5

Persisting Your Application Using Files

The previous chapter discussed building user interfaces and stand-alone Java applications using the Java Foundation Classes (JFC). A key feature of many applications is the ability to save its state off to a file. Image manipulation programs need to read and write images to disk. Word processing and other office-productivity applications need to read and write spreadsheets, presentations, and text-based documents. Essentially, to do any of these save operations, an application must take its in-memory representation of its state, and write it to disk. Later on, this file can be read back into memory, putting the application back to exactly where the user had left using it.

Different applications need to save different pieces of information to disk. Some applications really only need to save their configuration to disk, because they may save their other data to a database (the subsequent chapter shows you how to persist your application's data to a database). A typical single-user application such as a word processor or image manipulation program will need to save its state to files. Java provides a couple of built-in mechanisms for saving or serializing data to disk. The two major APIs in the JDK for persisting application data to disk are the Java Serialization API for generic serialization and the XMLEncoder/Decoder API for serializing JavaBean components.

This chapter looks at the Java Serialization API, the XMLEncoder/Decoder API, and the Java API for XML Binding (JAXB). JAXB provides mechanisms to read and write to user-defined XML file formats. Each of these three APIs has different approaches to serialization and as such should be used in different circumstances. This chapter looks at the Serialization API first, followed by the XMLEncoder/Decoder API, and finishes with JAXB.

Application Data

Every application has some sort of in-memory data structure from which to retrieve its data. Besides data structures like maps, lists, sets, and trees, custom data structures are often built. For an application to save its data, the data in these structures must be saved to disk, and then at a later time, loaded back into the same data structure. Web browsers, for example, create what's called a Document Object Model (DOM) in memory for every web page that is loaded. It is their internal data structure for displaying HTML pages. Word processors also keep some sort of document object model as well — some way to represent the fact that certain pieces of text are aligned to the right, or possibly that other paragraphs of text are highlighted in a particular color. These custom data structures are necessary for the application to display the data properly to the user.

Applications like web browsers essentially read files and display them to the user. A web browser first reads HTML files over a network or from a disk, and parses the data into its internal in-memory data, the DOM. Now that the data is in the web browser's data structure, its functions can properly display the page to the user. Image viewing programs are similar; they read an image into their internal data structure representing images, and then display that image to the user. Other types of applications, though, also allow the user to manipulate the data. Word processors, in addition to reading files into their internal data structures and displaying them, also must allow the user to manipulate the data, and therefore the internal data structure, and then write it back to disk.

Many of these other applications that must allow the user to manipulate data follow the Model-View-Controller (MVC) design pattern (see Chapter 3 more for information on design patterns). The internal data structures of the application are its *data model*. This data model is contained in structures that are separate from UI components and UI-related data structures. In Java-based applications, the data model usually consists of JavaBeans, along with other data storage and collection classes. These data classes are manipulated and modified by UI controller classes (such as events generated by buttons, menus, and so forth), and presented in a view by other UI components. A simple MVC diagram is shown in Figure 5-1, illustrating how only the data model of an MVC-based application needs to be saved to restore the state of the application. Swing or other UI toolkit/utility classes would be in both the view and controller areas, whereas the internal data model specific to the domain of the application would be contained in the data model. This step of separating domain data from UI components simplifies the process of saving and loading the data from disk because the data is all in one place, the model.

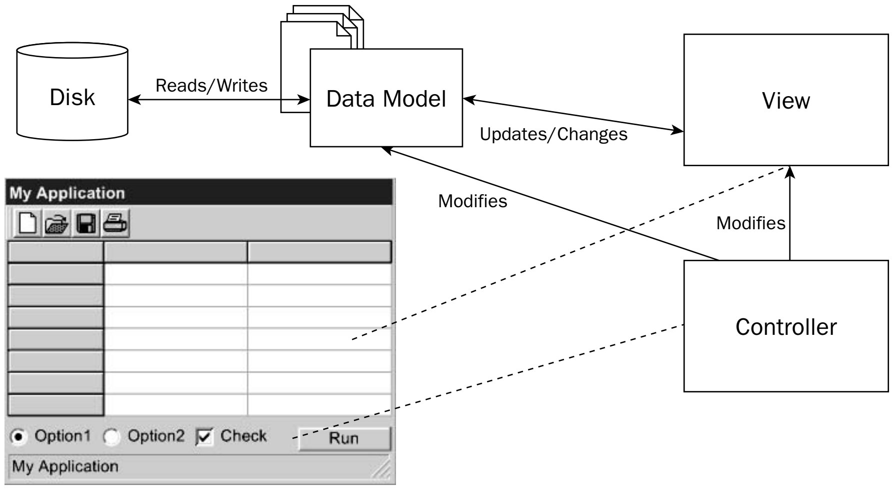


Figure 5-1

Once all the domain data is contained in its own model, separate from the UI components, the parts of the data model that need to be persisted can be identified. Some pieces of an internal data structure need not necessarily be saved. Some parts of the data structure in an application will not change from time to time, or can be re-created given that certain other aspects of the data structure exist. Developers wishing to save the state of their application must look carefully at the data they hold in memory in their model, identify the pieces that must be saved, and then write routines for saving and loading the data from the data structure to and from disk.

Saving Application Data

Now that application data structures have been discussed in a general sense, it is time to move to something a little more tangible and realistic. How exactly do Java applications store their data model in memory? Because Java is an object-oriented language, most applications have a set of data classes (which is the application's data model). Instances of these data classes reside in memory and the viewer and controller components (the UI) of the application interact with them to produce the functionality of the application.

Any Java class that has attributes (or *properties* in JavaBean terms) is a data structure. A simple data structure could be a `Person` class with two `String` attributes, for first name and last name. More complex classes that contain references to other classes, effectively form an *object graph*. An object graph is a graph where objects are the nodes in the graph and the connections are references from one instance of an object to another. The notion of object graphs is important because when you want to serialize the information contained in a class, you must also consider what data the class relies on in *other* classes and their dependencies and so on. The next section outlines a sample data model for a generic application's configuration, and you will view an example object graph.

Sample Configuration Data Model for an Application

Throughout this chapter you will be using a sample application and persisting its configuration using Java Serialization, the XMLEncoder/Decoder APIs, and JAXB. The application is fairly generic (and many applications could have similar configurations). Think of this example application as some sort of image editing/drawing program that includes a canvas with tool and palette windows. Different users of the application will undoubtedly have different preferences for some of the settings the application supports. At a high level, this user-preference or configuration information you want to persist includes the following:

- Location of the user's home directory or default directory to load and save files
- A list of recent files loaded or saved by the user
- Whether or not the application should use a tabbed windowed interface or a multiple document interface (MDI) with child windows
- Foreground and background colors last used (for drawing or painting operations)
- The positions of the tool and palette windows within the application when the application was last closed

In a full-fledged paint or photo editing application, there would probably be many more configuration options that users could potentially persist to a file. However, the process is the same, and can also be applied to how to save application data such as a custom image format, or reading and writing other image formats into your application's structure. Persisting information in Java objects to the file system is the same whether it is application configuration data or simply application data itself. Figure 5-2 shows the data model structure, and Figure 5-3 shows an example object graph of an instance of the data model.

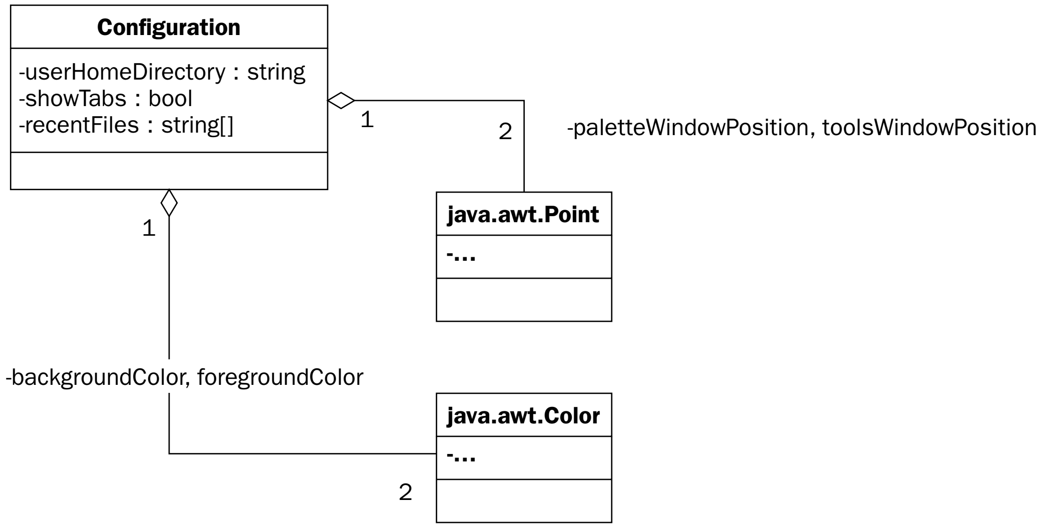

Configuration
-userHomeDirectory : string
-showTabs : bool
-recentFiles : string[]
1
2
-paletteWindowPosition, toolsWindowPosition
java.awt.Point
-...
1
-backgroundColor, foregroundColor
java.awt.Color
-...
2

Figure 5-2

`Configuration` is the root object. It uses classes from `java.awt` to represent colors and points. In the object graph shown in Figure 5-3, you can see that an instance of configuration also contains references to instances of `java.awt.Color` and `java.awt.Point`. When you persist the information in a `Configuration` instance to disk, you must also save the information contained in the `Color` and `Point` instances (and any other class instances *they* may also reference), if you want to be able to re-create the `Configuration` object at a later point in time.

You will design `Configuration` using the JavaBeans architecture (`getXXX` and `setXXX` for all properties in your class). The application itself will read the configuration settings from this class and appropriately apply them throughout the application. It is typical to use JavaBeans conventions to store data in Java-based data models. Using the JavaBeans standard allows the designer to use many tools that are based on those standards such as XMLEncoder/Decoder. Other tools that utilize JavaBeans conventions are object-relational-mapping tools, which allow the developer to map Java objects to a database. Most of Java's third-party serialization tools require classes to use JavaBeans conventions. It is just good practice and design.

