Migrating From Java 1.0 to Java 1.1

By Laura Bennett
BeanMachine Developer, IBM

Abstract

Even before Java 1.0 was shipped in 1996, the hype around it was off and running. Developers saw it as the product to replace C++, a language whose syntax was found to be too complex. Java offered a way to develop object-oriented software with a language that was not particularly hard to learn and not as difficult to develop software with. A year later, JavaSoft developers found many ways to improve Java, which resulted in a new version known as Java 1.1.

This paper assists those already familiar with Java 1.0 in making the transition to Java 1.1.x. In this paper, I point out the important changes, such as the deprecated methods and event handling and new features such as JavaBeans and internationalization. I will not be able to detail every single change-that would require writing a thick book. Instead, I will focus on the most significant changes and improvements that occurred over the last year.

What Changes Have Been Made to the Existing Code?

Updates to Java include changes to the base language specification, rewrites to the event handling process, and the introduction of a new and improved Java Native Method Interface (JNI) for Java 1.1. The following section covers these changes.

Base Language Specification Updates

As one would expect, several methods have been added, changed, or deleted. The number of classes have more than doubled in the latest version of Java. You will be notified of the changes as you compile your code instead of later during run time. The notice is a ?deprecated? warning message issued by the Java compiler javac. If you compile with the ?-deprecation? flag, javac will provide a detailed warning about each use of a deprecated feature. You will see that most of the changes occurred in the Abstract Window Toolkit (AWT). A sample of those changes appear in the table below. See the Java online documentation for a complete list.

JAVA 1.0 Deprecated Method	JAVA 1.1 Replacement
addItem	add
appendText	append
inside	contains
layout	doLayout
getPageIncrement	getBlockIncrement
deliverEvent	dispatchEvent
getBoundingBox	getBounds
locate	getComponentAt
insets	getInsets
countItems	getItemCount
hide	setVisible(false)
insertText	insert
inside	contains
isSelected	isIndexSelected
resize	setSize
show	setVisible(true)
size	getSize

While deprecated methods are supported at this time, it is recommended that you update your code to use the 1.1 version of those methods because support for older code will probably disappear soon.

Event Handling
The biggest change to Java is in the event handling process. The event model for Java 1.0 was inheritance-based. Let us first review this model and its problems. It was cumbersome and slow in developing large distributed programs. There were three methods to handle events:

- deliverEvent(),
- postEvent(),
- and handleEvent().

Every event that occurred was passed up the hierarchy of components in the graphical user interface (GUI) until it was intercepted or until it reached the top of the chain, where the event was disposed of. The problem was that there was only one entryway for all events. Imagine fifty mouse moves trying to get in the door and expecting to be handled at the same time. Also, this old model is heavyweight because all events are propagated even if they are not of interest. User interfaces with many layers of components can slow applications and applets for no good reason. Another problem is that return values from events will probably be sent back to the wrong AWT component; these returns are especially likely because all events had to pass through the same three methods. As applications and applets became more complex, these returns became more of a problem. Component classes not participating in the same lineage as the AWT Component and Container classes required extra code to receive the same events, because they would not receive any events through the normal chain of command. On the plus side, however, the inheritance-based process did make event handling easy to grasp and program. Therefore, it is still used for simple applets and applications.

Java 1.1 has completely redesigned event handling to include an additional event model known as delegation. With delegation, there are sources of events and there are listeners for the generated events. I liked the analogy I saw in one book, which related the new event model to the following scenario: Think of yourself as an event source and a friend as an event object. Someone kicks you and you say "Ouch!" You, the event source, produced a sound event object containing the "Ouch!" event. The "Ouch!" event travels through the air to the event listener, our friend. Your friend has an event listener object specific for picking up sounds: ears! The sound event listener receives the

"Ouch!" event through the sound event object. It's a simple concept; but don't worry, it need not be painful to grasp!

