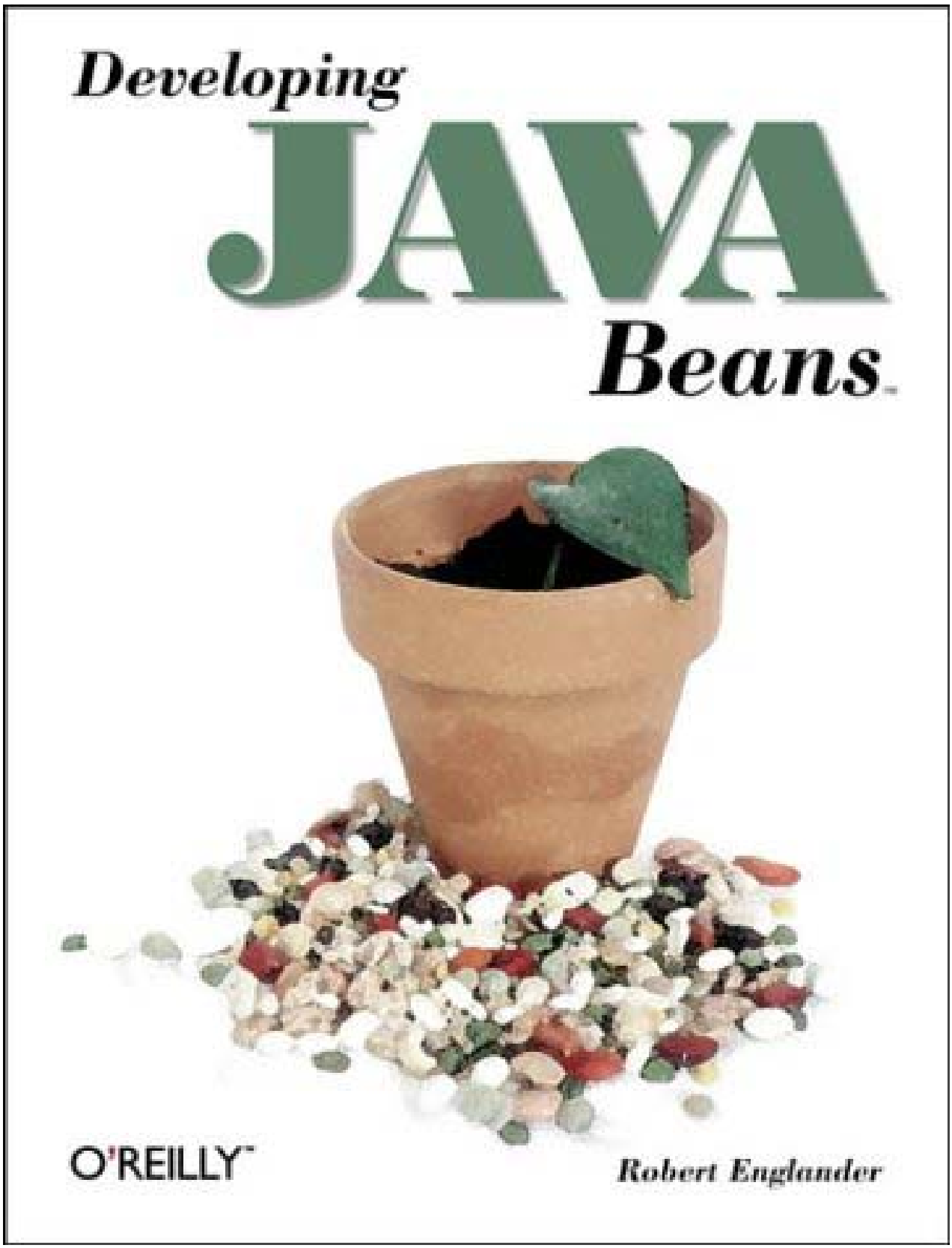


Dedication **2**

Preface **2**
Intended Audience 3
A Moment in Time 3
How the Book Is Organized 3
Conventions Used in This Book 5
Acknowledgments 5
How to Contact Us 6
Providing Feedback to the Author 6
Retrieving Examples Online 6

Chapter 1. Introduction **6**
The Component Model 7
The JavaBeans Architecture 9
JavaBeans Overview 11
Using Design Patterns 14
JavaBeans vs. ActiveX 14
Getting Started 15

Chapter 2. Events **15**
The Java Event Model 15
Events in the AWT Package 25

Chapter 3. Event Adapters **30**
Demultiplexing 30
Generic Adapters 35
Event Adapters in the AWT Package 40
Event Filtering 43
Event Queuing 44

Chapter 4. Properties **50**
Accessing Properties 50
Indexed Properties 53
Bound Properties 55
Constrained Properties 60
Handling Events for Specific Properties 64
A java.awt Example 64

Chapter 5. Persistence **70**
Object Serialization 71
The java.io.Serializable Interface 72
Class-Specific Serialization 78
Walking the Class Hierarchy 79

Serializing Event Listeners 90
Versioning 92
Object Validation 94
The java.io.Externalizable Interface 96
Instantiating Serialized Objects 99

Chapter 6. JAR Files 102
The jar Program 102
The Manifest 104
Using JAR Files with HTML 106
Using JAR Files on the CLASSPATH 107
Archive Signing 108
An Alternative to the jar Program 108