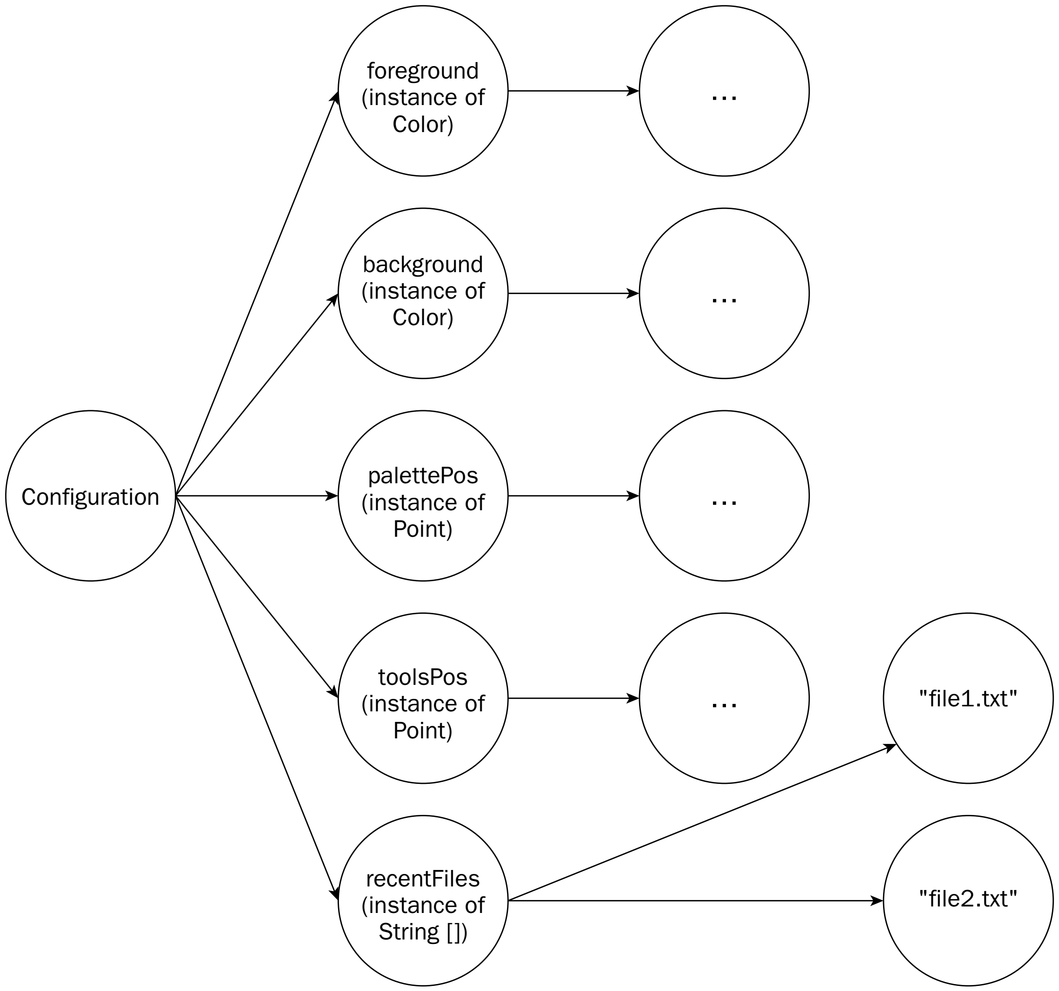

Configuration
foreground (instance of Color)
background (instance of Color)
palettePos (instance of Point)
toolsPos (instance of Point)
recentFiles (instance of String [])
...
...
...
...
"file1.txt"
"file2.txt"

Figure 5-3

Java Serialization: Persisting Object Graphs

The Java Serialization API takes one approach to saving a data model. It writes all of the object instances in the data model's graph to disk. To reconstruct the saved object graph, it reads the saved class instances from disk back into memory. *Serializing* an object instance is the process of writing its data members to disk. *Deserializing* an object instance is the process of reconstructing the instance from the data members written to disk. Suppose you have a simple class `MyPoint`:

```
package book;

public class MyPoint {
  public int x;
```

```
    public int y;

    public void doSomething() { ... }
  }
```

To save an instance of `MyPoint` to disk, you simply need to write its two data members to disk. Saving `x` and `y` allows you to create a new instance of `MyPoint` at a later time and set its `x` and `y` values to the ones saved to disk — effectively re-creating the original instance. The method `doSomething()` is already specified in the compiled class file, and there is no need to store any method information in the serialization process. All a class instance is in memory is the values for all of its attributes. To serialize an instance to disk, all of its data members must be saved. What if the data member is a reference to another object instance? The reference itself is just a memory address and would be meaningless to save. The object instance the reference points to also would need to be saved as well. Suppose you add a color attribute to `MyPoint`:

```
  package book;

  import java.awt.Color;

  public class MyPoint {
    public int x;
    public int y;

    private Color pointColor;

    public void doSomething() { ... }
  }
```

The data members of the instance of `java.awt.Color` must now also be saved. As you can see, the entire object graph of an object instance must be saved when it is serialized to disk. If you only saved `x` and `y` from `MyPoint` and then subsequently re-created `MyPoint` at a later time, its color information would be lost. So how is an external API able to access all of the fields of a particular class? Java's reflection mechanism allows the dynamic ability to find out the fields and field values of any class, whether those fields are marked `public` or `private`. Thankfully, the Java Serialization API takes care of all these details for you, and it is easy to serialize object instances to disk.

It is important to note that the file format used by the Java Serialization API is a special binary file format developed specifically for Java Serialization and therefore not human-readable. It is an efficient format, but also specific to Java.

Key Classes

The Java Serialization API hides most of the complexity required to save object graphs to disk (such as circular references and multiple references to the same object). There are really only two interfaces and two classes that need to be learned in order to use the API. `ObjectInputStream` and `ObjectOutputStream` are two stream classes that can be wrapped around any type of `java.io.InputStream` or `java.io.OutputStream`, respectively, making it possible to send serialized objects over the network or simply save them to disk. The two interfaces, `Serializable` and `Externalizable`, allow for implementing classes to be serialized. If a class does not implement one of these two interfaces, it cannot be serialized using the

API. This means that if a class that *does* implement either `Serializable` or `Externalizable` contains a reference to a class that *does not* implement that interface somewhere in its object graph, it cannot be serialized successfully without some modification (this is discussed later on in this chapter).

Following is a table of the Serializable and Externalizable classes:

Class or Interface (from java.io)	Function
`Serializable`	Interface for marking the fact that a class supports serialization
`ObjectInputStream`	Input stream used to read object instances that were written by an `ObjectOutputStream`
`ObjectOutputStream`	Output stream used to write object instance data that can later be read by an `ObjectInputStream`
`Externalizable`	Interface that extends `Serializable` to give a class complete control over how it is read and written to streams

Serializing Your Objects

Performing the actual serialization of objects is straightforward. There are four main steps:

1. Make sure your class has a default constructor (takes no arguments).
2. Implement the `Serializable` or `Externalizable` interface to mark your class as supporting serialization.
3. Use `ObjectOutputStream` to serialize your object.
4. Use `ObjectInputStream` to read a serialized object back into memory.

Classes you wish to serialize must have default constructors. This is because the serialization API needs to create blank instances of the class when it re-creates object instances saved to disk — it does so by calling the default constructor. After it creates the new class it populates the data members of the class via reflection (meaning accessor and mutator methods are not required for private data members). The class must also be marked as serializable by implementing the `Serializable` interface. The `Serializable` interface contains no method definitions; it is simply a marker to the serialization API to indicate that the class is indeed serializable. Not all classes store their data — for example, the classic example is a `java.sql.ResultSet` object, which is used in the Java DataBase Connectivity API (JDBC) to access data from a database. The `ResultSet` object is querying the database for data when its methods are called and hence does not store the information it returns. Because it is a mediator between the client and the database, it has no information to serialize! It would be incorrect to serialize an instance of `ResultSet` and expect to later on deserialize it and access the results of a database query, because the query results exist in the database and not in the `ResultSet`. The `Serializable` interface exists to give developers the ability to mark certain classes as potentially serializable — essentially meaning the author of a particular class planned for the fact that his class may be saved to disk. The `Externalizable` interface gives developers more control over the actual serialization process and is discussed in more detail later on in this chapter.

Configuration Example: Saving Your App's Configuration to Disk

Earlier in this chapter, you developed the high-level data model for a sample configuration for a generic image manipulation application. Suppose now you want to develop that data model and the UI components to save and read it from disk. The first step is translating your data model into code. You will have one class, `Configuration`, represent the application's configuration. You will model it using the JavaBeans conventions, implicitly provide it a default constructor (by having no constructors), and implement the `Serializable` interface. The two classes referenced in `Configuration`, `java.awt.Point` and `java.awt.Color`, also both implement `Serializable`, so the entire graph is guaranteed to serialize. The code for `Configuration` is as follows:

```
package book;

import java.awt.Color;
import java.awt.Point;
import java.io.Serializable;

public class Configuration implements Serializable {

  private String userHomeDirectory;

  private Color backgroundColor;
  private Color foregroundColor;

  private boolean showTabs;

  private Point paletteWindowPosition;
  private Point toolsWindowPosition;

  private String[] recentFiles;


  public Color getBackgroundColor() {
    return backgroundColor;
  }

  public void setBackgroundColor(Color backgroundColor) {
    this.backgroundColor = backgroundColor;
  }

  public Color getForegroundColor() {
    return foregroundColor;
  }

  public void setForegroundColor(Color foregroundColor) {
    this.foregroundColor = foregroundColor;
  }

  public Point getPaletteWindowPosition() {
    return paletteWindowPosition;
  }

  public void setPaletteWindowPosition(Point paletteWindowPosition) {
      this.paletteWindowPosition = paletteWindowPosition;
    }

    public String[] getRecentFiles() {
      return recentFiles;
    }

    public void setRecentFiles(String[] recentFiles) {
      this.recentFiles = recentFiles;
    }

    public boolean isShowTabs() {
      return showTabs;
    }

    public void setShowTabs(boolean showTabs) {
      this.showTabs = showTabs;
    }

    public Point getToolsWindowPosition() {
      return toolsWindowPosition;
    }

    public void setToolsWindowPosition(Point toolsWindowPosition) {
      this.toolsWindowPosition = toolsWindowPosition;
    }

    public String getUserHomeDirectory() {
      return userHomeDirectory;
    }

    public void setUserHomeDirectory(String userHomeDirectory) {
      this.userHomeDirectory = userHomeDirectory;
    }
  }
```

Writing the Configuration to Disk

Now that you have your configuration data model, you can write the code to serialize and deserialize instances of `Configuration`. Saving an instance of `Configuration` is almost too easy. First, create an `ObjectOutputStream` object, and because you want to save your instance of `Configuration` to a file, wrap it around a `FileOutputStream`:

```
  ObjectOutputStream out = new ObjectOutputStream(
                                    new FileOutputStream("appconfig.config"));
```

Now create an instance of `Configuration` and save it to the file `appconfig.config`:

```
  Configuration conf = new Configuration();
  // ... set its properties

  out.writeObject(conf);
```

