the standard Java class libraries (including the user interface classes) can be found at java.sun.com/j2se/1.4/docs/api.

If you look at the documentation for the \texttt{java.util.Date} class, you will see that it has a method called \texttt{toString()} (all Java objects have this method). Here is how to call the \texttt{toString()} method.
If you are familiar with Java, you would know that to call this method you would write in Java:

```java
// Java code
dateObject.toString();
```

It looks almost the same in Mathematica except that we use the `@` symbol instead of the dot, and we use square brackets instead of parentheses. Throughout J/Link, we use the `@` like Java’s dot, to mean “call this method (or access this field) on the preceding object.” Below is a table taken from Section 1.1.6 of the User Guide that compares the Mathematica and Java syntaxes for all the different ways you use Java objects.

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Mathematica</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constructors</strong></td>
<td><code>MyClass obj = new MyClass(args);</code></td>
<td><code>obj = JavaNew[&quot;MyClass&quot;, args];</code></td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td><code>obj.methodName(args);</code></td>
<td><code>obj@methodName[args];</code></td>
</tr>
<tr>
<td><strong>Fields</strong></td>
<td><code>obj.fieldName = 1;</code></td>
<td><code>obj@fieldName = 1;</code></td>
</tr>
<tr>
<td></td>
<td><code>value = obj.fieldName;</code></td>
<td><code>value = obj@fieldName;</code></td>
</tr>
<tr>
<td><strong>Static methods</strong></td>
<td><code>MyClass.staticMethod(args);</code></td>
<td><code>MyClass</code>`staticMethod[args];`</td>
</tr>
<tr>
<td><strong>Static fields</strong></td>
<td><code>MyClass.staticField = 1;</code></td>
<td><code>MyClass</code>`staticField = 1;`</td>
</tr>
<tr>
<td></td>
<td><code>value = MyClass.staticField;</code></td>
<td><code>value = MyClass</code>`staticField;`</td>
</tr>
</tbody>
</table>

If this chart is hard to understand or the Mathematica syntax looks unfamiliar, do not worry. After just a few minutes of writing J/Link code yourself, it will become very natural.

We have already seen an example of how to call a method on an object (a so-called “instance” method). Classes can also have static methods and fields, which are associated with the class itself and not any
specific object of that class. For example, Java has a class called `Math` that contains a number of mathematical functions like `abs()`, `sqrt()`, `sin()`, and so on. You do not ever create an object of type `Math` and call methods on it. It is just a container for a collection of static methods that operate on numbers.

To call a static method, the first thing you must do is load the class into `Mathematica`. Any Java class you use with `J/Link` must be loaded into `Mathematica`, but this happens automatically for you when you call `JavaNew` to create the first object of that class. When calling a static method, however, you are not using an object of that class, so the class might not have been loaded yet. You have to do it manually using `LoadJavaClass`.

```
LoadJavaClass["java.lang.Math"]
JavaClass[java.lang.Math, 2]
```

When you call a static method you use the class name as a prefix, separated from the method name by a context mark.

```
Math`cos[N[Pi]]
```

You will note that static methods and fields are the exception to the rule that dots in Java become `@` in `Mathematica`. Although this sometimes causes confusion for novice `J/Link` programmers, it is actually quite natural. Java uses the dot in two very different ways—as the member access operator, as in `obj.method()`, and as the scope resolution operator, as in `package.Class.staticMethod()`. Some other object-oriented languages use different operators for these two meanings (C++ uses `::` as the scope resolution operator). Although it is not often described in this way, the context mark is `Mathematica`’s scope resolution operator. In other words, do not complain that `Mathematica` requires two different operators, complain that Java overloads the dot for two very different meanings.

`J/Link` maps Java types to their natural equivalents in `Mathematica`. For example Java integer types (`char`, `int`, `long`, and so on) map to `Mathematica` integers. Arrays in Java map to `Mathematica` lists of the appropriate depth. Java objects map to `JavaObject` expressions. When I say “map,” I mean that if a Java method takes an `int`, then you call it from `Mathematica` with an integer argument. If a Java method returns an `int`, you will get an integer in `Mathematica` as the return value. Section 1.1.4 of the User Guide has a table showing the complete set of type mappings. It should correspond with most people’s intuition about which types are most logically equivalent (for example, `boolean` maps to `True` or `False` in `Mathematica`, and the `null` object maps to `Mathematica`’s symbol `Null`).
Underscores in Java Names

Some Java names have underscores (_) in them. These characters are especially common in the names of constants, for example \texttt{TOP_ALIGNMENT}. This causes a problem for \texttt{J/Link} because the underscore is not a valid character in a \texttt{Mathematica} symbol name. \texttt{J/Link} solves this problem by mapping the underscore into a capital U in symbol names. The rule is that when you use a Java name in a \texttt{Mathematica} symbol, you convert the _ to a U, but when you use the name in a \texttt{Mathematica} string, you do no conversion (in other words, you convert only when necessary). For example, if you have a class named \texttt{My\_Class} with a field named \texttt{SOME\_CONSTANT}, you would get the value of that field like this.

\begin{verbatim}
val = MyUClass`SOMEUCONSTANT
\end{verbatim}

But when you refer to the class name in a string, you keep it unmodified.

\begin{verbatim}
LoadJavaClass["My\_Class"]
\end{verbatim}

Cleaning up Unused Objects

\texttt{J/Link} keeps track of which Java objects have been sent to \texttt{Mathematica} so that it can prevent the Java garbage collector from trying to free them while they are still in use by \texttt{Mathematica}. Unless you explicitly tell \texttt{J/Link} that you are finished with a Java object in \texttt{Mathematica}, it will never allow the object to be freed and objects will pile up in your Java session. This is generally not a problem if you are just tinkering with \texttt{J/Link}, but, in code that you write for others to use, you should make sure that you clean up any Java objects that are no longer needed.

You use the function \texttt{ReleaseJavaObject} to tell \texttt{J/Link} that you no longer wish to use a Java object in \texttt{Mathematica}. Here we release the \texttt{Date} object we created earlier.

\begin{verbatim}
ReleaseJavaObject[dateObject]
\end{verbatim}

It is an error to refer to an object in \texttt{Mathematica} after it has been released.