Chapter 7. The BeanBox Tool 109
Running BeanBox 109
Dropping Beans on BeanBox 111
Editing a Bean's Properties 111
Hooking Up Beans 113
Saving and Restoring the BeanBox Form 115
Adding Your Own Beans to BeanBox 115

Chapter 8. Putting It All Together 116
Defining the Temperature Control Simulator 117
Building the Simulator 118
A Sample Simulator Applet 145
Creating a JAR File 147
Recreating the Sample Using BeanBox 148

Chapter 9. Introspection 150
The BeanInfo Interface 150
Providing Additional BeanInfo Objects 166
Introspecting the Environment 168
The BeansBook.SimulatorBeanInfo Classes 169

Chapter 10. Property Editors and Customizers 173
Property Editors 174
Customizers 188

Chapter 11. ActiveX 197
The JavaBeans ActiveX Bridge 198
Technology Mapping 203
Using Beans in Visual Basic 204

Appendix A. Design Patterns 212

A.1 Event Objects 212
A.2 Event Listeners 213
A.3 Registering for Event Notification 213
A.4 Registering for Unicast Event Notification 213
A.5 Multiple Parameter Event Methods 213
A.6 Property Access Methods 214
A.7 Indexed Property Access Methods 214
A.8 Constrained Property Access Methods 214
A.9 Registering for Bound and Constrained Property Event Notifications 215
A.10 Naming a BeanInfo Class 215

Appendix B. The java.beans Package 215

Developing Java Beans

Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Dedication

For my daughter Jessica

Preface

JavaBeans is one of the most important developments in Java™ since its inception. It is Java's component architecture, which allows components built with Java to be used in graphical programming environments. Graphical development environments let you configure components by specifying aspects of their visual appearance (like the color or label of a button) in addition to the interactions between components (what happens when you click on a button or select a menu item). This means that someone can use a graphical tool to connect some Beans together and make an application without actually writing any Java code—in fact, without doing any programming at all. Developing an application isn't necessarily a matter of producing thousands of lines of code that can only be read by computer professionals. It's more like working with Lego blocks: you can build large structures using snap-together pieces. The result is that applications can be created by people who are good at designing user interfaces and aspects of the interaction between the user and the computer. The guts of an application can be written by software developers, who are great at coding, but not necessarily good at understanding users. This is how it should be, and in fact how it is in many other industries. The engineer who designed the engine of your car is certainly not the same person who designed the interior. JavaBeans allows us to make the same kind of distinction in the software business.

As Beans become widely available, we will see more developers using them to build applications. But before these applications can be built, someone has to build the components. That's what this book is all about.

You won't find any hype in this book, and you won't find vague descriptions of technology that may or may not appear in the future. JavaBeans is here now, and programmers must have the

information at hand to begin creating components. So if you're ready to get right into the techniques and concepts used by the JavaBeans architecture, and if you want to understand the underpinnings of the technology that makes it work, this book is for you.

Intended Audience

This book is for everyone who wants to know how to build reusable components using the JavaBeans architecture and Java class libraries. It is designed to be used by programmers, students, and professionals that are already familiar with Java, so it doesn't concentrate on any of the basic concepts or syntax of the language. However, if you are experienced with other object-oriented languages such as C++ or Smalltalk, you should be able to follow along. If you aren't familiar with Java, you may want to keep a book on the Java language close by, like the *Java Language Reference* (O'Reilly). In any case, the material should prove useful to both novice and experienced programmers.

One chapter discusses the interaction between JavaBeans and ActiveX components, and has some examples using Visual Basic. I assume that readers interested in this topic are already familiar with VB and the ActiveX component architecture, and don't attempt to explain them. Many good books on Visual Basic are available if you need an introduction.

A Moment in Time

The JavaBeans architecture continues to evolve. This book describes the technology in its first release, coinciding with version 1.1 of the Java Development Kit. The concepts and techniques that I cover will continue to be relevant in future Java releases, and new things will no doubt be added along the way. With the rapid rate of change that Java is currently undergoing, the best that any book can do is capture a moment in time.

How the Book Is Organized

The chapters in this book are organized so that each one builds upon the information presented in previous chapters, so it's best if you read the chapters in order.

Chapter 1

> This chapter provides a general description of the component model, followed by an overview of the JavaBeans architecture.

Chapter 2

> This chapter describes the event model introduced in Java 1.1. It covers event listener interfaces, event objects, and event sources, and covers the semantics of event delivery. Topics also include design patterns, event listener registration, and multicast and unicast events.

Chapter 3

> This chapter describes how to use event adapters to simplify an event listener, and how to adapt an object to an event listener interface. Topics include demultiplexing, using low-level reflection to create generic adapters, event filtering, and event queuing.

Chapter 4

This chapter describes properties, the named attributes that define the state and behavior of a component. Properties represent some part of the Bean that is likely to be manipulated by nontraditional programming techniques such as visual editors. Topics include design patterns, accessor methods, indexed properties, bound and constrained properties, and property-related events.

Chapter 5

This chapter describes the use of object serialization provided by Java 1.1 for saving and restoring the state of a Bean. It discusses what can and can't be stored and how storage and retrieval is accomplished. Topics include automatic and class-specific serialization, serializing an object's class hierarchy, class versioning and compatibility, and serialized versions of Beans.