The name *delegation* derives from the fact that event sources are delegated to specific event listeners. User interface components that have received some type of user action are usually the event sources, and classes that contain methods to handle those user actions are event listeners. The event listeners are interfaces that have subclassed themselves from java.util.EventListener. The listener listens for and processes Java events. The event source has to register with the event listener. In layman's terms, the event source is basically saying, *"Hey, pay attention to me when I've been touched!"* For instance, when a user clicks a mouse, the event source is notified. When the actual event has occurred, an event object is created and is passed to the appropriate listener method for handling. The event object passes class constants, which represent the individual event type that occurred. In this model, any user interface objects can request to be notified of events. The only stipulation is that they register with an event listener to show interest in the event. Let's look at a code sample. In this instance, a button registers with an ActionListener so that it can respond to a typical button event like a button press.

```
import java.awt.*
import java.awt.event.*;
import java.applet.*;

public class ShowEvent extends Applet
{
Button myButton;

public void init()
{
myButton = new Button("Yes"); // the event source
myButton.addActionListener // button will listen for action
// events
}
```

What types of events are implemented by the Java API? Well, the java.awt.Event class has been replaced by a set of classes that extend from java.util.EventObject. They can be grouped as follows:

Low-Level Events: very specific (subclassed from java.awt.event.ComponentEvent)	FocusEvent, InputEvent, KeyEvent, MouseEvent, ContainerEvent and WindowEvent
High-Level Events: very general	ActionEvent, AdjustmentEvent, ItemEvent and TextEvent

Each of these event classes represents a group of events. For instance, MouseEvent represents a mouseUp(), mouseDown(), and mouseMove(). The table below details what events are available and what methods are fired up based on a user action against a given component.

1.0		1.1
Event/Method	**Generated By**	**Methods**
ACTION_EVENT/action	Button, List, MenuItem, TextField	actionPerformed(ActionEvent)
	Checkbox, CheckboxMenuItem, Choice	itemStateChanged(ItemEvent)
WINDOW_DESTROY WINDOW_EXPOSE WINDOW_ICONIFY WINDOW_DEICONIFY	Dialog, Frame	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) (no 1.0 equivalent) windowActivated(WindowEvent) (no 1.0 equivalent) windowDeactivated(WindowEvent) (no 1.0 equivalent)
WINDOW_MOVED	Dialog, Frame	componentMoved(ComponentEvent) componentHidden(ComponentEvent) (no 1.0 equivalent) componentResized(ComponentEvent) (no 1.0 equivalent) componentShown(ComponentEvent) (no 1.0 equivalent)
SCROLL_LINE_UP SCROLL_LINE_DOWN SCROLL_PAGE_UP SCROLL_PAGE_DOWN SCROLL_ABSOLUTE SCROLL_BEGIN SCROLL_END	Scrollbar	adjustmentValueChanged(AdjustmentEvent)
LIST_SELECT LIST_DESELECT	Checkbox, CheckboxMenuItem, Choice, List	itemStateChanged(ItemEvent)
MOUSE_DRAG/mouseDrag MOUSE_MOVE/mouseMove	Canvas, Dialog, Frame, Panel, Window	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
MOUSE_DOWN/mouseDown MOUSE_UP/mouseUp MOUSE_ENTER/mouseEnter MOUSE_EXIT/mouseExit	Canvas, Dialog, Frame, Panel, Window	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent) (no 1.0 equivalent)
KEY_PRESS/keyDown KEY_RELEASE/keyUp KEY_ACTION/keyDown KEY_ACTION_RELEASE/keyUp	Component	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent) (no 1.0 equivalent)
GOT_FOCUS/gotFocus LOST_FOCUS/lostFocus	Component	focusGained(FocusEvent) focusLost(FocusEvent)
No 1.0 equivalent.		componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
No 1.0 equivalent.		textValueChanged(TextEvent)

The listener class implements an interface based on the type of events. In each of the interfaces, a set of methods need to be overridden. All of the listener classes extend from java.util.EventListener. The listener's responsibility is to simply *listen* to a designated component to see if any events were initiated from these tagged objects. If that happens, then it is up to the listener to call the appropriate method to handle the event. What type of listeners are available in Java 1.1? See the table below for a summary.

Low-Level Listener Interfaces	ComponentListener, ContainerListener, FocusListener, KeyListener, MouseListener, MouseMotionListener and WindowListener
Semantic Event Listener Interfaces	ActionListener, AdjustmentListener, ItemListener and TextListener

Let's look at another sample. In this case, we are creating a text field and asking or *registering* it with the textListener. So, if its value were to change, then the listener that has been registered with *aTextField* will immediately call the *textValueChanged()* method.

```
......

private TextField aTextField = null;
aTextField.setBackground(Color.gray);
aTextField.addTextListener(this);

public void textValueChanged(TextEvent e)
{
if
(e.getSource() == localLocation)
String textValue = aTextField.getText();
}

.......
```

Keep in mind that a component can register with a multitude of listeners. Also, listeners can be removed from a component as easily as they are added with the *removeXXXListener()* method.