In actual practice, \texttt{ReleaseJavaObject} is not used very often because it is too much effort to keep track of every object sent to \texttt{Mathematica}. What you usually want is a way to create a block of code in which all newly-created Java objects will be marked as temporary and automatically released when the block finishes. This is especially the case with user interface programs—your program will create a window with some controls in it and display it to the user, but, when the user closes the window, the program is done and does not need to leave anything behind. In the same way that any local variables that were used get cleaned up, you want any Java objects sent to \texttt{Mathematica} to be released. The function that accomplishes this is \texttt{JavaBlock}.

Many \texttt{J/Link} programs will have the following structure.
MyFunc[\_\_\_\_] :=
JavaBlock[
    Module[{obj, result, otherLocals},
        obj = JavaNew["SomeClass"];  
        ... work with obj, perhaps create more objects ...
        result
    ]
]

You can be guaranteed that \textbf{MyFunc} will not “leak” any objects, meaning that any objects sent to Mathematica for the first time during the execution of \textbf{MyFunc} will be released when it finishes. Note that only objects sent to Mathematica for the first time will be released. 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.

Remember that Java objects are sent to Mathematica not just when you call \textbf{JavaNew}, but also when you call a Java method that returns an object. You might not even have assigned the object to a variable, such as when you chain method calls.

\texttt{obj@methodThatReturnsObject[]@otherMethod[]}

In this example, \texttt{methodThatReturnsObject()} will cause a \texttt{JavaObject} expression to be created in Mathematica but not be assigned to any variable. This is why you should get in the habit of wrapping \texttt{JavaBlock} around code that does not need to create long-lived Java objects in Mathematica. You cannot tell whether a program creates objects in Mathematica simply by looking for calls to \texttt{JavaNew}.

There is one useful exception to the rule that all novel objects encountered during a \texttt{JavaBlock} are released when it ends. If a \texttt{JavaBlock} returns a single object, that object will not be released. This allows another common programming pattern in which you create several objects and operate on them, but you need just one of them to live beyond the boundary of the \texttt{JavaBlock}. We will see examples of this later, such as when you create a Frame object and populate it with controls (which are themselves objects), but you need the Frame object itself to persist after the function that created it finishes.

\section*{Getting Information about Classes}

\texttt{J/Link} provides a few utility functions that let you see what constructors, methods, and fields a class includes. Although this is no substitute for documentation on the classes you are using, it can be very useful as a quick help system. Here we inspect the constructors for the \texttt{java.awt.Button} class.

\begin{center}

<table>
<thead>
<tr>
<th>Constructors[&quot;java.awt.Button&quot;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button() throws java.awt.HeadlessException</td>
</tr>
<tr>
<td>Button(String) throws java.awt.HeadlessException</td>
</tr>
</tbody>
</table>

\end{center}
Here are the fields.

```
Fields["java.awt.Button"]
```

```
static final int ABORT
static final int ALLBITS
static final float BOTTOM_ALIGNMENT
static final float CENTER_ALIGNMENT
static final int ERROR
static final int FRAMEBITS
static final int HEIGHT
static final float LEFT_ALIGNMENT
static final int PROPERTIES
static final float RIGHT_ALIGNMENT
static final int SOMEBITS
static final float TOP_ALIGNMENT
static final int WIDTH
```

There are a lot of methods in the Button class, the vast majority of which are inherited from parent classes (especially the very large class java.awt.Component). If you are only interested in the methods unique to the Button class, you can use the option Inherited->False.

```
Methods["java.awt.Button", Inherited -> False]
```

```
void addActionListener(java.awt.event.ActionListener)
void addNotify()
javax.accessibility.AccessibleContext getAccessibleContext()
String getActionCommand()
java.awt.event.ActionListener[] getActionListeners()
String getLabel()
java.util.EventListener[] getListeners(Class)
void removeActionListener(java.awt.event.ActionListener)
void setActionCommand(String)
void setLabel(String)
```

You can also use string patterns to restrict the members you want to see. Here we specify that we only want to see the “set” methods (because we have left out the Inherited->False option, we see many methods defined in parent classes).
Methods["java.awt.Button", "set*"]

void setActionCommand(String)
void setBackground(java.awt.Color)
void setBounds(int, int, int, int)
void setBounds(java.awt.Rectangle)
void setComponentOrientation(java.awt.ComponentOrientation)
void setCursor(java.awt.Cursor)
void setDropTarget(java.awt.dnd.DropTarget)
void setEnabled(boolean)
void setFocusable(boolean)
void setFocusTraversalKeysEnabled(boolean)
void setFocusTraversalKeys(int, java.util.Set)
void setFont(java.awt.Font)
void setForeground(java.awt.Color)
void setIgnoreRepaint(boolean)
void setLabel(String)
void setLocale(java.util.Locale)
void setLocation(int, int)
void setLocation(java.awt.Point)
void setName(String)
void setSize(int, int)
void setSize(java.awt.Dimension)
void setVisible(boolean)

The examples above used the class name as the first argument, but you can also use an object of the class.

button = JavaNew["java.awt.Button"];  
Fields[button, "*_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
User Interfaces with \textit{J/Link}

After these basics of \textit{J/Link}, we are ready to begin writing user interfaces. As we have seen, when we write \textit{J/Link} programs we are writing the same sort of code that we would write in Java, except using \textit{Mathematica} syntax. This means that the user interface programs we write will look almost identical to programs of similar functionality written in Java. This is both a great strength of \textit{J/Link} and a drawback. As a drawback, it means that there is no higher-level “\textit{Mathematica}-like” layer on top of the raw Java calls. Such a layer would attempt to make it simpler to create interfaces and reduce the amount of code that needed to be written. But the fact that the only programming functions you need to learn are the Java user interface classes has some important advantages. First, if you already have experience using the Java classes, that knowledge carries over directly. If you do not know the Java classes, then the knowledge you gain from using them with \textit{J/Link} will have advantages in the future if you ever do Java programming. In other words, you do not have to invest time learning a special \textit{Mathematica}-only user interface toolkit. Another important benefit is that you can take advantage of the huge number of existing books on Java programming and the large amount of sample code out there. Finally, you have exactly the amount of power that Java’s libraries give you—no less. If you use a simplified toolkit, then you risk running into limitations of the toolkit if you have special needs or want to do something very fancy. In the future, Wolfram Research may very well make available another way of creating user interfaces that sits on top of \textit{J/Link} and hides some of the raw Java calls (but still allows power users to get down to the metal if necessary). For now, though, writing user interface programs in \textit{Mathematica} is essentially just like writing them in Java.