Now all you have to do is close the stream:

```
out.close();
```

Multiple object instances (and of potentially differing types) can be written to the same `ObjectOutputStream`*. Simply call* `writeObject()` *more than once, and the next object is appended to the stream. Also note the file extension,* `config`*, appended to the file was arbitrarily chosen.*

Reading the Configuration from Disk

Deserializing objects back into memory is as easy as serializing them. To read your configuration data model from disk, create an `ObjectInputStream` wrapped around a `FileInputStream` (because in this case you saved your `Configuration` to a file):

```
ObjectInputStream in = new ObjectInputStream(
                                  new FileInputStream("appconfig.config"));
```

The counterpart to `ObjectOutputStream`'s `writeObject()` is `readObject()` in `ObjectInputStream`. If more than one object was explicitly written with a call to `writeObject()`, `readObject()` can be called more than once. The method `readObject()` returns an `Object` that needs to be cast the proper type — so the developer must know some of the details about the order in which object instances were saved to the stream. In addition to potentially throwing a `java.io.IOException` if the stream was corrupted or other I/O error, `readObject()` can throw a `java.lang.ClassNotFoundException`. The `ClassNotFoundException` occurs if the VM cannot find the class for the type of the object instance being deserialized. The following line of code reads the `Configuration` object back into memory:

```
Configuration conf = (Configuration) in.readObject();
```

After reading the object back in, you can use it as you would use any normal Java object. After you are done with the `ObjectInputStream`, close it like you would any other subclass of `InputStream`:

```
in.close();
```

As you can see, reading and writing objects using `ObjectInputStream` and `ObjectOutputStream` is a simple process with powerful functionality. Later, this section discusses customizing and extending the serialization process as well as some of the pitfalls that can occur along the way.

Wrapping Your Serialization and Deserialization Code Inside Swing Actions

Now that you have seen how to create and store data models, it is time to see your configuration data model serialization and deserialization code in the context of a real application. Because your application is a JFC-based Swing application, you will integrate your code to serialize and deserialize `Configuration` into the UI framework via Swing's `javax.swing.Action` interface. Actions are a useful way to generalize UI commands — such as a save or open command. These commands usually appear in multiple places in a UI; in the case of save and open, usually in the File menu and on the application's toolbar. Swing components such as menus and toolbars allow actions to be added and they create the necessary events and properties to control them. Actions abstract some of the UI code away, and allow the developer to concentrate on the logic of an action, like saving a file to disk. Your actions will need a reference to your application, to get and set its configuration before it serializes or deserializes the `Configuration` instance. Your actions will inherit from the class `javax.swing.AbstractAction`, because that class takes care of all of the methods in the `Action` interface except for the event method `actionPerformed()`. The class diagram shown in Figure 5-4 illustrates where your actions, `LoadConfigurationAction` and `SaveConfigurationAction`, fit with respect to `Action` and `AbstractAction`.

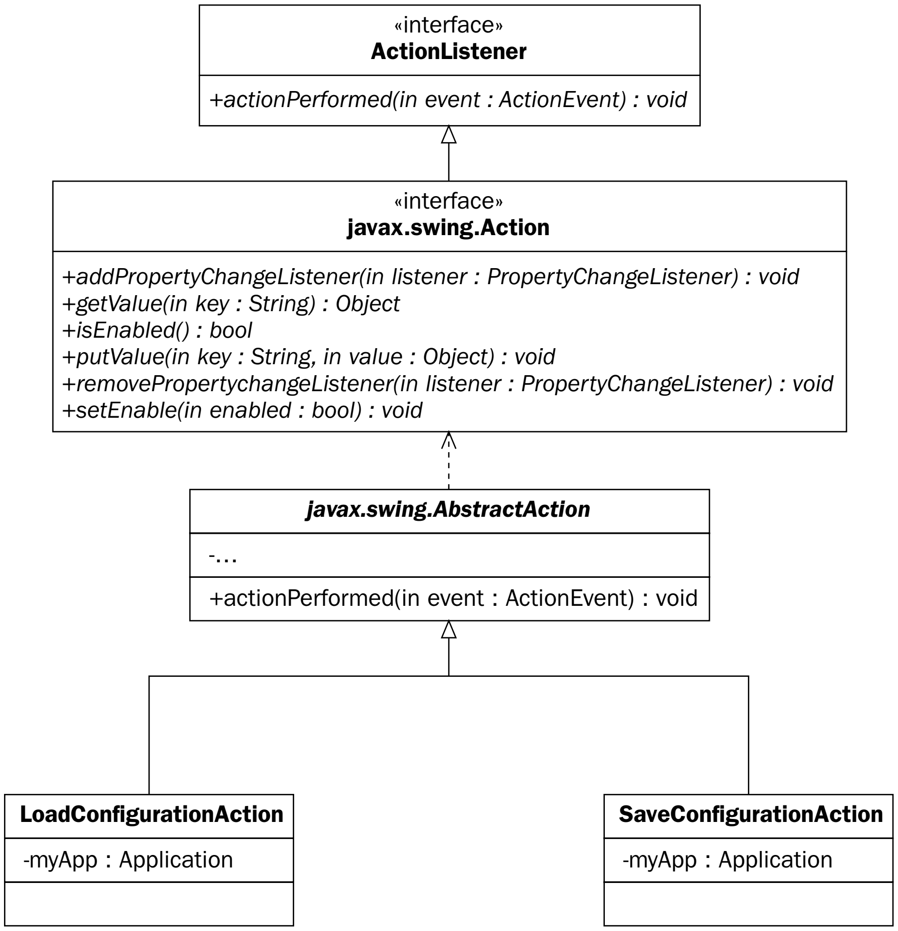

«interface»
ActionListener
+actionPerformed(in event : ActionEvent) : void
«interface»
javax.swing.Action
+addPropertyChangeListener(in listener : PropertyChangeListener) : void
+getValue(in key : String) : Object
+isEnabled() : bool
+putValue(in key : String, in value : Object) : void
+removePropertychangeListener(in listener : PropertyChangeListener) : void
+setEnable(in enabled : bool) : void
javax.swing.AbstractAction
-...
+actionPerformed(in event : ActionEvent) : void
LoadConfigurationAction
-myApp : Application
SaveConfigurationAction
-myApp : Application

Figure 5-4

All of the code for both of these actions will reside in the event-driven method, `actionPerformed()`. When the user of the application clicks the save configuration menu item or button, this code will be invoked. The same goes for the action to load the application's configuration.

Implementing the Save Configuration Action

The main area of interest in any `Action` implementation is the `actionPerformed()` method. This method is called when a user clicks the menu item or button containing the `Action`. For your save action, you want the user to be first prompted to choose a file location, and then save the application's `Configuration` object instance to that file location. The implementation is fairly straightforward. Display a file chooser first, and if the user selects a file, the application's `Configuration` instance is retrieved:

```
    public void actionPerformed(ActionEvent evt) {
      JFileChooser fc = new JFileChooser();
      if (JFileChooser.APPROVE_OPTION == fc.showSaveDialog(myApp)) {
        try {
          Configuration conf = this.myApp.getConfiguration();
```

Now that you know the file location, simply serialize the `Configuration` object to disk:

```
          ObjectOutputStream out = new ObjectOutputStream(
                                  new FileOutputStream(fc.getSelectedFile()));

          out.writeObject(conf);

          out.close();

        } catch (IOException ioe) {
          JOptionPane.showMessageDialog(this.myApp, ioe.getMessage(), "Error",
                                        JOptionPane.ERROR_MESSAGE);

          ioe.printStackTrace();

        }
      }
    }
```

Implementing the Load Configuration Action

The load action is similar to the save action. First, the user is prompted for a file. If the user selects a file, you will try to open it:

```
    public void actionPerformed(ActionEvent evt) {
      JFileChooser fc = new JFileChooser();
      if (JFileChooser.APPROVE_OPTION == fc.showOpenDialog(myApp)) {
        try {
          ObjectInputStream in = new ObjectInputStream(
                                  new FileInputStream(fc.getSelectedFile()));
```

If the user selects a file that is not a serialized instance of `Configuration`, an `IOException` will be thrown. If the instance of `Configuration` is successfully read, load it into your application via the application's `setConfiguration()` method:

```
          Configuration conf = (Configuration) in.readObject();

          in.close();

          myApp.setConfiguration(conf);
        } catch (IOException ioe) {
          JOptionPane.showMessageDialog(this.myApp,
                                 "File is not a configuration file!", "Error",
                                       JOptionPane.ERROR_MESSAGE);

          ioe.printStackTrace();

        } catch (ClassNotFoundException clEx) {
          JOptionPane.showMessageDialog(this.myApp,
                                 "Classpath incorrectly set for application!",
                                 "Error", JOptionPane.ERROR_MESSAGE);

          clEx.printStackTrace();
        }
      }
    }
```

Giving Your Application a Time-Based License Using Serialization

Serialization can be used to solve a variety of problems. It is easy to save JavaBeans and the data models for various kinds of application data as you saw in the last example. Serialization is not limited to simply saving objects to disk. Because `ObjectInputStream` and `ObjectOutputStream` are subclasses of `InputStream` and `OutputStream`, respectively, they can be used in any situation a normal stream could be. Objects can be serialized over the network, or read from a JAR file. Serialization is a fundamental aspect of Java's Remote Method Invocation (RMI) — it is the technology behind passing objects by value in RMI method calls.

To continue with the drawing application example, suppose you want to give it a time-based license. For the demo version of the application, you only want the application to be fully active for 30 days. After 30 days, you will require users to purchase a full license to use the product. There are many ways to do this, but using the serialization API could be an effective way to produce a time-based license file. The biggest challenge to creating time-based licenses is making it difficult for users to overcome the license, either by setting their computer's clock at an incorrect time, or by modifying whatever license file gets distributed (or registry key for some Windows-based applications and so on). Because Java's serialization produces a binary format that is unfamiliar to anyone except Java developers, it will make a good format for your application's license file. You will also need some mechanism to guard against users setting the incorrect date on their computer clock to give them a longer license. To do so, you will require them to authenticate against a timeserver on your network. The high-level design looks like Figure 5-5.

The next step in the design is to model the license file. Because you are using Java Serialization, all you need to do is produce a class that implements `Serializable` and contains the necessary fields to do license validation against the timeserver. The `License` class will look like Figure 5-6.