Chapter 6

This chapter describes the Java Archive (JAR) file, used to package one or more Beans, classes, or associated resource files. The *jar* utility provided with the JDK is discussed, along with manifest entries, and how to use them to identify a component as a Bean.

Chapter 7

This chapter introduces the BeanBox program, a visual Bean-testing tool provided with the Beans Development Kit (BDK). It describes how to include your Beans for testing in the BeanBox, how to wire Beans together based on their events and properties, and how to save and restore a collection of inter-operating Beans.

Chapter 8

This chapter takes all of the concepts and techniques described in the previous chapters and uses them to develop a fully functioning example. This example includes some Beans, as well as supporting classes, that all work together to create a temperature simulator. The Beans that are developed are wired together using traditional programming, as well as the BeanBox testing tool.

Chapter 9

This chapter introduces the introspection mechanism used to expose the events, methods, and properties of a Bean. It describes how to create special Java classes that explicitly provide this information for your Beans.

Chapter 10

This chapter describes property editors, the Java classes that are used by programming tools to provide visual editors for changing a Bean's property values. It shows you how to use the standard property editors, and how to create your own custom property editors. Customizers are interfaces to customize the behavior and/or appearance of an entire Bean. This chapter shows you how to create your own customizer, as well as how to use a property editor as part of your customizer.

Chapter 11

This chapter describes the JavaBeans ActiveX Bridge, a tool that allows you to package your Beans as ActiveX components. The mapping between the two technologies is discussed, as well as those parts of JavaBeans that don't map well to ActiveX. You will also see how Beans can be used in an ActiveX container.

Appendix A

All of the design patterns that are introduced throughout the book are put here for easy reference.

Appendix B

This is a basic class reference for all of the classes and interfaces in the `java.beans` package. Each class and interface has a brief description, along with a class definition that shows its methods.

Conventions Used in This Book

`Constant Width` is used for:

- Anything that might appear in a Java program, including keywords, operators, data types, constants, method names, variable names, class names, and interface names, and also for Java packages.
- Command lines and options that should be typed verbatim on the screen.

Italic is used for:

- Pathnames, filenames, and Internet addresses, such as domain names and URLs. Italic is also used for Bean properties and executable files.

Acknowledgments

I was first introduced to Java by my friend Rinaldo DiGiorgio. He tried to convince me that Java was going to be big, and that I should get involved. Eventually, I broke down and followed his advice. More recently, he had to twist my arm to take a hard look at JavaBeans, and subsequently to write this book. I am grateful to Rinaldo for pointing me in the right direction.

My sincere thanks go to Mitch Duitz, Hugh Lynch, and John Zukowski for their detailed technical reviews of the book, and for providing valuable feedback. They managed to find errors, point out omissions, and offer advice that makes this a better book than it would have been without their help. Thanks to Max Spivak and Jonathan Knudsen for reading the draft and offering their comments and suggestions. My appreciation also goes to Mike Loukides, the book's editor, for believing this was an important book to write and for helping me to do so.

A thank you is due to the O'Reilly design and production crew: David Futato, the production editor and copyeditor for the book; Nancy Kotary for proofreading and quality control; Seth Maislin, who produced the index; Edie Freedman, who designed the cover; Nancy Priest, for internal design; Chris Reilley and Rob Romano, who were responsible for the figures; and Sheryl Avruch, the production manager.

I also have to thank my friends and family for putting up with me while I was writing. Everyone's encouragement helped me to finish this project.

How to Contact Us

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
1-800-998-9938 (in the U.S. or Canada)
1-707-829-0515 (international/local)
1-707-829-0104 (FAX)

You can also send us messages electronically. To be put on the mailing list or request a catalog, send email to:

info@oreilly.com

To ask technical questions or comment on the book, send email to:

bookquestions@oreilly.com

We have a web site for the book, where we'll list examples, errata, and any plans for future editions. You can access this page at:

http://www.oreilly.com/catalog/javabeans/

For more information about this book and others, see the O'Reilly web site:

http://www.oreilly.com

Providing Feedback to the Author

I've tried to be accurate and complete in my description of JavaBeans, but it's inevitable that there will be errors and omissions. If you find any mistakes, if you think I've left something out, or if you have any suggestions for a future edition of this book, please let me know. In addition to contacting O'Reilly, you may contact me directly at rob@mindstrm.com. The JavaBeans architecture will continue to change over time, and this book will certainly attempt to keep up with those changes.

Retrieving Examples Online

Much of the code in this book is available for download. On the World Wide Web, go to http://www.oreilly.com/catalog/javabeans/. You can also retrieve the files via FTP (either with your web browser or another FTP client) at ftp://ftp.oreilly.com/published/oreilly/java/javabeans/.

Chapter 1. Introduction

As software developers, we are constantly being asked to build applications in less time and with less money. And, of course, these applications are expected to be better and faster than ever before. Object-oriented techniques and component software environments are in wide use now, in the hope

that they can help us build applications more quickly. Development tools like Microsoft's Visual Basic have made it easier to build applications faster by taking a building-block approach to software development. Such tools provide a visual programming model that allows you to include software components rapidly in your applications.

The JavaBeans architecture brings the component development model to Java, and that's the subject of this book. But before we get started, I want to spend a little time describing the component model, and follow that with a general overview of JavaBeans. If you already have an understanding of these subjects, or you just want to get right into it, you can go directly to Chapter 2. Otherwise, you'll probably find that the information in this chapter sets the stage for the rest of the book.