As mentioned earlier, an obvious consequence of the fact that you will be making calls to the Java user interface class libraries is that you will want documentation for these libraries. You can go a long way just by copying fragments out of the \textit{J/Link} example programs below and the ones included with \textit{J/Link}, but there is no substitute for a reference book. Good reference materials are online at Sun’s Java pages. The JavaDocs for all the standard Java classes can be found at java.sun.com/j2se/1.4/docs/api, and there are good tutorials on Java user interfaces at java.sun.com/docs/books/tutorial/ui, if you want to use the older but simpler Abstract Window Toolkit (AWT) classes, and at java.sun.com/docs/books/tutorial/uiswing/index.html, if you want to use the newer and more powerful Swing library.

As just mentioned, there are two user interface class libraries that are a standard part of Java. The original one is called AWT, and the newer one is called Swing. The AWT is easy to use but suffers from some limitations, most notably that it can be hard to get user interfaces to look very polished on every platform. It also is not suitable for very complicated interfaces, but that would not be a limitation for most of the interfaces \textit{Mathematica} programmers are likely to create. In response to perceived problems with AWT, Sun created a new user interface library called Swing. Swing is a little more complicated to use but has more features and gives you a better chance of having your interface look good on every platform. We will use Swing for the examples in this article, primarily because that is what most programmers are using.
nowadays. The examples included with J/Link use mostly the older AWT because when they were created, some platforms did not have Java runtimes modern enough to run Swing.

### Your First Window

Let us get started right away with our first window. The Swing class that represents a top-level window is called JFrame. Instead of using JFrame, we will use a subclass of it that is included in J/Link, called MathJFrame. MathJFrame adds two features to a standard JFrame that make it useful for J/Link programmers: the window goes away when its Close box is clicked, and it has special support for so-called “modal” J/Link interfaces, which will be discussed later.

```mathematica
myWindow = JavaNew["com.wolfram.jlink.MathJFrame"]
<<JavaObject[com.wolfram.jlink.MathJFrame] »
```

You may notice a slight delay the first time a class is loaded into Mathematica, especially if it (or one of its parent classes) has many methods, as is the case with the JFrame class. We have now created a window, but it is not visible. Before we make it visible, let us make it a reasonable size and place it somewhere other than the default location at the top left corner of the screen.

```mathematica
myWindow@setSize[200, 200]
myWindow@setLocation[300, 300]
```

The JFrame class has methods called setVisible() and toFront() that sound like they would be useful for making the window appear on top of all other windows. Unfortunately, these methods are not enough to force the window to the foreground on all platforms, so J/Link includes a special function called JavaShow that does this for you.

```mathematica
JavaShow[myWindow]
```

Here is what the window looks like.
Congratulations, you have now created your first *Mathematica* user interface! For a little fun, play around with resizing and repositioning it with *Mathematica*.

\[
\text{myWindow@setSize[400, 400]}
\]

You have just experienced one of the incredible advantages of *J/Link* programming versus writing programs in Java. In *Mathematica* you can build your interface a line at a time, *while it is running*. There is no need to write a complete program, compile it, and run it just to see what it does. This ability to interactively experiment with Java classes and methods makes *J/Link* an ideal tool for learning about a class library.

**Adding a Component to the Window**

An empty window is not the most useful interface, so let us add something to it: a button that changes the background color of the window every time it is clicked. The Swing class for buttons is *JButton*, and one of the constructors lets us specify the text in the button.

\[
\text{myButton = JavaNew["javax.swing.JButton", "Click Me!"]} \\
\quad \text{«JavaObject[javax.swing.JButton]} \\
\]

When you add components to a Swing *JFrame*, you do not add them directly to the *JFrame*—instead, you add them to a special “pane” called the content pane. The Swing documentation goes into detail about the motivation for having the content pane, but you do not really need to care. Just remember to call `getContentPane()`.

\[
\text{myWindow@getContentPane[]@add[myButton]} \\
\quad \text{«JavaObject[javax.swing.JButton]} \\
\]
Components that can contain other components support a special class called a layout manager, which can supply complex logic for controlling the sizes and positions of child components. We will use a layout manager in a later example, but for now we will specify the button’s size and position explicitly. To do this we need to get rid of the content pane’s default layout manager.

```java
myWindow.getContentPane().setLayout(null)
```

Next we position the button.

```java
myButton.setSize(100, 30)
myButton.setLocation(50, 70)
```

## Handling Events

Now that we have the window and its button, we need to add some behavior. We want a click in the button to cause the background color of the window to change to a random color generated by Mathematica. Java user interface components fire events in response to user actions, and other components indicate their interest in these events by registering as event listeners. In our example, the button will fire an event when it is clicked, and something else will listen for the event and change the content pane’s color in response. The “something else” that listens for the button-click event is not the content pane itself. To put that logic into the content pane would require creating a subclass of the pane’s class. It would be very cumbersome to have to create a new subclass of each component class just to allow it to respond to events in a certain way. 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. The only specialty code goes into the adapter class, allowing the components that fire and respond to events to be generic.

Swing buttons fire an `ActionEvent` when clicked, and the corresponding event-listener interface is `ActionListener`. In our example, a Java programmer would create a tiny new class that implements the `ActionListener` interface. This interface has only one method, `actionPerformed()`, and in the body of the `actionPerformed()` method the programmer would set the content pane’s color. The programmer would wire up this event behavior by calling the button’s `addActionListener()` method, supplying an instance of this new class. When the button is clicked, it notifies all its registered listeners by calling their `actionPerformed()` methods.

We see from this discussion that Java programmers typically need to create a new listener class for every behavior they want to add to their program. Mathematica programmers using J/Link cannot write new Java classes, so how do we add behaviors to our programs? The answer is that J/Link comes with ready-made listener classes for the user interface events typically encountered in Java programs. These classes are referred to generically as “MathListener” classes because they listen for events and respond by calling user-specified code in Mathematica. To add a behavior to your program, you simply create an instance of
the appropriate MathListener class and tell it what Mathematica function you want executed for each of the events that it is capable of responding to.