Note that even though the original event model is still available, it is highly recommended that you don't mix the two models in the same application or applet.

Interfacing with Native Methods
The inventors of Java hid machine-specific facilities from the programmer. However, there are times when access to native DLL code is required. For instance, suppose you need to use a feature of your operating system that is not yet available in Java, such as real-time audio from a microphone. Java provides a means for accessing native methods in order to make Java more powerful and to let programmers include platform-specific features. In Java 1.0, native methods were called with the following process:

1. Load the DLL using the *loadLibrary()* call.
2. Declare the method with the *native* directive.
3. Call the method using normal Java calling conventions.

The Java interpreter went between the Java code and the native DLL. Native methods were created using the *javah* utility, which was used to produce a header file and an associated stub C file.

There are disadvantages to using this method. The most obvious is that you would be working against Java's main purpose: platform-independence. Another disadvantage is that the process for interfacing with native methods is not exactly the same across all platforms; therefore, you must write different code for each environment you support. Incidentally, other vendors who have licensed Java are also writing native method interfaces; however, using Sun's Java 1.0 interfaces with third-party vendor's code is not typical. One more disadvantage is that interfacing with native methods via the original design prohibits running applets in a browser. *?The Java Tutorial?* by Mary Campione and Kathy Walrath discusses some of the restrictions that Java applets enforce; one of them is disallowing applets from loading code written in any non-Java language. Ah, but we do have a solution. Java 1.1 has introduced the Java Native Method Interface (JNI) to overcome the deficiencies in the 1.0 model. It works the same way across all platforms and even works with third-party interfaces. When a native method is implemented and the JNI is used, one part implements the Java side, and the other contains the native methods you wish to use.

1. Declare the native methods in Java. Declare any methods that you intend to implement natively. Also, you need to load the native library that contains the native methods you with to use.
2. Implement the native methods in C/C++. It is in C/C++ that you implement the native methods and compile them, using a native compiler. Here, you obtain a pointer to a Java class; retrieve a pointer (ID) to this class's method, constructor, field, or other members; and finally access the specified Java class's member.

You declare a native method by adding the keyword ?native? to the method declaration. For example:

```
// Declare native method.
native String modifyString(String myString);
```

Then load the native library that contains the native methods. For example:

```
// Load the library mylib.dll
static
{
System.loadLibrary("mylib);
}
```

Loading the library in a static block of code ensures that it is only invoked once.

Although Java 1.1 still supports the older model, it is recommended that you convert to the JNI model because the older model will eventually be phase out.

What New Features Have Been Added?

When Java was first released, its focus was platform-independence, not bells and whistles. This lack of features caused developers to continually look for more components and containers so that they could build their applications. JavaSoft has enhanced the tool kit with many of these requested features, such as customizable desktop colors, pop-up menus, and scrolling. JavaBeans, a component technology, is another added bonus to Java 1.1. Also, the latest level of Java introduces a tool for creating a new file format called the Java ARchive (JAR) file format. This tool is an enormous help in using Java applets on the Internet. Last, it only makes sense that JavaSoft has added classes for internationalization, since the Internet is in the global market. There are many other new features, such as Inner classes, Object serialization, reflection, Java Database Connectivity, and Remote Method Invocation. In the following sections, I will highlight a handful of the new features.

Customizing Colors via the System Colors
One important feature is customizable desktop colors. Most GUI applications allow users to customize their windows via the system configuration process. In Windows, customization is done through the Appearance tab on the Display Properties notebook within the Control Panel. Users can establish central themes for their backgrounds and all applications will follow suit-except Java applications. But now, with the addition of this new feature, Java programs can query the current desktop to determine the color scheme used and closely match it. A new class has been created, known as *java.awt.SystemColor.* Keep in mind that Java does not allow dynamic change of colors; it will attempt a match only on start-up of the program. The example below demonstrates how to set the background based on a query of the user's current system color:

```
setBackground(SystemColor.window);
```

Not all environments provide support for custom desktop colors. In this case, a default color will be substituted.