1.1 The Component Model

Components are self-contained elements of software that can be controlled dynamically and assembled to form applications. But that's not the end of it. These components must also interoperate according to a set of rules and guidelines. They must behave in ways that are expected. It's like a society of software citizens. The citizens (components) bring functionality, while the society (environment) brings structure and order.

JavaBeans is Java's component model. It allows users to construct applications by piecing components together either programmatically or visually (or both). Support of visual programming is paramount to the component model; it's what makes component-based software development truly powerful.

The model is made up of an architecture and an API (Application Programming Interface). Together, these elements provide a structure whereby components can be combined to create an application. This environment provides services and rules, the framework that allows components to participate properly. This means that components are provided with the tools necessary to work in the environment, and they exhibit certain behaviors that identify them as such. One very important aspect of this structure is containment. A container provides a context in which components can interact. A common example would be a panel that provides layout management or mediation of interactions for visual components. Of course, containers themselves can be components.

As mentioned previously, components are expected to exhibit certain behaviors and characteristics in order to participate in the component structure and to interact with the environment, as well as with other components. In other words, there are a number of elements that, when combined, define the component model. These are described in more detail in the following sections.

1.1.1 Discovery and Registration

Class and interface discovery is the mechanism used to locate a component at run-time and to determine its supported interfaces so that these interfaces can be used by others. The component model must also provide a registration process for a component to make itself and its interfaces known. The component, along with its supported interfaces, can then be discovered at run-time. Dynamic (or late) binding allows components and applications to be developed independently. The dependency is limited to the "contract" between each component and the applications that use it; this contract is defined by interfaces that the component supports. An application does not have to include a component during the development process in order to use it at run-time; it only needs to know what the component is capable of doing. Dynamic discovery also allows developers to update components without having to rebuild the applications that use them.

This discovery process can also be used in a design-time environment. In this case, a development tool may be able to locate a component and make it available for use by the designer. This is important for visual programming environments, which are discussed later.

1.1.2 Raising and Handling of Events

An event is something of importance that happens at a specific point in time. An event can take place due to a user action such as a mouse click?when the user clicks a mouse button, an event takes place. Events can also be initiated by other means. Imagine the heating system in your house. It contains a thermostat that sets the desired comfort temperature, keeps track of the current ambient temperature, and notifies the boiler when its services are required. If the thermostat is set to keep the room at 70 degrees Fahrenheit, it will notify the boiler to start producing heat if the temperature dips below that threshold. Components will send notifications to other objects when an event takes place in which those objects have expressed an interest.

1.1.3 Persistence

Generally, all components have state. The thermostat component has state that represents the comfort temperature. If the thermostat were a software component of a computer-based heating control system, we would want the value of the comfort temperature to be stored on a non-volatile storage medium (such as the hard disk). This way if we shut down the application and brought it back up again, the thermostat control would still be set to 70 degrees. The visual representation and position of the thermostat relative to other components in the application would be restored as well.

Components must be able to participate in their container's persistence mechanism so that all components in the application can provide application-wide persistence in a uniform way. If every component were to implement its own method of persistence, it would be impossible for an application container to use components in a general way. This wouldn't be an issue if reuse weren't the goal. If we were building a monolithic temperature control system we might create an application-specific mechanism for storing state. But we want to build the thermostat component so that it can be used again in another application, so we have to use a standard mechanism for persistence.

1.1.4 Visual Presentation

The component environment allows the individual components to control most of the aspects of their visual presentation. For example, imagine that our thermostat component includes a display of the current ambient temperature. We might want to display the temperature in different fonts or colors depending on whether we are above, below, or at the comfort temperature. The component is free to choose the characteristics of its own visual presentation. Many of these characteristics will be properties of the component (a topic that will be discussed later). Some of these visual properties will be persistent, meaning that they represent some state of the control that will be saved to, and restored from, persistent storage.

Layout is another important aspect of visual presentation. This concerns the way in which components are arranged on the screen, how they relate to one another, and the behavior they exhibit when the user interacts with them. The container object that holds an assembly of components usually provides some set of services related to the layout of the component. Let's consider the thermostat and heating control application again. This time, the user decides to change the size of the application window. The container will interact with the components in response to this action, possibly changing the size of some of the components. In turn, changing the size of the thermostat component may cause it to alter its font size.

As you can see, the container and the component work together to provide a single application that presents itself in a uniform fashion. The application appears to be working as one unit, even though with the component development model, the container and the components probably have been created separately by different developers.

1.1.5 Support of Visual Programming

Visual programming is a key part of the component model. Components are represented in toolboxes or palettes. The user can select a component from the toolbox and place it into a container, choosing its size and position. The properties of the component can then be edited in order to create the desired behavior. Our thermostat control might present some type of user interface to the application developer to set the initial comfort temperature. Likewise, the choice of font and color will be selectable in a similar way. None of these manipulations require a single line of code to be written by the application developer. In fact, the application development tool is probably writing the code for you. This is accomplished through a set of standard interfaces provided by the component environment that allow the components to publish, or expose, their properties. The development tool can also provide a means for the developer to manipulate the size and position of components in relation to each other. The container itself may be a component and allow its properties to be edited in order to alter its behavior.