This is all very straightforward when you see it in action. Let us add the click behavior to our button. First we create an instance of the appropriate MathListener class, which for a button is MathActionListener.

```java
actionListener = JavaNew["com.wolfram.jlink.MathActionListener"]
```

Now we assign the Mathematica function we want called. To do this, we use the setHandler() method, which takes the name of the event-listener method (“actionPerformed”) and the name (as a string) of a Mathematica function to be called to implement the actionPerformed() method. Our Mathematica function will be called ButtonClickHandler.

```java
actionListener@setHandler["actionPerformed", "ButtonClickHandler"]
```

Now we need to register our actionListener with the button. This is done by calling the button’s addActionListener() method.

```java
myButton@addActionListener[actionListener]
```

All that remains is to write the ButtonClickHandler function. To see what arguments will be passed to your Mathematica handler function, consult the table in Section 1.2.7.4 of the User Guide. Your function will always be passed the event object itself, and possibly other information that might be useful (this extra information can always be obtained from the event object—it is provided separately simply as a convenience). In this example, we are not interested in any of the arguments.

```java
ButtonClickHandler[___] :=
JavaBlock[
    myWindow@getContentPane[]@setBackground[
        JavaNew["java.awt.Color", Random[], Random[], Random[]]
    ]
]
```

Note that we wrapped the body of the function in JavaBlock to release the Color object that we created. We used the Color constructor that takes RGB values as real numbers in the range 0 to 1.

It has only taken a few lines of Mathematica code to wire up a simple behavior for the button. At this point you might be wondering how to know what events a component fires and what MathListener to use. Events
and event listeners are well documented in Java, but in practice you can use a very quick shortcut. Simply look in the JavaDocs for the component you are interested in (JButton, JPanel, JComboBox, JComponent, and so on) for one or more methods with names of the form addXXXListener. Do not forget to look in the methods inherited from parent classes as well. Once you find such a method, you know that the component fires the events listed in the XXXListener interface. The corresponding MathListener will be named MathXXXListener.

In some cases you will find that there is no MathListener class provided for the event-listener interface that you need to use. For example, at the time of this writing there is no MathMenuListener in J/Link for handling events from JMenu components. If there is no existing MathListener, you can use the Mathematica function ImplementJavaInterface to create an appropriate class “on the fly.” We will see an example of using ImplementJavaInterface later, and Section 1.2.17 of the User Guide describes ImplementJavaInterface in detail and gives an example of using it to handle events from a JMenu.

Did you click the button to see if it changed the color of the window? If you have not done this yet, do not, because it will not work! One crucial step remains, which is the subject of the next section.

Modal and Modeless Interfaces

Consider what will happen if you click on the button in your running example program. Java will detect the click, and the button will notify the MathActionListener by calling its actionPerformed() method. The MathActionListener will send commands to Mathematica to execute the ButtonClickHandler function. These commands will arrive on the MathLink that connects the Mathematica kernel and Java. What will Mathematica do when these commands arrive on the link? Absolutely nothing! That is because the Mathematica kernel is not paying any attention to the link to Java. The kernel spends its life waiting for commands to arrive on the MathLink that connects to the front end. Before any Java user interface program can interact with Mathematica, Mathematica needs to be put into a state where it is receptive to commands arriving from Java.

There are two main ways for making Mathematica receptive to commands initiated in Java (a third, less commonly used method is described in Section 1.2.7.11 of the User Guide). The method you choose will depend on the requirements of your user interface program. Consider a Java dialog box that is used to control some initial values for a computation. The dialog box is created and displayed; the user enters some values and then dismisses it by clicking the OK button. The entire lifetime of the dialog box falls within the bounds of a single computation in Mathematica. In contrast, consider a Java interface that is intended to remain visible and active for a long stretch of time in a user’s session. This could be some kind of tool or utility that the user would use occasionally, similar to a palette in the front end.

The first type of interface, which is created and destroyed within the span of a single computation, is called a “modal” interface. The term “modal” is used here with a slightly different meaning than the usual one. In standard computing terminology, a modal window is one that must be dismissed before anything else can
be done in the application. A typical example is a Save File dialog box. In J/Link, what we call a modal interface is not necessarily modal with respect to other Java windows—it is modal with respect to the Mathematica kernel. That is, the kernel is kept busy dealing with commands arriving from Java until the Java window is dismissed.

The second type of interface, which remains active for longer than a single kernel computation, is called “modeless.” It is modeless with respect to the Mathematica kernel, meaning that the kernel is not kept busy servicing commands from Java. The user is free to do computations using the front end while the Java interface is still active.

Modal interfaces are more common than modeless ones, and they are conceptually simpler. They also offer better performance if the Java program needs to call Mathematica many times in quick succession, such as when a slider is being dragged. Modal interfaces are run using the DoModal function, which does not return until the modal interface is dismissed. Modeless interfaces are enabled with the ShareKernel function, which puts the kernel into a state where it is receptive to commands arriving from either the front end or Java.

Let us now run our button example program in a modal way. Before we do this, go back and click the Click Me! button if you have not done so already. From the discussion in the previous few paragraphs we know it will not work yet, but it is interesting to see exactly what happens. Not only does the color of the window not change, but the Java user interface thread hangs. You can see this by clicking the button again and noting that it does not even give any visual feedback of being pressed. In fact, if you bring another window in front of the Java window and then click the Java window to bring it back in front, it will not repaint itself, a sure sign that the Java user interface thread is hanging. This makes sense because the thread is waiting for Mathematica to respond to its attempt to call the ButtonClickHandler function.

The user interface thread will hang until we tell Mathematica to start paying attention to commands arriving from Java, which we can do by calling DoModal[] (or ShareKernel[] if we want a modeless interface). The DoModal function will not return until the Java side calls EndModal[]. There are many ways to arrange for this to happen, but by far the easiest and most common way is to use a special facility of the JFrame class. If you call a JFrame’s setModal() method, it will call EndModal[] when it is closed, thus causing DoModal to return.

myWindow.setModal[]

Now we are finally ready to run the button example as an interactive program.

DoModal[]

The call to DoModal will not return until you close the Java window. Click the Click Me! button a few times and see how the window color changes. Here is what it looks like in action.
The Complete ButtonClick Program

We have now completed a trivial J/Link user interface program. Note that it only took a few lines of code, and all of it is written in Mathematica. The code is almost line-for-line comparable to a pure Java version of the same program. Now let us collect all the code together and package it as a simple program called ButtonClick.
The ButtonClick function is a very typical J/Link program. The whole function is wrapped in JavaBlock, as we have no need for any of the Java objects created to persist in Mathematica after the function ends. Another common idiom is to call InstallJava at the start, in case Java is not already running. One trick to note is in the line that calls setHandler(). We want the buttonClickHandler function to be local to the Module, and this means that its true name is something like buttonClickHandler$123. We have to pass the name of this function as a string in setHandler(), and we can be sure to capture its true name by calling ToString[buttonClickHandler], instead of incorrectly hard-coding "buttonClickHandler".
A More Advanced Example

We have now worked our way through a trivial J/Link user interface program. Now let us look at something considerably more advanced, and examine some new techniques along the way. In particular, we will see the following.