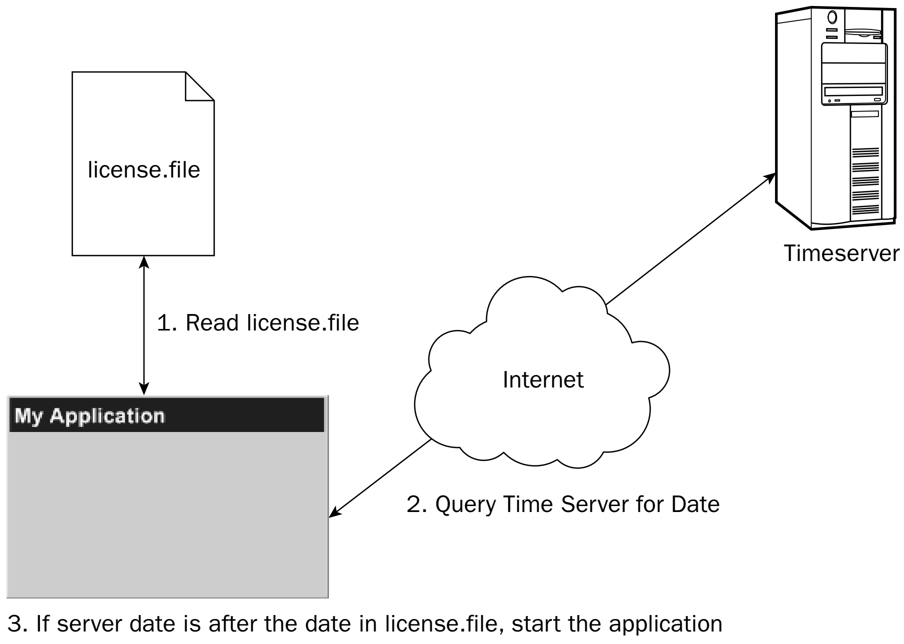

license.file
1. Read license.file
My Application
Internet
Timeserver
2. Query Time Server for Date
3. If server date is after the date in license.file, start the application

Figure 5-5

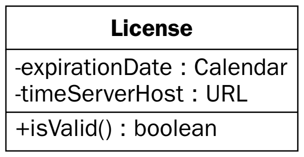


Figure 5-6

Implementing the License

The license file will consist of a serialized instance of the `License` class. The two data attributes it contains are `expirationDate`, which is the date when the license expires (stored in a `java.util.Calendar` instance), and `timeServerHost`, which is the `java.net.URL` representing the Internet address of your timeserver. Save the address as well as the expiration date to prevent tampering with the URL. The `isValid()` method gets the current date from the timeserver and checks to see if the expiration date is before the date returned from the timeserver. If it is, the license is valid. Actually implementing, the `License` yields the following code listing:

```
package book;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.Serializable;
import java.net.URL;
import java.util.Calendar;

public class License implements Serializable {
  private Calendar expirationDate;

  private URL timeServerHost;

  public boolean isValid() throws IOException, ClassNotFoundException {
    ObjectInputStream in = new ObjectInputStream(timeServerHost.openStream());

    Calendar serverDate = (Calendar) in.readObject();

    in.close();

    return serverDate.before(expirationDate);
  }

  public Calendar getExpirationDate() {
    return expirationDate;
  }

  public void setExpirationDate(Calendar expirationDate) {
    this.expirationDate = expirationDate;
  }

  public URL getTimeserverHost() {
    return timeServerHost;
  }

  public void setTimeServerHost(URL timeServerHost) {
    this.timeServerHost = timeServerHost;
  }
}
```

Look into the implementation for `isValid()`. One thing not yet discussed in detail is the protocol you need to define between the timeserver and the `License`. How does the `isValid()` method get the current date from the timeserver? A normal `HTTP GET` request is sent to a URL on the timeserver, and instead of it returning an HTML page, it will return an instance of `java.util.Calendar`. Using the `timeServerHost URL` object, you open an `ObjectInputStream` via an HTTP request over the network:

```
ObjectInputStream in = new ObjectInputStream(timeServerHost.openStream());
```

From here, you simply read in a `Calendar` object just like any other object in Java serialization. After the object is read in, compare the `expirationDate` to see if it is before or after the date returned from the timeserver:

```
Calendar serverDate = (Calendar) in.readObject();

in.close();

return serverDate.before(expirationDate);
```

Serialization can make complex tasks very straightforward. Java programmers can serialize and deserialize information without ever really leaving the Java environment because actual class instances can be serialized. Rather than creating your own date format on the server, you simply returned an instance of `Calendar`. All the low-level details of marshalling information over the network and finding a format you can use for date information were all taken care of by Java Serialization and the `URL` class.

Implementing the Timeserver

Now that you know what the timeserver is supposed to do, you must actually implement it. The timeserver will run as a Java web application (see Chapters 7 and 8 for much more detailed information on web applications). All you need is a simple servlet. The servlet will take care of the HTTP request and response, and allow you to write a `Calendar` object out to the client. Here is the servlet code that runs on the timeserver:

```
package book;

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.logging.Logger;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServerDate extends HttpServlet {

  private Logger logger;

  public void init(ServletConfig config) throws ServletException {
    logger = Logger.getLogger(ServerDate.class.getName());
  }

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
                      throws IOException, ServletException {

    logger.info("Received date request");

    ObjectOutputStream out = new ObjectOutputStream(resp.getOutputStream());
    Calendar calendar = new GregorianCalendar();

    out.writeObject(calendar);

    out.close();

    logger.info("Wrote the date: " + calendar.getTime());
  }
}
```

By implementing the `doGet()` method, the servlet handles `HTTP GET` requests (which you are expecting from your `License` clients). The method is straightforward. All you do is wrap an `ObjectOutputStream` around the normal `ServletOutputStream`:

```
ObjectOutputStream out = new ObjectOutputStream(resp.getOutputStream());
```

Once you have your output stream back to the client, simply write a new instance of `Calendar` (which corresponds to the current date and time):

```
Calendar calendar = new GregorianCalendar();

out.writeObject(calendar);

out.close();
```

The `License` class and `ServerDate` servlet take care of the actual license file, and the means to validate the date it stores. In the next section, you see how to integrate the components in this example, with your configuration data model and Swing actions, into an actual Swing application.

Tying Your Serialization Components into the Application

You have developed Swing actions that load and save your configuration data model. You wrote a licensing system that uses serialization to specify both the license file format and the date and time format of your simple timeserver. Actually tying these pieces into your application is not very difficult, but helps to paint the larger picture of how serialization can fit into a real application design.

The first task your application does at startup is load the license file and verify that the date contained therein is before the date returned on the timeserver. The `license.file` is read in from the application's Java ARchive file (JAR) and then the validity of the license is verified against the timeserver found at the URL in the serialized license:

```
    try {
      ObjectInputStream in = new ObjectInputStream(
                         Application.class.getResourceAsStream("license.file"));

      License license = (License) in.readObject();

      in.close();

      if (!license.isValid()) {
        JOptionPane.showMessageDialog(this, "Your license has expired",
                                        "License", JOptionPane.ERROR_MESSAGE);
        System.exit(1);
      }

    } catch (Exception ex) {
      JOptionPane.showMessageDialog(this, ex.getMessage(), "License",
                                           JOptionPane.ERROR_MESSAGE);
      System.exit(1);
    }
```

Notice how the license file, `license.file`, is loaded as a resource. Your application assumes that the license was packaged into the same JAR file as the application. This means that there must be some sort of license managing utility to put the `license.file` into the same JAR file as the application. That utility is not discussed, however, because it is irrelevant to this example. Getting the `license.file` from the JAR reduces the risk of a user attempting to tamper with its contents to gain a longer license. The Java Serialization API is a binary format that is not human-readable, but could potentially be recognized by another Java developer. If you really cared an awful lot about no one tampering with your `license.file`, you could always encrypt it using the Java Cryptography Extension (JCE) API. JCE allows you to encrypt any `OutputStream` and hence you could encrypt (and later decrypt) an `ObjectOutputStream`.

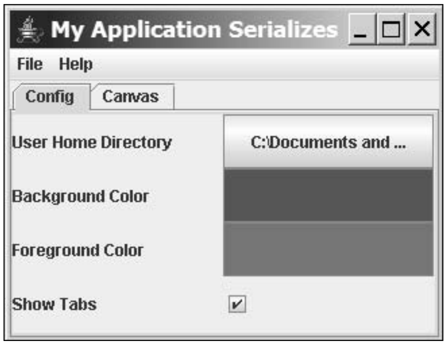
Adding your Swing actions to the File menu looks like this:

```
fileMenu.add(new JMenuItem(new LoadConfigurationAction(this)));
fileMenu.add(new JMenuItem(new SaveConfigurationAction(this)));
```

Now you have tied in all of your components based on serialization. The following is a stripped-down code listing for your basic application. Look at the `setConfiguration()`, `loadConfiguration()`, and `getConfiguration()` methods, because these are what your Swing actions interact with:

```
package book;

import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.io.ObjectInputStream;

import javax.swing.*;

public class Application extends JFrame {

  private Configuration configuration = new Configuration();

  private JButton hdButton;

  private JButton bcButton;
  private JButton fgButton;
  private Color defaultColor;

  private JCheckBox showTabsCheckBox;

...

  public Application() {
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setTitle("My Application Serializes");

    try {
      ObjectInputStream in = new ObjectInputStream(
                           Application.class.getResourceAsStream("license.file"));

         License license = (License) in.readObject();

         in.close();

         if (!license.isValid()) {
           JOptionPane.showMessageDialog(this, "Your license has expired",
                                           "License", JOptionPane.ERROR_MESSAGE);
           System.exit(1);
         }

       } catch (Exception ex) {
         JOptionPane.showMessageDialog(this, ex.getMessage(), "License",
                                                  JOptionPane.ERROR_MESSAGE);
         System.exit(1);
       }

...

    JMenuBar menu = new JMenuBar();
    JMenu fileMenu = new JMenu("File");
    fileMenu.add(new JMenuItem(new LoadConfigurationAction(this)));
    fileMenu.add(new JMenuItem(new SaveConfigurationAction(this)));
    fileMenu.addSeparator();
...
    this.pack();
    this.setVisible(true);
  }

  private JPanel createConfigDisplayPanel() {
...
    return panel;
  }

  private void loadConfiguration() {
    hdButton.setText(this.configuration.getUserHomeDirectory());

    Color bcColor = this.configuration.getBackgroundColor();
    if (bcColor != null) {
      bcButton.setBackground(bcColor);
      bcButton.setText(null);
    } else {
      bcButton.setText("<No color set>");
      bcButton.setBackground(this.defaultColor);
    }

    Color fgColor = this.configuration.getForegroundColor();
    if (fgColor != null) {
      fgButton.setBackground(fgColor);
      fgButton.setText(null);
      } else {
        fgButton.setText("<No color set>");
        fgButton.setBackground(this.defaultColor);
      }

      showTabsCheckBox.setSelected(this.configuration.isShowTabs());
    }

    public Configuration getConfiguration() {
      return configuration;
    }

    public void setConfiguration(Configuration configuration) {
      this.configuration = configuration;

      this.loadConfiguration();
    }

    public static void main(String[] args) {
      Application app = new Application();
    }
  }
```

Figure 5-7 shows the application editing part of the configuration data model. To get to this point means that the application was able to verify the license. Notice in the `loadConfiguration()` method in the previous code listing how the color buttons are set, the checkbox is checked, and the user's home directory is placed on the first button when a configuration is loaded. Users can then change these options, which modifies the application's `Configuration` object.