1.2 The JavaBeans Architecture

JavaBeans is an architecture for both using and building components in Java. This architecture supports the features of software reuse, component models, and object orientation. One of the most important features of JavaBeans is that it does not alter the existing Java language. If you know how to write software in Java, you know how to use and create Beans. The strengths of Java are built upon and extended to create the JavaBeans component architecture.

Although Beans are intended to work in a visual application development tool, they don't necessarily have a visual representation at run-time (although many will). What this does mean is that Beans must allow their property values to be changed through some type of visual interface, and their methods and events should be exposed so that the development tool can write code capable of manipulating the component when the application is executed.

Creating a Bean doesn't require any advanced concepts. So before I go any further, here is some code that implements a simple Bean:

```
public class MyBean implements java.io.Serializable
{
   protected  int theValue;

   public MyBean()
   {
   }

   public void setMyValue(int newValue)
   {
      theValue = newValue;
   }

   public int getMyValue()
   {
      return theValue;
   }
}
```

This is a real Bean named *MyBean* that has state (the variable `theValue`) that will automatically be saved and restored by the JavaBeans persistence mechanism, and it has a property named *MyValue* that is usable by a visual programming environment. This Bean doesn't have any visual representation, but that isn't a requirement for a JavaBean component.

JavaSoft is using the slogan "Write once, use everywhere." Of course "everywhere" means everywhere the Java run-time environment is available. But this is very important. What it means is that the entire run-time environment required by JavaBeans is part of the Java platform. No special libraries or classes have to be distributed with your components. The JavaBeans class libraries provide a rich set of default behaviors for simple components (such as the one shown earlier). This means that you don't have to spend your time building a lot of support for the Beans environment into your code.

The design goals of JavaBeans are discussed in Sun's white paper, "Java Beans: A Component Architecture for Java." This paper can be found on the JavaSoft web site at http://splash.javasoft.com/beans/WhitePaper.html. It might be interesting to review these goals before we move on to the technology itself, to provide a little insight into why certain aspects of JavaBeans are the way they are.

1.2.1 Compact and Easy

JavaBeans components are simple to create and easy to use. This is an important goal of the JavaBeans architecture. It doesn't take very much to write a simple Bean, and such a Bean is lightweight?it doesn't have to carry around a lot of inherited baggage just to support the Beans environment. If a Bean does not require the advanced features of the architecture, it doesn't get them, nor does it get the code that goes with them. This is an important concept. The JavaBeans architecture scales upward in complexity, not downward like other component models. This means it really is easy to create a simple Bean. (The previous example shows just how simple a Bean can be.)

1.2.2 Portable

Since JavaBeans components are built purely in Java, they are fully portable to any platform that supports the Java run-time environment. All platform specifics, as well as support for JavaBeans, are implemented by the Java virtual machine. You can be sure that when you develop a component using JavaBeans it will be usable on all of the platforms that support Java (version 1.1 and beyond). These range from workstation applications and web browsers to servers, and even to devices such as PDAs and set-top boxes.

1.2.3 Leverages the Strengths of the Java Platform

JavaBeans uses the existing Java class discovery mechanism. This means that there isn't some new complicated mechanism for registering components with the run-time system.

As shown in the earlier code example, Beans are lightweight components that are easy to understand. Building a Bean doesn't require the use of complex extensions to the environment. Many of the Java supporting classes are Beans, such as the windowing components found in `java.awt`.

The Java class libraries provide a rich set of default behaviors for components. Use of Java Object Serialization is one example?a component can support the persistence model by implementing the `java.io.Serializable` interface. By conforming to a simple set of design patterns (discussed later

in this chapter), you can expose properties without doing anything more than coding them in a particular style.

1.2.4 Flexible Build-Time Component Editors

Developers are free to create their own custom property sheets and editors for use with their components if the defaults aren't appropriate for a particular component. It's possible to create elaborate property editors for changing the value of specific properties, as well as create sophisticated property sheets to house those editors.

Imagine that you have created a `Sound` class that is capable of playing various sound format files. You could create a custom property editor for this class that listed all of the known system sounds in a list. If you have created a specialized color type called `PrimaryColor`, you could create a color picker class to be used as the property editor for `PrimaryColor` that presented only primary colors as choices.

The JavaBeans architecture also allows you to associate a custom editor with your component. If the task of setting the property values and behaviors of your component is complicated, it may be useful to create a component wizard that guides the user through the steps. The size and complexity of your component editor is entirely up to you.

1.3 JavaBeans Overview

The JavaBeans white paper defines a Bean as follows:

A Java Bean is a reusable software component that can be manipulated visually in a builder tool.

Well, if you have to sum it up in one sentence, this is as good as any. But it's pretty difficult to sum up an entire component architecture in one sentence. Beans will range greatly in their features and capabilities. Some will be very simple and others complex; some will have a visual aspect and others won't. Therefore, it isn't easy to put all Beans into a single category. Let's take a look at some of the most important features and issues surrounding Beans. This should set the stage for the rest of the book, where we will examine the JavaBeans technology in depth.

1.3.1 Properties, Methods, and Events

Properties are attributes of a Bean that are referenced by name. These properties are usually read and written by calling methods on the Bean specifically created for that purpose. A property of the thermostat component mentioned earlier in the chapter could be the comfort temperature. A programmer would set or get the value of this property through method calls, while an application developer using a visual development tool would manipulate the value of this property using a visual property editor.