- How to display Mathematica graphics in a Java window.
- How to use ShareKernel to run an interface modelessly.
- How to use ImplementJavaInterface to allow event callbacks to Mathematica when there is no appropriate existing MathListener class.

The program we will create allows the user to experiment with Taylor series approximations to functions. They can enter a function and a range over which to plot the function and its Taylor series approximation. They then use a slider to modify the number of terms in the series expansion. The series expansion is displayed along with a plot showing the function and the Taylor series approximation. The plot and series expansion are updated on the fly as the slider is dragged. Here is a screen shot of the program in action.

The general flow of the Taylor program is as follows.
Laying out the Window

As a first step in creating the Taylor program, we will write a preliminary version, Taylor1, that only builds the interface and displays it. Behavior will be added in subsequent revisions. The code to create the window and all its components is somewhat lengthy. One reason for this is that we will do the work necessary to ensure that the components in the window lay out nicely on all platforms and reposition correctly if the window is resized. In the ButtonClick example, we specified the button’s size and location with explicit numerical values. This works fine for simple windows that cannot be resized and have just a few components, or if you are not concerned about platforms beyond the one on which you are developing the program. But if you hardcode component positions and sizes for a more complex window, it is practically guaranteed that on some platforms (perhaps Mac OS X, or Linux with some unusual window manager and fonts) there will be layout problems. Typical examples are buttons that are too small to hold all their text, labels that are cut off by other components, and text fields that are the wrong height for a line of text.

The best way to avoid layout problems like these is to use a layout manager, which is a special Java class that knows how to lay out child components within a container component. Proper use of a layout manager helps ensure that components have their “natural” sizes on each platform and reposition sensibly if their container is resized. Using layout managers can be a tricky part of Java programming, but it is worth the effort if you want to create high-quality interfaces that look good on every machine. In the Taylor program, we will use a BoxLayout, which is a new layout manager provided in Java 1.3. BoxLayout is useful when you have several rows with different numbers of components in each. A useful tutorial on layout managers can be found at java.sun.com/docs/books/tutorial/uiswing/mini/layout.html.

The section of the program that creates the window and lays out the components is too long to try to work through and explain. Frankly, this part of any program is somewhat tedious and uninteresting, and it is beyond the scope of this article to try to teach readers how to write this type of code. The Mathematica code you write using J/Link is essentially identical to the code a Java programmer would write, and for guidance you can refer to any of the copious books, articles, and other resources on user interface programming in Java.

The code in Taylor1 is presented as a complete function, but of course this is not how it was created. Naturally, it was built up bit by bit with much experimentation. This interactive programming style is the huge advantage of using J/Link versus programming directly in Java. It is especially valuable when trying
to get the layout of components just right. In fact, I had never used a BoxLayout and its helper Box class before, and being able to play with them a line at a time was extremely useful for learning.

Two things that are temporarily left out of **Taylor1** are a JavaBlock wrapping the entire function and local variables specified in the `Module`. It is very important that these are left out because this **Taylor1** program is not self-contained. What we want for now is a program that simply displays the window with its final appearance. We will then interactively add behavior via event callbacks to *Mathematica*. To do this we need access to the various Java objects we have created (the frame, the plot panel, the text fields, and so on), hence they cannot yet be made local to the `Module` or released by a JavaBlock.

```plaintext
Taylor1[] := Module[{}, (* No local declarations yet. *)
  (*/ Create the window and position it. */
   window = JavaNew["com.wolfram.jlink.MathJFrame"];
   window@setLocation[50, 50];
   window@setTitle["Taylor"];

   (*/ Create the two panels we will use. topPanel holds all except
      the plot. */
    topPanel = JavaNew["javax.swing.JPanel"];
    topPanel@setLayout[JavaNew["javax.swing.BoxLayout", topPanel,
      BoxLayout`YUAXIS]];
    LoadJavaClass["java.awt.Color"];
    plotPanel@setBackground[Color`white];
    plotPanel@setPreferredSize[JavaNew["java.awt.Dimension", 500, 500]];

   (*/ Add the two panels to the top and bottom of the window. */
     LoadJavaClass["java.awt.BorderLayout"];
     window@getContentPane[]@add[topPanel,
      ReturnAsJavaObject[BorderLayout`NORTH]];
     window@getContentPane[]@add[plotPanel,
      ReturnAsJavaObject[BorderLayout`SOUTH]];

   (*/ Now create and configure the various components. There are two
      main rows: row1 has all the text fields and Compute button, and row2
      has the slider. Use Box, Border and Glue components to ensure a nice
      layout on all platforms and at varying window sizes. */
    row1 = Box`createHorizontalBox[];
    row2 = Box`createHorizontalBox[];
    functionField = JavaNew["javax.swing.JTextField", "Sin[x]", 10];
    functionField@setMaximumSize[JavaNew["java.awt.Dimension", 20, 20]];
    aboutPointField = JavaNew["javax.swing.JTextField", "0", 2];
    aboutPointField@setMaximumSize[JavaNew["java.awt.Dimension", 8, 20]];)
```
aboutPointLabel = JavaNew["javax.swing.JLabel", " About point: "];
functionLabel = JavaNew["javax.swing.JLabel", "Function of x: "];
rangelabel = JavaNew["javax.swing.JLabel", " Range: "];
rangeField = JavaNew["javax.swing.JTextField", "3", 3];
rangelField@setMaximumSize[JavaNew["java.awt.Dimension", 8, 20]];
computeButton = JavaNew["javax.swing.JButton", "Compute"];
row1@add[Box`createHorizontalGlue[]];
row1@add[functionLabel]; row1@add[functionField];
row1@add[Box`createHorizontalGlue[]];
row1@add[aboutPointLabel]; row1@add[aboutPointField];
row1@add[Box`createHorizontalGlue[]];
row1@add[rangeLabel]; row1@add[rangeField];
row1@add[Box`createHorizontalGlue[]];
title = JavaNew["javax.swing.JLabel", "Taylor Series Explorer"];
title@setFont[JavaNew["java.awt.Font", "Dialog", BitOr[Font`BOLD, Font`ITALIC], 24]];
title@setAlignmentX[Component`CENTERUALIGNMENT];
LoadJavaClass["javax.swing.BorderFactory"];
title@setBorder[BorderFactory`createEmptyBorder[20, 0, 20, 0]];
slider = JavaNew["javax.swing.JSlider", 1, 10, 1];
slider@setSnapToTicks[True]; slider@setPaintLabels[True];
slider@setPaintTicks[True]; slider@setMajorTickSpacing[1];
slider@setMaximumSize[JavaNew["java.awt.Dimension", 180, 50]];
slider@setBorder[BorderFactory`createEmptyBorder[20, 0, 20, 0]];
row2@add[JavaNew["javax.swing.JLabel", " Number of terms: "]];
row2@add[slider];
seriesLabel = JavaNew["javax.swing.JLabel", "Series expansion"];
seriesLabel@setAlignmentX[Component`CENTERUALIGNMENT];
seriesLabel@setBorder[BorderFactory`createEmptyBorder[10, 0, 4, 0]];
seriesTextArea = JavaNew["javax.swing JTextArea"];
seriesTextArea@setRows[4];
seriesTextArea@setFont[
JavaNew["java.awt.Font", "Monospaced", Font`PLAIN, 12]];
plotLabel = JavaNew["javax.swing.JLabel", "Plot"];
plotLabel@setAlignmentX[Component`CENTERUALIGNMENT];
plotLabel@setBorder[BorderFactory`createEmptyBorder[10, 0, 4, 0]];