User Home Directory: C:\Documents and ...
Background Color
Foreground Color
Show Tabs: ✓

Figure 5-7

Once the `Configuration` object is changed, it can be saved back to disk. Because the whole configuration data model is contained in the `Configuration` object, all you need to do is export it to disk using your action, as shown in Figure 5-8.

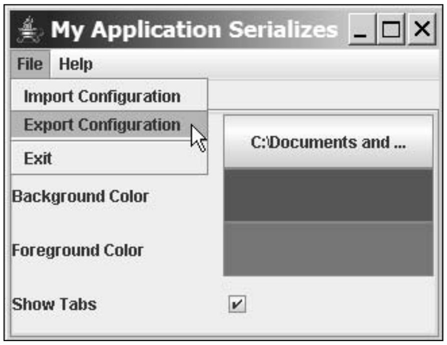


Figure 5-8

Extending and Customizing Serialization

Though most of the time the Java Serialization API provides enough functionality out of the box, there are some cases where a greater level of control is necessary for the developer. Sometimes you will not want every field of a class serialized to disk. Other times, you may want to append additional information not included in class fields into the stream — or maybe modify some of the class's members before serialization occurs. A very common case for customizing serialization occurs when a class definition is modified (in other words, the code is changed and the class recompiled) — that is, fields are renamed, or other fields added and others removed; classes serialized prior to these changes will have errors upon deserialization. This section discusses some of the commonly used mechanisms for customizing and extending Java Serialization.

The Transient Keyword

The `transient` keyword in the Java language is used for Java Serialization. Any field marked `transient` will not be saved to disk. This is useful when a class contains a reference to another object that does not implement `Serializable`, but you still would like to persist a class instance to disk. Sometimes certain fields are runtime-dependent and should not be persisted. Suppose in your `Configuration` object you wanted to additionally store a reference to your application (for callbacks perhaps). When you saved your application to disk, you would certainly not want to persist the application and every object associated with it on its object graph (even if all its objects implemented `Serializable`). To mark a field `transient`, simply put the keyword before the definition of the object or primitive:

```
    private transient Application application;
```

The `transient` keyword is an easy way to quickly mark which fields of your class you would like the Serialization API to skip over and not save.

However, when a class is reconstructed after being serialized these fields marked `transient` *will be* `null` *(or if they are primitives, their default value), unless they are given a default value, or set in the default constructor of the class.*

Customizing the Serialization Format

Sometimes there is a need to perform additional operations either right before an object is serialized or right after it is deserialized. This need could arise if a class must retrieve data stored externally, such as on a server or in a cache, right before it is serialized. Objects may wish to verify some of their fields right after deserialization and fill in or create some of the fields marked `transient`. There are two methods you can add to a class to add behavior to the serialization and deserialization process, `writeObject()` and `readObject`. These methods are not part of any interface, and for them to be called by the Java serialization engine, they must have the exact signature as shown in the following listing:

```
private void writeObject(ObjectOutputStream out) throws IOException {
  // can do things like validate values, get data from an external source, etc

  out.defaultWriteObject(); // invokes normal serialization process on this object
}

private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException {
   in.defaultReadObject(); // invokes normal deserialization process on this object

  // can do things like validate values, produce new values based on data, etc
}
```

The method `writeObject()` is called right before a class is serialized. The user can control when the class is actually serialized by calling `defaultWriteObject()` on the `ObjectOutputStream` as shown in the previous code. Doing so invokes the normal Java Serialization process on the current object. Before or after the object is written to the stream, though, values to current data members could be changed or updated. Additional information can also be written to the `ObjectOutputStream`. The `ObjectOutputStream` also implements the `java.io.DataOutput` interface, which includes methods for writing primitives (and `Strings`).

The `readObject()` method is called right before an object is deserialized. It is the natural counterpart to `writeObject()`. Similarly, the user can control when the object is deserialized by calling `defaultReadObject()` on the `ObjectInputStream`. After an object is deserialized, fields that did not have values could be assigned default values, or the values that were assigned could be checked. If any extra data was written to the `ObjectOutputStream` in `writeObject()` it *must* be read back in the `readObject()` method. For example, if the user wrote `java.util.Date` object to the stream before writing the current object (to signify when the object was serialized), the `Date` object would have to be read in *before* `defaultReadObject()` was called, or the stream would be in the incorrect place to read the instance in using `defaultReadObject()`, and an exception would occur.

Verification and Validation for Configuration

One example of how implementing `writeObject()` and `readObject()` could be useful to your `Configuration` object is data verification and validation. Your `Configuration` object stores the user's home directory and a list of recently accessed files. Between the time when a `Configuration` instance is serialized and later deserialized, these files and directory may not exist (they could have been moved or deleted). When your `Configuration` instance is deserialized, you want to remove the references to the directory or files that no longer exist where they originally did. To do this, implement the `readObject()` method as shown in the following code. After calling `defaultReadObject()` to populate the current instance of the object, you can go through the `userHomeDirectory` field and the `recentFiles` field to check if the files (and directory) exist. If a file or directory does not exist, simply set it to `null`:

```
    private void writeObject(ObjectOutputStream out) throws IOException {
      out.defaultWriteObject();
    }

    private void readObject(ObjectInputStream in) throws IOException,
  ClassNotFoundException {
      in.defaultReadObject();

      if (this.userHomeDirectory != null) {
        File f = new File(this.userHomeDirectory);
        if (!f.exists())
          this.userHomeDirectory = null;
      }

      if (this.recentFiles != null) {
        List list = new LinkedList();
        Collections.addAll(list, this.recentFiles);

        ListIterator it = list.listIterator();
        while (it.hasNext()) {
          String curr = (String) it.next();
          File f = new File(curr);
          if (!f.exists()) {
            it.remove();
          }
        }

        this.recentFiles = new String[list.size()];
        list.toArray(this.recentFiles);
      }
    }
```

The Externalizable Interface

Besides implementing `readObject()` and `writeObject()`, there is also an interface that extends `Serializable` that allows for greater customization of serialization and deserialization. The `java.io.Externalizable` interface allows more control of the serialization format than `readObject()` and `writeObject()`. It exists to allow developers to write their own custom formats for a class. With `Externalizable`, only the class identity is written to the stream by the Java Serialization API; the rest is left for the developer. The `Externalizable` interface looks like this:

```
  public interface java.io.Externalizable extends java.io.Serializable {
    public void readExternal(java.io.ObjectInput in) throws java.io.IOException,
        java.lang.ClassNotFoundException { }

    public void writeExternal(java.io.ObjectOutput out) throws java.io.IOException
        { }
  }
```

The methods `writeExternal()` and `readExternal()` are public instead of private like `readObject()` and `writeObject()`. Other classes can call these methods to read and write a class to disk without specifically invoking Java Serialization. `Externalizable` is not generally used very often, because normally when you want to save a class to disk, there is no need to completely customize the format. However, there may be times when `Externalizable` could come in handy. If you wanted to serialize a class that represented an image, and the in-memory representation was huge because it represented every pixel (like a bitmap), the `Externalizable` interface could be used to write the image in a different and compressed format (such as JPEG). The same could be done with `readObject()` and `writeObject()`, but these methods are not public, and in the case of your image-saving class, you may also want to save your image to disk outside of a serialization stream.

Versioning

The biggest stumbling block most developers run into with serialization is versioning. Many times classes will be serialized to disk, and then the definition of the class will change as source code is modified and the class recompiled. Maybe a field gets added, or one gets taken away. Design decisions could force the change of some internal data structures — for example, from lists to maps or trees. Any change to a class, by default, results in no other previously serialized instance to be deserialized — a version error results. Serialization versioning works by hashing a class based on its fields and class definition. Even if one of the field *names* is changed (but not its data type), previously serialized instances will not deserialize — the hash for the class has changed. Sometimes when the definition of a class changes, there would be no way to retain backward compatibility with previously saved instances. With smaller changes, especially things like name changes, or the addition or removal of one field, you probably would want to retain backward compatibility.

The Java Serialization API provides a way to manually set the hash of a class. The following field must be specified exactly as shown to provide the hash of the class:

```
private static final long serialVersionUID = 1L;  // version 1 of our class
```

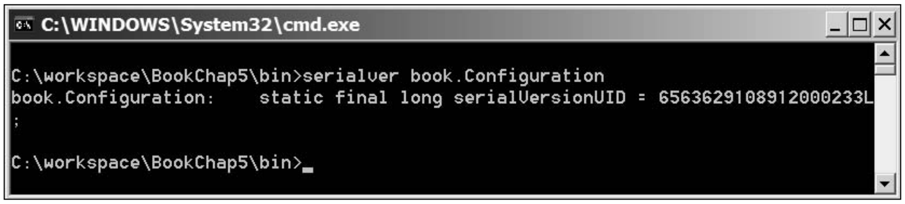
If the `serialVersionUID` is specified (and is `static` and `final`), the value given will be used as the version for the class. This means that if you define a `serialVersionUID` for your class, you will not get versioning errors when deserializing instances of previous class definitions. The Serialization API provides a best-effort matching algorithm to try to best deserialize classes saved with an older class definition against a newer definition. If a field was added since a class was serialized, upon deserialization, that field will be `null`. Fields whose names have changed or whose types have changed will be `null`. Fields removed will not be set. You will still need to account for these older versions, but by setting the `serialVersionUID`, you are given a chance to do so, rather than just have an exception be thrown right when the deserialization process is attempted. It is recommended to set a `serialVersionUID` for a class that implements `Serializable`, and to change it only when you want previously serialized instances to be incompatible.

Say you have previously serialized class instances and want to change a field or add another. You did not set a `serialVersionUID`, so *any* change you make will render it impossible to deserialize the old instances. The JDK provides a tool to identify a class's hash that did not have a `serialVersionUID` field. The `serialver` tool identifies the JVM's current hash of a compiled class file. Before you modify your class, you can find the previous version's hash. For the `Configuration` object, for example, you did not previously define a `serialVersionUID` field. If you add a field, you will not be able to deserialize old instances. *Before* modifying the class, you need to find the hash. By running the `serialver` tool, you find the hash by:

```
serialver book.Configuration
```

`Serialver` *is located in the* `\bin` *directory of your JDK.*

`Configuration` must be on the classpath for the `serialver` tool to work. The output of the tool looks like Figure 5-9.