The methods of a Bean are just the Java methods exposed by the class that implements the Bean. These methods represent the interface used to access and manipulate the component. Usually, the set of public methods defined by the class will map directly to the supported methods for the Bean, although the Bean developer can choose to expose only a subset of the public methods.

Events are the mechanism used by one component to send notifications to another. One component can register its interest in the events generated by another. Whenever the event occurs, the interested component will be notified by having one of its methods invoked. The process of registering interest in an event is carried out simply by calling the appropriate method on the

component that is the source of the event. In turn, when an event occurs a method will be invoked on the component that registered its interest. In most cases, more than one component can register for event notifications from a single source. The component that is interested in event notifications is said to be *listening* for the event.

1.3.2 Introspection

Introspection is the process of exposing the properties, methods, and events that a JavaBean component supports. This process is used at run-time, as well as by a visual development tool at design-time. The default behavior of this process allows for the automatic introspection of any Bean. A low-level reflection mechanism is used to analyze the Bean's class to determine its methods. Next it applies some simple design patterns to determine the properties and events that are supported. To take advantage of reflection, you only need to follow a coding style that matches the design pattern. This is an important feature of JavaBeans. It means that you don't have to do anything more than code your methods using a simple convention. If you do, your Beans will automatically support introspection without you having to write any extra code. Design patterns are explained in more detail later in the chapter.

This technique may not be sufficient or suitable for every Bean. Instead, you can choose to implement a BeanInfo class which provides descriptive information about its associated Bean explicitly. This is obviously more work than using the default behavior, but it might be necessary to describe a complex Bean properly. It is important to note that the BeanInfo class is separate from the Bean that it is describing. This is done so that it is not necessary to carry the baggage of the BeanInfo within the Bean itself.

If you're writing a development tool, an `Introspector` class is provided as part of the Beans class library. You don't have to write the code to accomplish the analysis, and every tool vendor uses the same technique to analyze a Bean. This is important to us as programmers because we want to be able to choose our development tools and know that the properties, methods, and events that are exposed for a given component will always be the same.

1.3.3 Customization

When you are using a visual development tool to assemble components into applications, you will be presented with some sort of user interface for customizing Bean attributes. These attributes may affect the way the Bean operates or the way it looks on the screen. The application tool you use will be able to determine the properties that a Bean supports and build a property sheet dynamically. This property sheet will contain editors for each of the properties supported by the Bean, which you can use to customize the Bean to your liking. The Beans class library comes with a number of property editors for common types such as `float`, `boolean`, and `String`. If you are using custom classes for properties, you will have to create custom property editors to associate with them.

In some cases the default property sheet that is created by the development tool will not be good enough. You may be working with a Bean that is just too complex to customize easily using the default sheet. Beans developers have the option of creating a customizer that can help the user to customize an instance of their Bean. You can even create smart wizards that guide the user through the customization process.

Customizers are also kept separate from the Bean class so that it is not a burden to the Bean when it is not being customized. This idea of separation is a common theme in the JavaBeans architecture. A Bean class only has to implement the functionality it was designed for; all other supporting features are implemented separately.

1.3.4 Persistence

It is necessary that Beans support a large variety of storage mechanisms. This way, Beans can participate in the largest number of applications. The simplest way to support persistence is to take advantage of Java Object Serialization. This is an automatic mechanism for saving and restoring the state of an object. Java Object Serialization is the best way to make sure that your Beans are fully portable, because you take advantage of a standard feature supported by the core Java platform. This, however, is not always desirable. There may be cases where you want your Bean to use other file formats or mechanisms to save and restore state. In the future, JavaBeans will support an alternative externalization mechanism that will allow the Bean to have complete control of its persistence mechanism.

1.3.5 Design-Time vs. Run-Time

JavaBeans components must be able to operate properly in a running application as well as inside an application development environment. At design-time the component must provide the design information necessary to edit its properties and customize its behavior. It also has to expose its methods and events so that the design tool can write code that interacts with the Bean at run-time. And, of course, the Bean must support the run-time environment.

1.3.6 Visibility

There is no requirement that a Bean be visible at run-time. It is perfectly reasonable for a Bean to perform some function that does not require it to present an interface to the user; the Bean may be controlling access to a specific device or data feed. However, it is still necessary for this type of component to support the visual application builder. The component can have properties, methods, and events, have persistent state, and interact with other Beans in a larger application. An "invisible" run-time Bean may be shown visually in the application development tool, and may provide custom property editors and customizers.

1.3.7 Multithreading

The issue of multithreading is no different in JavaBeans than it is in conventional Java programming. The JavaBeans architecture doesn't introduce any new language constructs or classes to deal with threading. You have to assume that your code will be used in a multithreaded application. It is your responsibility to make sure your Beans are thread-safe. Java makes this easier than in most languages, but it still requires some careful planning to get it right. Remember, thread-safe means that your Bean has anticipated its use by more than one thread at a time and has handled the situation properly.

1.3.8 Security

Beans are subjected to the same security model as standard Java programs. You should assume that your Bean is running in an untrusted applet. You shouldn't make any design decisions that require your Bean to be run in a trusted environment. Your Bean may be downloaded from the World Wide Web into your browser as part of someone else's applet. All of the security restrictions apply to Beans, such as denying access to the local file system, and limiting socket connections to the host system from which the applet was downloaded.