Tabbing Between Components
Another new feature is tabbing between components. Users of windowing systems are accustomed to tabbing with the tab key (or Shift+Tab for reverse movement). Java 1.0 does not support this tabbing. Java 1.1 does, which will make your programs more compliant with modern graphical user interfaces.

Java implemented focus traversal through a FocusManager object. The object, which is registered on a Window, handles the focus traversal based upon both the order of the creation of the components and whether a component will accept keyboard focus when tabbing. There is an *isFocusTraversable()* method that will answer the latter.

Pop-Up Menus
Pop-up menus were added in Java 1.1. Upon a right mouse-click, a temporary menu appears. The pop-up menu's parent is the GUI component, not a menu bar. The new class, java.awt.PopupMenu, contains three methods: *add()*,*remove()*, and *show()*. The example below shows the creation of a pop-up menu and the addition of one item to the menu:

```
setPopupMenu popup = new PopupMenu("Edit");
MenuItem mItem = new MenuItem("Test");
popup.add(mItem);
```

To activate a pop-up menu, a mouse event must be trapped by adding a mouse event listener to the main frame. The *show()* method actually invokes the pop-up menu and takes three parameters: The first is the container to which the pop-up menu belongs, and the second and third parameters represent the x and y locations for the pop-up menu.

Component Technology: JavaBeans
The biggest extension to Java is support for reusable software components called beans. The JavaBeans 1.01 specification offers the following definition of a Java bean: *"A Java Bean is a reusable software component that can be manipulated visually in a builder tool."* A whole new package, known as java.beans, has been added to define classes and interfaces that work with beans.

Those who are familiar with object-oriented programming (OOP) know that it is better to design programs as collections of objects instead of functions; reuse is easier and implementation is cleaner. Component technology picks up where OOP left off. A component is a piece of software that is completely reusable. It should be able to stand alone and also be combined with other components in a larger program. With the release of Java 1.1, Sun provided the JavaBeans API. JavaBeans takes component technology one step further by facilitating independence among platforms, unlike other component technologies. Originally, component technology was focused on visual components that were used to develop applications rapidly. These old components were modular, but they didn't stand alone. They were used in a variety of Windows development environments, such as Visual Basic and Visual C++. Components in the Java environment are independent and can stand alone.

A bean usually supports the following features:

- **Introspection.** Introspection allows a Java bean to be looked at during run time. This introspection can be used by a component editor, graphical user interface (GUI) editor, or a bean programmer to find out information about a specific Java bean; such information could include the bean's properties, methods, or events..
- **Customization.** This feature allows you to specify and/or choose among properties in a bean. For example, in a component editor or GUI builder, users copying a bean to their projects need to be able to add special attributes to the bean and/or specify those attributes.
- **Events.** Events are one of the basic parts of a Java bean. Through events (and event handling), you can integrate your bean with other components or a component editor (or GUI builder).
- **Properties.** A property represents a changeable attribute that affects the behavior and/or appearance of the bean to which it belongs.
- **Persistence.** In most beans, it is necessary to save the bean and its customizations, including both its external properties and internal state.

Introspection
Introspection was designed because a builder tool needs some way of identifying and analyzing the

properties, events, and methods in a Java bean. Methods defined in the introspection classes follow a specific design pattern. The uniform syntax allows tools and other libraries to recognize and use these patterns for instrospection. The name is based on the property, event, or method contained in the Java bean. For instance, if you wanted to retrieve the value of a Boolean property, you would use a corresponding isXXX() method (where XXX is the name of the property). Or, if a Java bean has implemented the BeanInfo interface, you can learn about the properties, methods, and events of that Java bean by calling methods from that interface. If you call a BeanInfo class, it returns a descriptor object containing information on the specified JavaBean.

Customization
The user's power to customize is vital. The user should be able to customize a clock part, for example, with properties like format (digital or analog) or time zone. Beans design should allow these properties to be retrieved and set. A property editor allows the user to change properties in a visual format. Although the developer usually creates a property editor, the Java API can also facilitate creation of property editors. The classes are contained in java.beans.PropertyEditorManger class. For more sophisticated beans, the JavaBeans API has defined a Customizer interface that behaves like a wizard, helping you through the process step-by-step. Any class you design to be a customizer should inherit the java.awt.Component interface and should implement the interface.