(* Add everything to the top panel. *)
topPanel@add[title]; topPanel@add[row1]; topPanel@add[row2];
topPanel@add[seriesLabel]; topPanel@add[seriesTextArea];
topPanel@add[plotLabel];
window@pack[];
JavaShow[window]
Yikes, that is a lot of code, but for the most part it is straightforward: create the components, set some properties, and put them where they belong. Scattered throughout are calls to manage properties of the layout, such as borders and spacing. This will pay off in allowing the window to be resized and still keep a nice appearance. One \textit{J/Link} programming issue is worth noting in this code—the line that reads:

\begin{verbatim}
window@getContentPane[]@add[topPanel, ReturnAsJavaObject[BorderLayout`NORTH]];
\end{verbatim}

You might be wondering why this is not the more obvious code:

\begin{verbatim}
window@getContentPane[]@add[topPanel, BorderLayout`NORTH];
\end{verbatim}

The answer is that the \texttt{add()} method takes an \texttt{Object}, not a \texttt{String}, as its second argument. The constant \texttt{BorderLayout.NORTH} is a Java string, and so it arrives in \texttt{Mathematica} as a literal string. The problem is that \textit{J/Link} does not allow you to pass a \texttt{Mathematica} string to an argument slot that is typed to take a Java object. What you need to pass as the second argument to \texttt{add()} is a \texttt{JavaObject} that is a reference to a Java string with the appropriate value. An easy way to do this is to use the \texttt{ReturnAsJavaObject} function to ask \textit{J/Link} to return \texttt{BorderLayout.NORTH} to \texttt{Mathematica} as a \texttt{JavaObject} instead of its usual behavior of converting it into a \texttt{Mathematica} string.

\subsection*{Adding Behavior}

Now that we have a function that builds the interface, let us start it running so that we can experiment with adding behavior.

\begin{verbatim}
Taylor1[]
\end{verbatim}

The \texttt{Taylor1} program shows the window and then returns, leaving the window on the screen. Now we want to add behavior—specifically, we want the two computed components (the series expansion text field and the plot panel) to recompute when either the slider is dragged or the \texttt{Compute} button is clicked. As we have seen, we will do this by registering event listeners that call back to \texttt{Mathematica} when the user manipulates either of these two controls.

The first thing we need is a function that extracts the necessary values from the input fields in the window, computes the necessary results, and updates the series expansion text box and the plot panel. We will call this function \texttt{computeTaylor}, and here is what it looks like.
We have made `computeTaylor` accept (and ignore) any sequence of arguments because it will eventually be called from Java in two different circumstances with different arguments, none of which are we interested in. The first four lines of `computeTaylor` extract the values from the input fields and slider. The function then computes the series expansion, puts the `OutputForm` of this result in the series expansion text area, and creates the plot. Because we are only interested in the `Graphics` object returned by `Plot`, we use a `Block` to control the value of `$DisplayFunction` so that no PostScript output is created.

The last line of `computeTaylor` causes the plot to appear in the plot panel. This panel is an instance of a special class provided with `J/Link` called `MathGraphicsJPanel`, which is a subclass of the Swing `JPanel` class with some special capabilities for displaying `Mathematica` graphics and typeset expressions. The `MathGraphicsJPanel` class and its counterpart for AWT-style programs, `MathCanvas`, are discussed in more detail in Section 1.2.8 of the User Guide. For many programs, using either of these classes is as simple as calling the `setMathCommand()` method, which takes a string giving a `Mathematica` expression that evaluates to a graphics expression. The panel evaluates the expression and displays the resulting image, automatically centered and scaled to fit the panel’s current dimensions.

We can now call `computeTaylor` and see its effects in the window. Make sure the input fields in the window have reasonable values and then execute this line.

```
computeTaylor[]
```

The only thing left is to wire up the button and slider so that they call the `computeTaylor` function. Before we can allow calls to `Mathematica` from the Java user interface thread, we need to ensure that `Mathematica` is in a state where it is receptive to calls originating in Java. In the `ButtonClick` example, we achieved this by using `DoModal` to put the kernel into a loop where it was continuously reading from
the link to Java. We will use `DoModal` again with the `Taylor` program, but only when we have fully packaged it in its final form. It is inconvenient to use `DoModal` during development because we want to be able to tinker with the program while it is running. The `DoModal` function does not return until the window is closed, so we would need to set up all the behavior in advance. You can use the `ShareKernel` function to put the kernel into a state where it is receptive to calls from either the front end or Java. This is convenient during development even for programs, like `Taylor`, that will be run modally when deployed.

This starts sharing.

```plaintext
sharingToken = ShareKernel[];
```

We save the result from `ShareKernel` to pass to `UnshareKernel` later on, when we turn off the sharing state.