If your Bean is intended to run only in a Java application on a single computer, the Java security constraints do not apply. In this case you might allow your Bean to behave differently. Be careful,

because the assumptions you make about security could render your Bean useless in a networked environment.

1.4 Using Design Patterns

The JavaBeans architecture makes use of patterns that represent standard conventions for names, and type signatures for collections of methods and interfaces. Using coding standards is always a good idea because it makes your code easier to understand, and therefore easier to maintain. It also makes it easier for another programmer to understand the purpose of the methods and interfaces used by your component. In the JavaBeans architecture, these patterns have even more significance. A set of simple patterns are used by the default introspection mechanism to analyze your Bean and determine the properties, methods, and events that are supported. These patterns allow the visual development tools to analyze your Bean and use it in the application being created. The following code fragment shows one such pattern:

```
public void setTemperatureColor(Color newColor)
{
   . . .
}

public Color getTemperatureColor()
{
   . . .
}
```

These two methods together use a pattern that signifies that the Bean contains a property named *TemperatureColor* of type `Color`. No extra development is required to expose the property. The various patterns that apply to Beans development will be pointed out and discussed throughout this book. I'll identify each pattern where the associated topic is being discussed.

The use of the term "design pattern" here may be confusing to some readers. This term is commonly used to describe the practice of documenting a reusable design in object-oriented software. This is not entirely different than the application of patterns here. In this case, the design of the component adheres to a particular convention, and this convention is reused to solve a particular problem.

As mentioned earlier, this convention is not a requirement. You can implement a specific BeanInfo class that fully describes the properties, methods, and events supported by your Bean. In this case, you can name your methods anything you please.

1.5 JavaBeans vs. ActiveX

JavaBeans is certainly not the first component architecture to come along. Microsoft's ActiveX technology is based upon COM, their component object model. ActiveX offers an alternative component architecture for software targeted at the various Windows platforms. So how do you choose one of these technologies over the other? Organizational, cultural, and technical issues all come into play when making this decision. ActiveX and JavaBeans are not mutually exclusive of each other?Microsoft has embraced Java technology with products like Internet Explorer and Visual J++, and Sun seems to have recognized that the desktop is dominated by Windows and has targeted Win32 as a strategic platform for Java. It is not in anyone's best interest to choose one technology to the exclusion of another. Both are powerful component technologies. I think we should choose a technology because it supports the work we are doing, and does so in a way that meets the needs of the customer.

The most important question is how Beans will be used by containers that are designed specifically to contain ActiveX controls. Certainly, all Beans will not also be ActiveX controls by default. To address the need to integrate Beans into the world of ActiveX, an ActiveX Bridge is available that maps the properties, methods, and events exposed by the Bean into the corresponding mechanisms in COM. This topic is covered in detail in Chapter 11.

1.6 Getting Started

If you plan to play along, you should make sure that you have installed the latest versions of the Java Development Kit (JDK) and the Beans Development Kit (BDK). Both of these can be downloaded from the JavaSoft web site at http://java.sun.com/.[1]

[1] Beans development requires JDK1.1; however, JDK1.1.1 is now available.

Remember that if you don't have a browser that supports JDK1.1, you will have to run your applets in the *appletviewer* program that is provided in the JDK. At the time of this writing, the only browser that supports JDK1.1 is HotJava.

The chapters in this book are arranged so that they build on concepts presented in preceding chapters. I suggest that you try to follow along in order. Of course, this is entirely up to you. If you are comfortable with the technology, you may find that you can jump around a bit.

Chapter 2. Events

Events are messages sent from one object to another, notifying the recipient that something interesting has happened. The component sending the event is said to *fire* the event; the recipient is called a *listener*, and is said to *handle* the event. Many objects can listen for a particular event; thus, components must be able to fire an event to an arbitrary number of objects. (Java allows you to define events that can have at most one listener, but outside of this special case, you must assume that there can be many listeners.) Likewise, an object is free to listen for as many different events as it desires.

The process of firing and handling events is a common feature of windowing systems. Many programmers learned how to write software using this mechanism when they began writing code for Microsoft Windows or the X Windows system. Now the techniques of object-oriented programming are showing us that the event model can be used for applications other than windowing. Firing and handling events is one of two ways that objects communicate with each other. The other is by invoking methods on each other. We will see shortly that these two mechanisms can be one and the same.