C:\WINDOWS\System32\cmd.exe

```
C:\workspace\BookChap5\bin>serialver book.Configuration
book.Configuration:    static final long serialVersionUID = 6563629108912000233L
;

C:\workspace\BookChap5\bin>_
```

Figure 5-9

Now you can add this `serialVersionUID` value to your `Configuration` class:

```
    private static final long serialVersionUID = 6563629108912000233L;
```

You can now add new fields and still deserialize your older instances. Versioning is such an issue with serialization that you should *always* set a `serialVersionUID` for any class that implements `Serializable` right off the bat. This is especially important if your class is to be serialized and deserialized on JVMs from different vendors, because the default hashing algorithm to find a class's `serialVersionUID` is implementation dependent.

When to Use Java Serialization

Java Serialization is a simple but very powerful API. It is easy to use and can serialize most any type of data your application could have. Its main strengths are:

- ❑ Can serialize complex Java class structures with little code from the developer
- ❑ An efficient binary file format

The file format defined by the Serialization API is usually what determines its suitability for an application. It is a fairly efficient file format, because it is binary as opposed to XML or other text file format. However, the file format also produces the following weaknesses (though potentially not weaknesses depending on your application's requirements):

- ❑ Not human-readable
- ❑ Only Java-based applications can access the serialized data

Because the data is in a binary format, it cannot be edited with simple text editors. Your application's configuration from the example could only be modified from the application. The data was not in an XML format (or other text format) where you could edit it in both the application or in an external editor. Sometimes this is important, but certainly not always. The key downside to Java Serialization is that only Java-based applications can access the serialized data. Because the serialization format is storing actual Java class instances, in a file specification particular to Java, no parsers have been written in other languages for parsing data serialized with the Java Serialization API.

The Java Serialization API is most useful when developing data models for Java applications and persisting them to disk. If your application needs a common file format with other applications not written in Java, serialization is the wrong design choice. If the files do not need to be human-readable, and the only applications written for reading them will be in Java, serialization can be a great design choice.

Serialization can sometimes be a good temporary solution. Every Java application will have some sort of in-memory data model. Certain classes will store data in memory for the application to use. These classes could be persisted to disk, or populated from reading some other file format. Serialization could be initially used to save and restore these class instances. Later on though, as the need for a common file format between non-Java-based applications arises, routines could be written to take the data in those classes and persist it to another format. In other words, the same classes would still be used for the application's internal memory model, just the load and save routines would have to change. You will see in the next sections how you can serialize your application's configuration data in other formats and still retain the use of `Configuration` as your in-memory way of representing that data. Only the load and save code will need to change — not the actual data model.

JavaBeans Long-Term Serialization: XMLEncoder/Decoder

The XMLEncoder/Decoder API is the new recommended persistence mechanism for JavaBeans components starting from the 1.4 version of the JDK. It is the natural progression from serialization in many respects, though it is not meant to replace it. Like Java Serialization, it too serializes object graphs. XMLEncoder/Decoder came around in response to the need for long-term persistence for Swing toolkit components. The Java Serialization API was only good for persisting Swing components in the short-term because it was only guaranteed to work for the same platform and JDK version. The reason for this is that some of the core UI classes that Swing depends on must be written in a platform-dependent manner. Their private data members may not always match up — leading to problems with the normal Serialization API. The Swing API also has had a lot of fluctuation in its implementation. Classes like `JTable` used to take up 30MB of memory alone. As the implementation has improved, the internal implementations of many of these Swing classes have drastically changed. A new serialization API was developed in response to the challenge of true portability between different implementations and versions of the JDK for Swing/JFC classes. XMLEncoder/Decoder thus has a different set of design criteria than the original Java Serialization API. It was designed for a different usage pattern. Both APIs are necessary, with XMLEncoder/Decoder filling in some of the gaps of the Java Serialization API. XMLEncoder is a more robust and resilient API for long-term serialization of object instances, but is limited to serializing only JavaBeans components, and not any Java class instance.

Design Differences

Because the XMLEncoder/Decoder API is designed to serialize only JavaBeans components, the designers had the freedom to make XMLEncoder/Decoder more robust. Version and portability problems were some of the key issues many developers had with the original Java Serialization API. The XMLEncoder/Decoder API was written in response to these issues. The XMLEncoder/Decoder API serializes object instances without *any* knowledge of their private data members. It serializes based upon the object's methods, its JavaBean properties, exposed through the JavaBeans convention of getters and setters (`getXXX` and `setXXX`). By storing an object based upon its public properties rather than its underlying private data members, the underlying implementation is free to change without affecting previously serialized instances (as long as the public properties remain the same). This supports more robust, long-term persistence, because a class's internal structure could be completely rewritten, or differ across platforms, and the serialized instance would still be valid. A simple example of a JavaBean follows:

```
public class MyBean {
  private String myName;

  public String getMyName() { return this.myName; }

  public void setMyName(String myName) { this.myName = myName; }
}
```

Internal data members could be added, the field `myName` could be changed to a character array or `StringBuffer`, or some other mechanism of storing a string. As long as the methods `getMyName()` and `setMyName()` did not change, the serialized instance could be reconstructed at a later time regardless of other changes. You will notice that `MyBean` does not implement `Serializable`. XMLEncoder/Decoder does not require classes it serializes to implement `Serializable` (or any other `interface` for that matter). Only two requirements are levied upon classes for XMLEncoder/Decoder to serialize:

1. The class must follow JavaBeans conventions.
2. The class must have a default constructor (a constructor with no arguments).

In the upcoming "Possible Customization" section, you will see how both these requirements can be side-stepped, but at the expense of writing and maintaining additional code to plug into the XMLEncoder/Decoder API.

XML: The Serialization Format

The XMLEncoder/Decoder API lives true to its name and has its serialization format based in XML text, in contrast to the binary format used by Java Serialization. The format is a series of processing instructions telling the API how to re-create a given object. The processing instructions instantiate classes, and set JavaBean properties. This idea of serializing *how* to re-create an object, rather than every private data member of an object, leads to a robust file format capable of withstanding any internal class change. This section does not get into the nitty-gritty details of the file format. It is helpful, though, to see the result of serializing a JavaBean using the XMLEncoder/Decoder API. The following code listing is the output of an instance of the `Configuration` object, serialized using the XMLEncoder/Decoder API. Because `Configuration` follows JavaBeans conventions, no special code additions were necessary to serialize an instance using XMLEncoder/Decoder. Notice how the whole object graph is again saved like the Java Serialization API, and because `java.awt.Color` and `java.awt.Point` follow JavaBeans conventions, they are persisted as part of the graph. XMLEncoder/Decoder also optimizes what information is saved — if the value of a bean property is its default value, it does not save the information:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0-beta2" class="java.beans.XMLDecoder">
 <object class="book.Configuration">
  <void property="recentFiles">
   <array class="java.lang.String" length="3">
    <void index="0">
     <string>c:\mark\file1.proj</string>
    </void>
    <void index="1">
     <string>c:\mark\testproj.proj</string>
    </void>
    <void index="2">
     <string>c:\mark\final.proj</string>
```

```
        </void>
       </array>
      </void>
      <void property="userHomeDirectory">
       <string>C:\Documents and Settings\Mark\My Documents</string>
      </void>
      <void property="showTabs">
       <boolean>true</boolean>
      </void>
      <void property="foregroundColor">
       <object class="java.awt.Color">
        <int>255</int>
        <int>255</int>
        <int>51</int>
        <int>255</int>
       </object>
      </void>
      <void property="backgroundColor">
       <object class="java.awt.Color">
        <int>51</int>
        <int>51</int>
        <int>255</int>
        <int>255</int>
       </object>
      </void>
     </object>
    </java>
```

One key point about the XML file format used by XMLEncoder/Decoder is that even though an XML parser in any language could read the file, the file format is still specific to Java. The file format encodes processing instructions used to re-create serialized JavaBean class instances, and is therefore not directly useful to applications written in other languages. It would be possible to implement a reader in another language that read some data from this file format, but it would be a large and fairly difficult task. The other language would also need to have some sort of notion of JavaBeans. Think of this format as a Java-only file format and do not rely on it for transmitting data outside of the Java environment. Later this chapter discusses the Java API for XML Binding (JAXB), which is far more suited to exporting data to non-Java consumers.

Because XML is human-readable, it is possible to save class instances to disk and then edit the information with a text editor. Editing the previous XML document would not be for the casual user; it would be more useful to a developer because some knowledge of how the XMLEncoder/Decoder API stores information is necessary to understand *where* to modify the file. If you wanted users to be able to save the `Configuration` object to disk and then edit it outside of the application, you probably would not choose the XMLEncoder/Decoder XML file format. In the previous file, for example, `java.awt.Color` was persisted using four integer values, described only by `int` for each one. What user would know that they correspond to the red, blue, green, and alpha channels of a color, and that they can range from 0–255? A descriptive configuration file format in XML would probably be a task for JAXB, as discussed later in the chapter. The file format used by XMLEncoder/Decoder is Java-specific and is also not well suited for general hand editing like many XML formats are. XML was simply the storage mechanism chosen — why define a new file type when you can use XML?

Key Classes

Using the XMLEncoder/Decoder API is similar to using the Java Serialization API. It was developed to have the same core methods and such that `java.beans.XMLEncoder` and `java.beans.XMLDeocoder` could literally be substituted for `ObjectOutputStream` and `ObjectInputStream`, respectively. `XMLEncoder` and `XMLDecoder` are the only classes needed for normal JavaBean serialization. The "Possible Customization" section briefly discusses some other classes needed to serialize JavaBeans that do not completely follow JavaBeans conventions. Following is a table of the classes needed to use XMLEncoder/Decoder:

Class (from java.beans)	**Function**
`XMLEncoder`	Class that takes an instance of a JavaBean and writes the corresponding XML representation of it to the `java.io.OutputStream` it wraps
`XMLDecoder`	Class that reads a `java.io.InputStream` and decodes XML formatted by `XMLEncoder` back into instances of JavaBeans

Serializing Your JavaBeans

The process of serializing JavaBeans using XMLEncoder/Decoder is almost exactly like the process of serializing a Java class using normal Java Serialization. There are four steps to serialization:

1. Make sure the class to be serialized follows JavaBeans conventions.
2. Make sure the class to be serialized has a default (no argument) constructor.
3. Serialize your JavaBean with `XMLEncoder`.
4. Deserialize your JavaBean with `XMLDecoder`.

To save an instance of the `Configuration` object to disk, simply begin by creating an `XMLEncoder` with a `FileOutputStream` object:

```
XMLEncoder encoder = new XMLEncoder(
                               new FileOutputStream("config.bean.xml"));
```

Then, write an instance of `Configuration`, `conf`, to disk and close the stream:

```
encoder.writeObject(conf);

encoder.close();
```

Reading the serialized instance of `Configuration` back in is just as simple. First the `XMLDecoder` object is created with a `FileInputStream` this time:

```
XMLDecoder decoder = new XMLDecoder(
                          new FileInputStream("config.bean.xml"));
```

Next, read in the object, much as you did with `ObjectInputStream`, and then close the stream:

```
Configuration config = (Configuration) decoder.readObject();

decoder.close();
```

On the surface, XMLEncoder/Decoder works much like Java Serialization. The underlying implementation though, is much different, and allows for the internal structure of classes you serialize to change drastically, yet still work and be compatible with previously saved instances. XMLEncoder/Decoder offers many ways to customize how it maps JavaBeans to its XML format, and some of those are discussed in the "Possible Customization" section.

Just like the Java Serialization API, multiple objects can be written to the same stream. `XMLEncoder's writeObject()` *method can be called in succession to serialize more than one object instance. When instances are deserialized, though, they must be deserialized in the same order that they were written.*

Robustness Demonstrated: Changing a Configuration's Internal Data

Suppose you want to change the way the `Configuration` object stores the references to the user's recently accessed files of your application. They were stored previously in the string array bean property, `recentFiles`, and accessed with the normal bean getter/setter pair, `getRecentFiles()` and `setRecentFiles()`. The `Configuration` object looked like this:

```
package book;

import java.awt.Color;
import java.awt.Point;
import java.beans.XMLDecoder;
import java.io.File;
import java.io.FileInputStream;
import java.util.ArrayList;
import java.util.List;

public class Configuration {

...

  private String[] recentFiles;

  public String[] getRecentFiles() {
    return recentFiles;
  }

  public void setRecentFiles(String[] recentFiles) {
    this.recentFiles = recentFiles;
  }

...

}
```

Now you would like to store them as a `java.util.List` full of `java.io.File` objects. If you do not change the signature of `getRecentFiles()` and `setRecentFiles()`, you can do whatever you like with the underlying data structure. The modified `Configuration` class illustrates changes to the storage of recent files to a `List` without changing the method signatures for the `recentFiles` bean property:

```
package book;

import java.awt.Color;
import java.awt.Point;
import java.beans.XMLDecoder;
import java.io.File;
import java.io.FileInputStream;
import java.util.ArrayList;
import java.util.List;

public class Configuration {

...

  private List recentFiles;

  public String[] getRecentFiles() {
    if (this.recentFiles == null || this.recentFiles.isEmpty())
      return null;

    String[] files = new String[this.recentFiles.size()];

    for (int i = 0; i < this.recentFiles.size(); i++)
      files[i] = ((File) this.recentFiles.get(i)).getPath();

    return files;
  }

  public void setRecentFiles(String[] files) {
    if (this.recentFiles == null)
      this.recentFiles = new ArrayList();

    for (int i = 0; i < files.length; i++) {
      this.recentFiles.add(new File(files[i]));
    }
  }

...

}
```

Notice how in the `setRecentFiles()` method you convert an array of `String` objects to a `List` of `File` objects. In the `getRecentFiles()` method you convert the `List` of `File` objects into an array of `String` objects. This conversion is the key to the information hiding principle that XMLEncoder/Decoder uses to serialize and deserialize object instances. Because XMLEncoder/Decoder only works with the operations of a class, the private data members can be changed. By keeping the interface the same, the `Configuration` class can undergo all kinds of incremental changes and improvements under the hood without affecting previously saved instances. This is the key benefit of XMLEncoder/Decoder that provides its ability to serialize instances not just in the short-term, but also in the long-term, by weathering many types of changes to a class's internal implementation.

The `main()` method shown in the following code demonstrates `XMLDecoder` deserializing an instance of `Configuration` previously saved with the older version of `Configuration` that stored the `recentFiles` property as a `String` array. The file this method is loading is the one shown previously in this section as sample output for XMLEncoder/Decoder (see the previous section "XML: The Serialization Format").

```
public static void main(String[] args) throws Exception {
  XMLDecoder decoder = new XMLDecoder(
               new FileInputStream("config.bean.xml"));

  Configuration conf = (Configuration) decoder.readObject();

  decoder.close();

  String[] recentFiles = conf.getRecentFiles();
  for (int i = 0; i < recentFiles.length; i++)
    System.out.println(recentFiles[i]);
}
```

As you can see, the output from the `main()` method confirms that not only was the old `Configuration` instance successfully read, but the new `List` of `File` objects is working properly, and populated with the correct objects:

```
c:\workingdir\file1.proj
c:\workingdir\testproj.proj
c:\workingdir\final.proj
```

Possible Customization

XMLEncoder/Decoder supports serialization of JavaBeans out of the box, but it can also be customized to serialize any class — regardless of whether or not it uses JavaBeans conventions. In fact, throughout the Swing/JFC class library, you will find classes that do not fully conform to JavaBeans conventions. Many types of collection classes do not; some Swing classes have other ways of storing data besides getters and setters. The following XML file is a serialized instance of a `java.util.HashMap`, and a `javax.swing.JPanel`. Both of these classes have their data added to them by methods that do not follow the JavaBeans convention:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0-beta2" class="java.beans.XMLDecoder">
 <object class="java.util.HashMap">
  <void method="put">
   <string>Another</string>
   <string>AnotherTest</string>
  </void>
  <void method="put">
   <string>Mark</string>
   <string>Test</string>
  </void>
 </object>
 <object class="javax.swing.JPanel">
  <void method="add">
   <object class="javax.swing.JLabel">
    <void property="text">
     <string>Mark Label</string>
    </void>
```

```
      </object>
     </void>
    </object>
   </java>
```

Note how data is added to a `HashMap` by its `put()` method, and components are added to `JPanels` by its `add()` method. How does the XMLEncoder/Decoder API know how to look for this — or even find the data that should be inserted via those methods? Because its file format is a series of processing instructions, XMLEncoder/Decoder can serialize the information necessary to make method calls to disk. This generic capability lets XMLEncoder/Decoder do any kind of custom initialization or setting of data that a class may require. Just because the file format supports this type of generic processing instructions, though, does not mean that the XMLEncoder automatically knows how to use them. The solution is the API's `java.beans.PersistenceDelegate` class.

Persistence Delegates

Every class serialized and deserialized has an instance of `java.beans.PersistenceDelegate` associated with it. It may be the default one included for classes following the JavaBeans conventions, or it could be a custom subclass of `PersistenceDelegate` that writes the processing instructions needed to re-create a given instance of a class. Persistence delegates are responsible only for writing an object to disk — *not* reading them. Because all objects are written in terms of known processing instructions, these instructions can be used to re-create the object. Reading an object consists solely of executing the processing instructions as they were written — independent of whether these instructions were written using a custom `PersistenceDelegate`. How to write a custom persistence delegate is a fairly complex topic that is out of the scope of this section.

For detailed information on how to use and create custom persistence delegates, visit `http://java.sun.com/products/jfc/tsc/articles/persistence4/` to read an article written by Philip Mine, the designer and author of XMLEncoder/Decoder API.

When to Use XMLEncoder/Decoder

Use of the XMLEncoder/Decoder API over the Java Serialization API is generally preferred when one is serializing object graphs consisting of JavaBeans and Swing components. It was designed precisely for that purpose and fixes the more generic Java Serialization API's shortcomings with respect to both JavaBeans, but especially Swing components. Prior to the XMLEncoder/Decoder API there was no built-in mechanism for the long-term serialization of Swing components. XMLEncoder/Decoder has only been in the JDK since version 1.4; if you must support any JDK released before 1.4 you cannot use XMLEncoder/Decoder.

Thinking in more general terms, and assuming your application has a data model you wish to persist to disk, XMLEncoder/Decoder has the following advantages:

- ❑ Can serialize complex Java class structures with little code from the developer
- ❑ You can add properties and remove properties from your JavaBeans class definitions without breaking previously serialized instances
- ❑ The internal private data structure of your beans can change without breaking previously serialized instances
- ❑ Instances are saved in XML, making the resulting files human-readable

Some of the potential downsides to choosing the XMLEncoder/Decoder for serializing your object graph of JavaBeans are:

- ❑ Even though the file format is human-readable, it is editable in the real world only by developers
- ❑ Even though the file format is XML, it is still Java-specific — it would take great effort to allow a non-Java-based application to read the data
- ❑ Every piece of data you want persisted in the class must be a JavaBean property (or customized with a special persistence delegate)

The XMLEncoder/Decoder API is perfect for what it is designed for — the long-term serialization of JavaBeans components for use later on by Java-based applications. Because it is so customizable, it can often be used for a variety of other purposes, and serialize a lot of data beyond ordinary JavaBeans. Generally though, its main advantage over normal Java Serialization is its robustness through even class definition changes. Apart from that, however, it still has the same limitations of the Java Serialization API. When you have an internal data model based with JavaBeans, XMLEncoder/Decoder makes sense. Once you would like your application's file formats to be read by other applications — other non-Java applications — eventually you will have to specify some other custom file format or write to an existing standard.

Flexible XML Serialization: Java API for XML Binding (JAXB)

Like both the Java Serialization API and the XMLEncoder/Decoder API, the Java API for XML Binding (JAXB) is a serialization framework for Java objects. It serializes and deserializes object graphs to and from an XML representation. The XML representation is flexible, and differs per Java class type being serialized. It is defined by mapping Java class members and bean properties to various XML elements or attributes. The mapping can be defined by either an XML Schema document or by simply using Java annotations to annotate Java classes, instructing the JAXB framework where a bean property or field member maps to which XML element or attribute in an XML document.

The XML Schema language is the World Wide Web Consortium's standard for specifying the XML structure for a particular XML file format. Think of XML Schema as being to XML what a database schema is to a relational database — it defines the blueprints and data storage for a particular domain of data. It is a widely accepted standard and already maps many primitive data types to XML. XML Schemas also let XML parsers validate an XML instance document to verify conformance to the schema's requirements. This can greatly reduce the time it takes to write code that must read XML because it does a lot of the validation for you. View the XML Schema specification at the following URL: `www.w3.org/TR/xmlschema-0/`**.**

Because JAXB lets the user actually define the mapping (or binding) between an XML representation and a Java object, the developer has control of the serialization file format. This flexibility allows the JAXB framework to read and write to third-party XML formats, as well as custom XML formats — allowing for much easier data interchange with other applications than data saved with either the Java Serialization API or the XMLEncoder/Decoder API, because both of these are dependent on Java (and having access to the particular classes used for serialization).