When a `JButton` is clicked, it notifies all registered `ActionListeners` by calling their `actionPerformed()` methods. Therefore the MathListener we need to use is `MathActionListener`.

```plaintext
buttonListener = JavaNew["com.wolfram.jlink.MathActionListener"];
buttonListener@setHandler["actionPerformed", "computeTaylor"];
computeButton@addActionListener[buttonListener];
```

The `computeTaylor` function will now be called when the button is clicked. It will be passed some arguments when it is called, but we are not interested in any of them. Try altering the input fields in the window and clicking the `Compute` button to see the program in action.

The `Compute` button is intended for when the user has edited the text fields and wants to recompute. We also want the slider to be “live,” so that dragging it automatically recomputes. To see what events a `JSlider` fires as it is dragged, we examine the JavaDocs for `JSlider` looking for methods of the form `addXXXListener()`. We find `addChangeListener()` and see that we need a class that implements the `javax.swing.event.ChangeListener` interface. Unfortunately, `J/Link` does not include a `MathChangeListener` class, so what do we do?

`J/Link` provides a function called `ImplementJavaInterface` that allows you to create a new Java class “on the fly” that implements any interface by calling into `Mathematica`. When you call `ImplementJavaInterface`, you specify a list of rules that associates each method in the interface with a `Mathematica` function to be called that provides the implementation of that method. The most common use of `ImplementJavaInterface` is to create MathListener-type classes for interfaces that `J/Link` does not have built-in MathListener classes for. Below we create the listener we need, specifying that the `stateChanged()` method of the `ChangeListener` interface should be implemented by calling the `computeTaylor` function.
sliderListener = 
ImplementJavaInterface["javax.swing.event.ChangeListener",
 {"stateChanged" → "computeTaylor"}];

Now register our new listener with the slider.

slider@addChangeListener[sliderListener]

Play around with the slider and other fields to see the interface in action. When you are finished with the Taylor window, close it by clicking in its Close box. We have now interactively created the code we need for the final version of Taylor, and are satisfied that it looks and works as desired. We were able to create the code in stages, interactively modifying the window and its behavior while it was running.

The final version of the Taylor program will be run modally, and because we are done with the interactive development phase, we call UnshareKernel to turn off kernel sharing. We pass the value that was returned from ShareKernel as the argument to UnshareKernel.

UnshareKernel[sharingToken];

- The Completed Program

Here is the complete code for the final Taylor program. We essentially just collect all the code we have created so far into a single function.

Taylor[f_:Sin[x], aboutPoint_:0, range_:3, nTerms_Integer:3] :=
JavaBlock[
Module[{window, topPanel, plotPanel, row1, row2, functionField,
    aboutPointField, aboutPointLabel, functionLabel,
    rangeLabel, rangeField, computeButton, title, slider,
    seriesLabel, seriesTextArea, plotLabel, computeTaylor,
    buttonListener, sliderListener},
    InstallJava[];

    (* Create the window and position it. *)
    window = JavaNew["com.wolfram.jlink.MathJFrame"];
    window@setLocation[50, 50];
    window@setTitle["Taylor"];  

    (* Create the two panels we will use. topPanel holds all except
the plot. *)
    topPanel = JavaNew["javax.swing.JPanel"]; 
    topPanel@setLayout[JavaNew["javax.swing.BoxLayout", topPanel,
BoxLayout:YUAXIS]]; 

    (* Create the function field.
    *)
    functionField = JavaNew["javax.swing.JTextField"]; 
    functionField@addListener[buttonListener]; 
    functionField@setText["Sin[x]"];

    (* Create the slider.
    *)
    slider = JavaNew["javax.swing.JSlider"]; 
    slider@setMin[0];
    slider@setMax[3];
    slider@setValue[0];
    slider@setMajorTickSpacing[1];
    slider@setMinorTickSpacing[0.5];
    slider@setPaintLabels[true];
    slider@addChangeListener[buttonListener];
    slider@setPaintTrack[true];
    slider@setContinuous[false];

    (* Create the about point field.
    *)
    aboutPointField = JavaNew["javax.swing.JTextField"]; 
    aboutPointField@addListener[buttonListener]; 
    aboutPointField@setText["0"];

    (* Create the range field.
    *)
    rangeField = JavaNew["javax.swing.JTextField"]; 
    rangeField@addListener[buttonListener];
    rangeField@setText["3"];

    (* Create the function label.
    *)
    functionLabel = JavaNew["javax.swing.JLabel"]; 
    functionLabel@setText["f(x) = Sin[x]"];

    (* Create the about point label.
    *)
    aboutPointLabel = JavaNew["javax.swing.JLabel"]; 
    aboutPointLabel@setText["About Point: 0"];

    (* Create the range label.
    *)
    rangeLabel = JavaNew["javax.swing.JLabel"]; 
    rangeLabel@setText["Range: 3"];

    (* Create the compute button.
    *)
    computeButton = JavaNew["javax.swing.JButton"]; 
    computeButton@setText["Compute Taylor"];
    computeButton@setActionCommand["Compute Taylor"]; 
    computeButton@setPreferredSize[JavaNew["java.awt.Dimension", 100, 30]]; 
    computeButton@addActionListener[buttonListener];

    (* Create the title label.
    *)
    title = JavaNew["javax.swing.JLabel"]; 
    title@setText["Taylor Series"];

    (* Create the slider listener.
    *)
    sliderListener = 
ImplementJavaInterface["javax.swing.event.ChangeListener",
 {"stateChanged" → "computeTaylor"}];

    (* Create the button listener.
    *)
    buttonListener = 
ImplementJavaInterface["javax.swing.event.ActionListener",
 {"actionPerformed" → "computeTaylor"}];

    (* Create the series label.
    *)
    seriesLabel = JavaNew["javax.swing.JLabel"]; 
    seriesLabel@setText["Series: 3 terms"];

    (* Create the series text area.
    *)
    seriesTextArea = JavaNew["javax.swing.JTextField"]; 
    seriesTextArea@addListener[buttonListener];
    seriesTextArea@setText["Sin[x]"];

    (* Create the plot label.
    *)
    plotLabel = JavaNew["javax.swing.JLabel"]; 
    plotLabel@setText["Plot of Sin[x]
for the Taylor series of degree 3"];

    (* Create the compute Taylor button.
    *)
    computeTaylor = JavaNew["javax.swing.JButton"]; 
    computeTaylor@setText["Compute Taylor"];
    computeTaylor@setActionCommand["Compute Taylor"]; 
    computeTaylor@setPreferredSize[JavaNew["java.awt.Dimension", 100, 30]]; 
    computeTaylor@addActionListener[buttonListener];