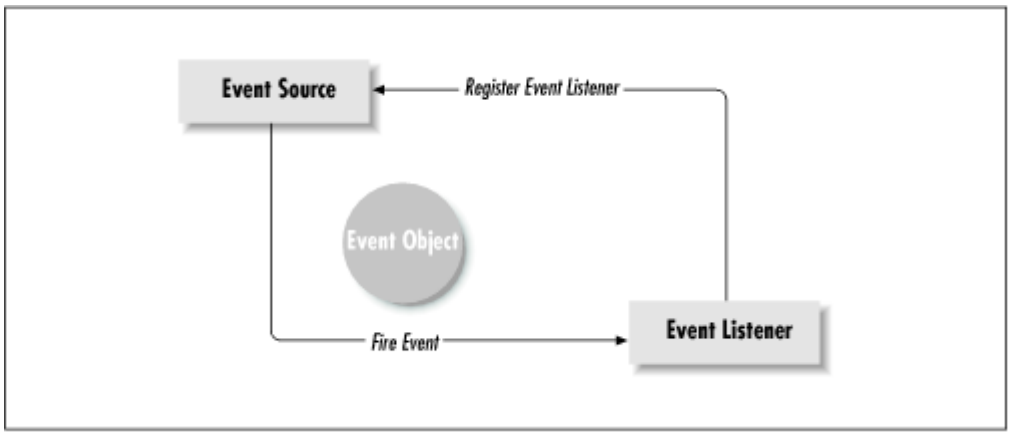
2.1 The Java Event Model

The JavaBeans architecture takes advantage of the event model that was introduced in version 1.1 of the Java language. Beans are treated the same as every other object within the event model. In fact, there is nothing at all about the event model that is specific to Beans. This is a common theme in JavaBeans. The Beans component model is designed to take advantage of features that are already available in Java. This is not to say that some of those features were not designed with Beans in mind, because apparently they were. But in many cases the features that are needed by Beans are generic enough to be provided by a core Java class library, making them available to any Java class whether or not it happens to be a Bean.

An event source object sends a notification that an event occurred by invoking a method on the target, or "listening," object. The notification mechanism uses standard Java methods. There is no new language construct or programming syntax required to create such a method. Every event has a specific notification method associated with it. When the event occurs, the associated method is invoked on every object that is listening for it. The method itself describes to the caller which event has taken place. When this method is called, an object is passed as an argument that contains specific information about that particular instance of the event. This object always contains a reference to the object that was the source of the event: it allows the object receiving the event to identify the sender, and is particularly important if an object listens for the same event from several sources. Without this reference, we would have no way of determining which object sent the event. The combination of the method called and the data sent completely defines the event.

The Java event model is comprised of *event objects*, *event listeners*, and *event sources*. These objects interoperate in a standard way, using method invocation to facilitate firing and handling events. Figure 2.1 is a simple diagram of this interaction. The event listener registers itself with the event source to receive event notifications. At some point the event source fires an event with the event object as a parameter, and the event listener handles the event.

Figure 2.1. Event sources and listeners



The Java package `java.util` provides basic support for the event model. It provides a base class for event objects, as well as a base interface for event listeners. All of the classes and interfaces in the Java class libraries that use the event model make use of these classes. When you are developing Beans, or any other Java class that leverages the event model, you will be working directly with these classes and interfaces or their subclasses.

2.1.1 Event Objects

An event object encapsulates the information that is specific to an instance of an event. For example, an event that represents a mouse click might contain the position of the mouse pointer, and possibly a way of indicating which mouse button was used. This object is passed as the parameter to an event notification method.

The class `java.util.EventObject` is used as the base class for all event objects. It contains a reference to the object that generated the event, along with a method for retrieving that object. It can also render itself as a human-readable string. Like many Java classes, it does this by overriding the `toString()` method. Table 2.1 shows the public methods provided by the `java.util.EventObject` class.

Table 2.1, Public Methods of EventObject

Method	Description
`public EventObject(Object source)`	The constructor for the event. It takes the event source object as a parameter.
`public Object getSource()`	Returns the object that generated the event.
`public String toString()`	Renders the event object in human-readable form.

Whenever you define an event object for passing information related to a particular event, you should create a subclass of `java.util.EventObject`. By convention, these classes should end in the word *Event*. This helps to quickly identify their purpose to programmers. All of the event subclasses provided by the core Java class libraries follow this convention. Later in this chapter we'll look at some of the classes and interfaces used by the `java.awt.event` package for dealing with events. There you'll notice that this convention is followed, as well as with others that I'll be mentioning.

You are free to pass instances of `java.util.EventObject` when notifying listener objects of events, but you'll probably want to create event-specific subclasses for this purpose. Subclassing allows you to evolve the object over time without redefining any method signatures that take the base class as an argument; it's also useful to ensure that only the desired event type is passed as a parameter during event firing. Equally important is that the event object is part of what defines the event. If you were to use `java.util.EventObject` without subclassing, you would be allowing any (in fact, every) event type to be passed to the event's recipients. This means we could pass a mouse event object when a keyboard event occurs. This is most definitely not a good design practice. You should always subclass `java.util.EventObject` even if you don't need to provide any additional information in the event object. If you want to create a new event type without providing any additional information, your code might look something like this:

```
public class SimpleEvent extends java.util.EventObject
{
   // construct the simple event type
   SimpleEvent(Object source)
   {
      super(source);
   }
}
```

In most cases you will need to extend the basic `java.util.EventObject` class in order to fully describe the event. As we'll see later, the event notification method has no way of distinguishing between event instances. It's up to the event object to provide the information that uniquely describes a specific instance of an event. The subclass encapsulates the information related to the event and provides any methods required to access and manipulate that data.

Before creating a realistic event class, a word about the examples in this book. I use a set of classes and interfaces to illustrate the concepts throughout each chapter. As new topics are introduced we'll build on our work. We'll be developing a package for a temperature control simulator. The package is named `BeansBook.Simulator`, and it will include classes for things like thermostats, boilers, and air conditioners. Any useful examples that are not specific to the simulator will be put into another package called `BeansBook.util`. Although many of the examples are contrived, they make use of all of the features of Beans development. So by the time we're done, you will have seen all of the concepts applied in working code.

The simulator is driven by changes in temperature. As the ambient temperature in our simulated system changes, various objects are going to react. Immediately we see a need for an event that