Events
JavaBeans use the delegation event model to communicate with each other. When designing a bean, you must decide which events it will transmit to registered listeners outside the bean. By default, a bean can transmit any events supported by its parent class. One thing to consider is what happens in a bean with subcomponents if an event occurs on one of them-for example, in a calendar bean, which contains a button instance for each day of the month. When the user clicks one of these buttons, the event source is the button, not the calendar bean. So the calendar bean must first intercept the event that occurred on the subcomponent and then create a new event. Post the new event to the system-event queue in such a manner that any of the bean's registered listeners will receive it.

Properties
In the JavaBeans architecture, properties give you the power to specify attributes for beans. Properties affect either a bean's appearance or its behavior and are always accessed via accessor methods, through which a property can be examined and modified. A good example of a property is a button label for the AWT component *java.awt.Button*, which has two methods, *getLabel()* and *setLabel()*. These methods let you read or specify the label for a particular button object. The JavaBean architecture also provides support for bound and constrained properties. A *bound* property notifies a set of listeners whenever the value of the property changes. This notification is done through event handling. *Constrained* properties ask permission from a set of listeners to make a value change. The listener has the option of accepting or denying the change.

Persistence
The bean architecture facilitates persistence during both design time and run time through the use of object serialization, in which the class or classes relating to the bean can be serialized and stored in a bean. Serialization basically means "flattening an object into a serial stream of data and saving the whole thing to disk in a recognizable state so it can be restored later." [*JavaBeans for Dummies,* by Emily VanderVeer and Joseph McIntyre.]

There are many advantages to designing with beans. JavaBeans supports internationalization, so you can design your beans to be used in different languages. They also support a wide variety of distributed technology that is available in Java 1.1. And JavaBeans is not propriety to a single component architecture. This means that, when you create a bean component, it can be used on any platform, including Microsoft's ActiveX. The same bean could then be used as a Live Object, using Apple's OpenDoc architecture, and, finally, it could be used in Netscape's Navigator via LiveConnect.

Below is an example JavaBean taken from the JavaBeans for Dummies. To use it, the first thing you need to do is download the JDK 1.1. The name of the file is FlavoredJellyBean.java; it inherits from a bean called JellyBean.

```
import java.awt.*
import java.beans.*;
import sunwdemo.jelly.*;

public class FlavoredJellyBean extends JellyBean {
/** The FlavoredJellyBean class inherits all of
JellyBean’s properties and methods and then adds
one property (flavor) and two accessor methods. */

public String getFlavor() { // getter
/* getFlavor() is a very simple accessor method */
return flavor;
}

public void setFlavor(String newFlavor) { // setter
flavor = newFlavor;
}

private String flavor = licorice; //private property
}
```

An increasing number of JavaBeans are available from software component developers. Integrated Java development environments typically come with several beans that developers can use. Some of the most popular beans are those that function as higher-level GUI widgets, such as multi-column list boxes and toolbar managers. Beans can also be small applications, like drawing programs, and non-visual software components, like a timer widget, which ticks at pre-determined intervals.

Java Archives (JAR Files)
One of the main complaints about Java was its slowness, especially when loading applets with many graphics. A new package was introduced in Java 1.1 called *java.util.zip*. This feature, called Java Archive (JAR) file format, uses zipping, which makes it possible to read and write compressed files. All of an application’s or applet’s files and/or directories are taken and compressed into one single file; this file can hold sound, image, and compiled class files. Therefore, in the case of an applet, only one transfer must take place to download the entire applet and all of its supporting files.

The process of creating JAR files requires two steps. First you must use the *jar.exe* tool to create the actual JAR file. Second, in the <APPLET> tag, you must specify the JAR file that belongs to the applet. Voila! You have made your applets more efficient and less time-consuming. JAR files usually also include a manifest file named manifest.mf, which describes the contents of the JAR file. The JAR tool automatically builds the manifest file based on the files that are added. This is important because a single bean can be composed of several classes. Typically the jar.exe command is not run directly; instead *nmake* or *make* is run, and it will run the command.