    (* Display the window.
    *)
    window@add(topPanel, BorderLayout.NORTH);
    window@add(plotPanel, BorderLayout.CENTER);
    window@setVisible[true];
    window@pack();
    window@show();
]
LoadJavaClass("java.awt.Color");
plotPanel.setBackground(Color\`white);
plotPanel.setPreferredSize[JavaNew["java.awt.Dimension", 500, 500]];

(* Add the two panels to the top and bottom of the window. *)
LoadJavaClass["java.awt.BorderLayout"];    
window\@getContentPane[]@add[topPanel, 
returnAsJavaObject[BorderLayout\`NORTH]];    
window\@getContentPane[]@add[plotPanel, 
returnAsJavaObject[BorderLayout\`SOUTH]];

(* Now create and configure the various components. There are
  two main rows: row1 has all the text fields and Compute button,
  and row2 has the slider. Use Box, Border and Glue components to
  ensure a nice layout on all platforms and at varying window
  sizes.
*)
LoadJavaClass["javax.swing.Box"];    
row1 = Box\`createHorizontalBox[];
row2 = Box\`createHorizontalBox[];
functionField = JavaNew["javax.swing.JTextField", ToString[f], 10];
functionField@setMaximumSize[JavaNew["java.awt.Dimension", 20, 20]];
aboutPointField = JavaNew["javax.swing.JTextField", ToString[aboutPoint], 2];
aboutPointField@setMaximumSize[JavaNew["java.awt.Dimension", 8, 20]];
aboutPointLabel = JavaNew["javax.swing.JLabel", " About point: "];
functionLabel = JavaNew["javax.swing.JLabel", "Function of x: "];
rangelabel = JavaNew["javax.swing.JLabel", " Range: "];    
rangeprimeField = JavaNew["javax.swing.JTextField", ToString[range], 3];    
rangeprimeField@setMaximumSize[JavaNew["java.awt.Dimension", 8, 20]];
computeButton = JavaNew["javax.swing.JButton", "Compute"];    
row1@add[Box\`createHorizontalGlue[]];    
row1@add[functionLabel]; row1@add[functionField];    
row1@add[Box\`createHorizontalGlue[]];    
row1@add[aboutPointLabel]; row1@add[aboutPointField];    
row1@add[Box\`createHorizontalGlue[]];    
row1@add[rangelabel]; row1@add[rangeprimeField];    
row1@add[Box\`createHorizontalGlue[]];    
row1@add[computeButton];    
row1@add[Box\`createHorizontalGlue[]];    
title = JavaNew["javax.swing.JLabel", "Taylor Series Explorer"];    
title@setFont[JavaNew["java.awt.Font", "Dialog", BitOr[Font\`BOLD, Font\`ITALIC], 24]];
title@setAlignmentX[Component\`CENTERUALIGNMENT];
LoadJavaClass["javax.swing.BorderFactory"];
title@setBorder[BorderFactory`createEmptyBorder[20, 0, 20, 0]];
slider = JavaNew["javax.swing.JSlider", 1, 10, 1];
slider@setValue[nTerms];
slider@setSnapToTicks[True]; slider@setPaintLabels[True];
slider@setPaintTicks[True]; slider@setMajorTickSpacing[1];
slider@setMaximumSize[JavaNew["java.awt.Dimension", 180, 50]];
slider@setBorder[BorderFactory`createEmptyBorder[20, 0, 20, 0]];
row2@add[JavaNew["javax.swing.JLabel", " Number of terms: "]];
row2@add[slider];
seriesLabel = JavaNew["javax.swing.JLabel", "Series expansion"];
seriesLabel@setAlignmentX[Component`CENTERUALIGNMENT];
seriesLabel@setBorder[BorderFactory`createEmptyBorder[10, 0, 4, 0]];
seriesTextArea = JavaNew["javax.swing.JTextArea"]; seriesTextArea@setRows[4];
seriesTextArea@setFont[
    JavaNew["java.awt.Font", "Monospaced", Font`PLAIN, 12]];
plotLabel = JavaNew["javax.swing.JLabel", "Plot"]; plotLabel@setAlignmentX[Component`CENTERUALIGNMENT];
plotLabel@setBorder[BorderFactory`createEmptyBorder[10, 0, 4, 0]];

(* Add everything to the top panel. *)
topPanel@add[title]; topPanel@add[row1]; topPanel@add[row2];
topPanel@add[seriesLabel]; topPanel@add[seriesTextArea];
topPanel@add[plotLabel];
window@pack[];

(* Define the function to be called when the button is clicked
or slider dragged. *)
computeTaylor[___] :=
    Module[{func, about, rng, sliderVal, series, plot},
        func = ToExpression[functionField@getText[]];
        about = ToExpression[aboutPointField@getText[]];
        rng = ToExpression[rangeField@getText[]];
        sliderVal = slider@getValue[];
        series = Collect[Normal[Series[func, {x, about, sliderVal}]], x];
        seriesTextArea@setText[ToString[series, OutputForm]]; Block[{$DisplayFunction = Identity},
            plot = Plot[{func, series}, {x, about-rng, about+rng},
                        PlotStyle → {Hue[0], Hue[0.7]}];
            plotPanel@setMathCommand[ToString[plot, InputForm]]
        ];
        (* Add behavior. *)
        buttonListener = JavaNew["com.wolfram.jlink.MathActionListener"];
]
Now run this final Taylor program. Quit the program by clicking the window’s Close box.

\[
\text{Taylor[]}
\]

You can also specify the initial function, point, range, and number of terms.

\[
\text{Taylor}[\text{Cos}[x], 0, 3, 8]
\]

### Where to Go from Here

This article has been a quick introduction to the subject of creating user interfaces for Mathematica programs using J/Link. There are many techniques and tips that were not covered. For more information, consult the J/Link User Guide. In particular, look at Section 1.2.7, “Creating Windows and Other User Interface Elements.” There are also several examples provided with J/Link. They can be found in the `<Mathematica directory>/AddOns/JLink/Examples/Part1` directory, and most are discussed in detail in Section 1.3 of the User Guide. Get yourself one of the many books on the Java user interface APIs, put a link to the Java Platform JavaDocs in your browser favorites, and get programming!

### About the Author

Todd Gayley is the creator of J/Link. He is currently working to bring the same capabilities to users of Microsoft’s .NET framework.

**Todd Gayley**  
Director of Java Technology
Wolfram Research, Inc.
tgayley@wolfram.com