Here is an example of running the jar command in its simplest form:

```
jar cf test *.*
```

The options are defined in the table below. The *.* means that every file in the current directory will be added to the archive. All directories are recursively added as well. Below is a table of the options to the JAR command:

Option	Definition
c	Creates a new archive
F	Specifies the JAR file
m	Use manifest template file
t	List table of contents for archive
x	Extract named (or all) files from archive
v	Generate verbose output on standard error

The final step is to include the JAR file within an HTML file. It is specified using the archive parameter in the APPLET HTML tag. See the example below.

```
<applet code="Test.class" WIDTH=300 HEIGHT=200
<param=ARCHIVES value="Test.jar">
</applet>
```

You can view the contents of a JAR file with the WinZip utility in addition to using the command

jar -tf <JAR FILE>

Internationalization
Internationalizing a program enables developers to compete in a global market. Java's capacity for switching languages is made possible through the Internationalization API. This API adds several new date, time, number, and text-formatting utility classes that were lacking in the 1.02 version. The classes are of four types:

- Locale information: where the program is running and in what language
- Font information and support: used to load and display fonts for different languages
- Text iteraters: used to traverse text containers
- Time and date interfaces and containers: used to represent the time and date of particular locales

The first step in internationalization is identifying in which locale a program needs to run. For those who are new to internationalization, the locale identifies a region with a particular language. The java.util package contains a locale class that sets the locale. The operating system relays to the Java program what the locale is. When Java initializes the input system, it reads font.properties for the locale. To set the default local in the program to Japanese, you would enter the following line of code:

Locale.setDefault(Locale.JAPANESE);

Next, a program must be able to read and write in the current locale. Since Java uses Unicode encoding for its characters and strings, its localized characters simply need to be converted from the local character-encoding to Unicode when reading and vice versa when writing. The *java.io* package has several new classes for this purpose. Classes in the *java.text* package convert numbers, dates, currency, and percentages to the appropriate locale. In the *java.util* package, a Collator class compares strings.

Finally, an application must display characters within the current locale. Java has made this step easy. You need to subclass the *ResourceBundle* class in which you define your messages that are read in at run time. Then create a subclass for each language. The ResourceBundle contains the messages your application uses. Each message has a key word association. On a search, if the key word is not found within a bundle, the parent ResourceBundle will automatically search for the key word. One other useful class is the *MessageFormat* class of the *java.text* package, which allows you to substitute dynamic values into static messages, such as error messages.

Where Do We Go From Here?
Without a doubt new versions of Java will be created and many features will be added. Such

features as full drag-and-drop of data between beans, new multimedia capabilities, and a way to dynamically exchange all sorts of data types between beans. In addition to enhancing features, people are working to increase the presence of Java in many other environments. Developers should continue to follow JavaSoft's Web site: http://www.javasoft.com to keep abreast of their activities with Java and to understand the direction they take.

What Does IBM Have to Offer in the Java Market?

Network computing, driven by the rise of the Internet and intranet, delivers the infrastructure that makes new business opportunities possible. A company's success in exploiting this unprecedented opportunity will be based largely on how quickly and efficiently they deploy Net-savvy applications. IBM has answered its competitors' challenges and has produced successful Java-enabled tools that compete in the open market.

BeanMachine is IBM's newest Web authoring tool targeted for non-programmers; it allows Webmasters to develop dazzling applets without knowing any Java code. BeanMachine generates a JAR and an HTML file that can be integrated with other tools, such as NetObjects Fusion and Lotus Notes 4.6. BeanMachine has recently been ported to Java 1.1 and is available for evaluation at http://www.lotus.com/home.nsf/welcome/beanmachine. Media-rich beans, such as animation, image, and database beans, are shipped with BeanMachine. Developers can also add their own beans using a simple wizard. BeanMachine has been internationalized and currently runs in the German, French, Spanish, and Italian locales and will soon run in Asian languages.

If you must write code, then use IBM's solution for developing Java applications: VisualAge for Java. This tool contains a class browser, an editor, and a debugger, and it offers incremental compilation. VisualAge for Java provides support for the basic component architecture in Java 1.1 and support for converting ActiveX components to JavaBeans. IBM's Visual Age for Java provides a powerful environment for creating and connecting your Web applications to existing databases and transaction systems. This connection allows programmers to extend server-based applications to the Internet or intranet instead of having to rewrite the application from scratch. Visit the Visual Age for Java Web site for more information: http://www.software.ibm.com/ad/vajava.

The VisualAge WebRunner Toolkit helps Java programmers create portable, robust, network-aware JavaBeans and high-performance, global Web server applications from components. For more information, see the Web site at http://www.software.ibm.com/ad/webrunner.

Visit IBM's central Java Web site, http://www.ibm.com/java, to see what other exciting things IBM is doing with Java.