The original JAXB 1.0 specification mainly addressed these data interchange possibilities. As XML Schema has matured, it has become an increasingly popular way of specifying file formats. Many publicly open file formats publish an XML Schema defining the format, allowing third-party applications to easily read and write data defined in that format. For example, the Open Financial Exchange format (OFX) is a format many financial institutions use for exchanging data. Many banks with online access support downloading account information in this format. Third-party financial software can then import the account data. Because JAXB allows the creation of JavaBeans based on an XML schema, JAXB can generate JavaBeans that will read and write this format (the OFX specification is available at `www.ofx.net/ofx/de_spec.asp`). JAXB is the perfect API to use for reading and writing these different standard file formats defined in XML Schema.

Starting with version 6, the JAXB 2.0 framework is now included with the JDK. Unlike JAXB 1.0, it now takes full advantage of Java annotations. Previously, JAXB 1.0 only allowed the generation of JavaBeans from an XML schema. These generated JavaBeans could then be used to serialize data to the XML format defined by the original XML schema. It was a useful framework if one was required to read and write a particular XML format defined by an XML schema (such as the OFX format). With the addition of new annotations to the framework, JAXB has increased its scope to now serializing potentially any Java object. Now Java classes can be annotated to let the JAXB framework know how to serialize a particular instance of a class to a custom-defined XML format.

In the overview of JAXB presented in the remainder of this chapter, two use cases for JAXB are discussed. For both, you will continue to utilize the configuration data model example in this chapter for the hypothetical image manipulating application. First, you will create an XML schema to allow JAXB to generate a set of JavaBean classes used to serialize and deserialize data, illustrating both a high-level overview of XML Schema, as well as the process you would take to import a third-party schema. The second use case will illustrate the ease and benefits of simply annotating an existing set of Java classes, an application's data model, to serialize and deserialize it to an XML format suitable for interchange with other applications.

Sample XML Document for the Configuration Object

Suppose you want to take your `Configuration` data model and define an XML schema to represent it. You will not be able to write an XML schema that maps directly to the already-existing `Configuration` class, but you can write an XML schema that saves all the necessary data attributes to re-create the `Configuration` instance. To refresh your memory on what data attributes are stored in `Configuration`, refer to Figure 5-2.

You need to store data to represent a color, a point, a directory and file locations, and a Boolean variable. XML Schema is more than equipped to handle this — you just have to actually define it. Following is an XML instance document that contains all of the information you would need to re-create the `Configuration` object. Notice how it is not only human-readable in the sense that it is text, but also conceivably modifiable by a user. Colors are obviously defined, and the user's home directory element is easily modified. The following XML is far more readable than the output from the XMLEncoder/Decoder API:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration xmlns="http://book.org/Configuration">
   <user-settings>
      <user-home-directory>C:\Documents and Settings\Mark\My Documents</user-home-
directory>

      <recent-files>
```

```
            <recent-file>c:\mark\file1.proj</recent-file>

            <recent-file>c:\mark\testproj.proj</recent-file>

            <recent-file>c:\mark\final.proj</recent-file>
         </recent-files>
      </user-settings>

      <ui-settings>
         <palette-window-position>
            <x-coord>5</x-coord>

            <y-coord>5</y-coord>
         </palette-window-position>

         <tools-window-position>
            <x-coord>10</x-coord>

            <y-coord>10</y-coord>
         </tools-window-position>

         <background-color>
            <red>51</red>

            <green>51</green>

            <blue>255</blue>

            <alpha>255</alpha>
         </background-color>

         <foreground-color>
            <red>255</red>

            <green>255</green>

            <blue>51</blue>

            <alpha>255</alpha>
         </foreground-color>

         <show-tabs>true</show-tabs>
      </ui-settings>
   </configuration>
```

Note, though, that this XML file is *not* the XML schema; it is a document that conforms to the XML schema you will define in the next section. JAXB will generate Java classes that read and write files like the preceding XML. It will give you JavaBean-like access to all of the data contained in the document.

Defining Your XML Format with an XML Schema

Now that you have looked at a sample XML instance document containing a sample set of `Configuration` data for your data model, you can look under the hood and see how to specify the schema. This section goes through the various data types for configuration and looks at how they can be defined in a schema. The following list reiterates what data needs to be stored in the configuration data model:

- The user's home directory, a String value
- A flag whether or not to use a tabbed interface, a Boolean value
- A list of recently accessed files by the user, an array of String values
- Two colors, foreground and background, for drawing operations, color values
- Two points, for the last position of the tool and palette windows, point values

Though this section is not a thorough guide to using XML Schema, it does go through how to define the data bullets listed. First, though, XML Schema must be discussed. XML Schema is a simple but powerful language for defining and specifying what types various XML elements can be, and where they can appear in a document. There are two types of XML elements you can define with XML Schema: simple elements and complex elements. Simple elements have no attributes and contain only text data — they also have no child elements. Any example of a simple element would be:

```
<hello>world</hello>
```

Complex elements can have attributes, child elements, and potentially mix their child elements with text. An example of a complex element is:

```
<complex c="12">
   <hello>world</hello>
</complex>
```

XML Schema is fairly intuitive, and a full and thorough coverage of it is outside the scope of this book. You can find a great introductory tutorial at `www.w3schools.com/schema/default.asp`.

Defining Your Data: Configuration.xsd

To define your data, you will be using both simple and complex elements. Looking back at the bullet list of data points you will need — both the user's home directory and your tabbed interface flag (the first two bullets), can probably be modeled with simple elements. Here is how they will be modeled in XML Schema:

```
<xs:element name="user-home-directory" type="xs:string" />
<xs:element name="show-tabs" type="xs:boolean" />
```

You are defining elements, and requiring that the text within those elements be of the type specified. An instance example of both of these elements is:

```
<user-home-directory>c:\my-home-directory</user-home-directory>
<show-tabs>true</show-tabs>
```

Your string array of recent files is slightly more complex to model. You are going to model it as a complex element, with a child element for each individual recent file. First, define the complex type:

```
      <xs:complexType name="recentFilesType">
         <xs:sequence>
            <xs:element name="recent-file" type="xs:string" maxOccurs="unbounded" />
         </xs:sequence>
      </xs:complexType>
```

After defining your complex type, which is a sequence of `recent-file` elements, define the element that uses your custom XML type. Note how the `type` attribute in the following element definition corresponds to the `name` attribute in the preceding complex type definition:

```
   <xs:element name="recent-files" type="recentFilesType" minOccurs="0" />
```

An example instance of the `recent-files` element looks like this:

```
         <recent-files>
            <recent-file>c:\workingdir\file1.proj</recent-file>

            <recent-file>c:\workingdir\testproj.proj</recent-file>

            <recent-file>c:\workingdir\final.proj</recent-file>
         </recent-files>
```

Defining colors presents an interesting challenge. You must make sure you have enough information specified in the XML file to construct a `java.awt.Color` object. If you specify in the XML file the `red`, `green`, `blue`, and `alpha` components of a color, you will have enough information to construct a `java.awt.Color` instance. Model your color type in the schema as follows:

```
      <xs:complexType name="colorType">
         <xs:sequence>
            <xs:element name="red" type="xs:int" />

            <xs:element name="green" type="xs:int" />

            <xs:element name="blue" type="xs:int" />

            <xs:element name="alpha" type="xs:int" default="255" />
         </xs:sequence>
      </xs:complexType>
```

As you can see, the complex type, `colorType`, contains child elements for the RGBA components. These components are integer values and if the `alpha` component is not specified, it defaults to 255 (a totally opaque color). You define two elements that take your newly defined type, `colorType`: the foreground and background colors for your application's configuration data model:

```
            <xs:element name="background-color" type="colorType" minOccurs="0" />
            <xs:element name="foreground-color" type="colorType" minOccurs="0" />
```

An example instance of a `foreground-color` element is shown here:

```
         <foreground-color>
            <red>255</red>

            <green>255</green>

            <blue>51</blue>

            <alpha>255</alpha>
         </foreground-color>
```

The last major custom type you must define is the type for point objects. This type must have enough information encoded in the XML to construct a `java.awt.Point` instance. All you essentially need are integer values representing the `x` and `y` coordinates of a point. The last two element definitions are also listed that use the new XML type for points, `pointType`. These elements represent the position of the palette window and the tool window of your application:

```
      <xs:complexType name="pointType">
         <xs:sequence>
            <xs:element name="x-coord" type="xs:int" />

            <xs:element name="y-coord" type="xs:int" />
         </xs:sequence>
      </xs:complexType>

   <xs:element name="palette-window-position" type="pointType" minOccurs="0" />
   <xs:element name="tools-window-position" type="pointType" minOccurs="0" />
```

Now that you have defined all the basic types in your schema, they can be organized around other elements for better readability of your XML instance documents. The actual schema listed at the end of this section will have more elements and complex type definitions to account for document readability. The next step is to generate JAXB classes from your schema to start reading and writing XML documents that conform to your schema.

The full XML Schema Definition (XSD) file for your configuration data model, `configuration.xsd`, is listed here:

```
   <?xml version="1.0" encoding="utf-8" ?>
   <xs:schema targetNamespace="http://book.org/Configuration"
   elementFormDefault="qualified" xmlns="http://book.org/Configuration"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:complexType name="configurationType">
         <xs:sequence>
            <xs:element name="user-settings" type="user-settingsType" />

            <xs:element name="ui-settings" type="ui-settingsType" />
         </xs:sequence>
      </xs:complexType>

      <xs:complexType name="recentFilesType">
         <xs:sequence>
            <xs:element name="recent-file" type="xs:string" maxOccurs="unbounded" />
         </xs:sequence>
      </xs:complexType>

      <xs:complexType name="pointType">
         <xs:sequence>
            <xs:element name="x-coord" type="xs:int" />

            <xs:element name="y-coord" type="xs:int" />
         </xs:sequence>
      </xs:complexType>

      <xs:complexType name="colorType">
         <xs:sequence>
            <xs:element name="red" type="xs:int" />

            <xs:element name="green" type="xs:int" />

            <xs:element name="blue" type="xs:int" />

            <xs:element name="alpha" type="xs:int" default="255" />
         </xs:sequence>
      </xs:complexType>

      <xs:complexType name="ui-settingsType">
         <xs:sequence>
            <xs:element name="palette-window-position" type="pointType" minOccurs="0"
            />

            <xs:element name="tools-window-position" type="pointType" minOccurs="0" />

            <xs:element name="background-color" type="colorType" minOccurs="0" />

            <xs:element name="foreground-color" type="colorType" minOccurs="0" />

            <xs:element name="show-tabs" type="xs:boolean" />
         </xs:sequence>
      </xs:complexType>

      <xs:complexType name="user-settingsType">
         <xs:sequence>
            <xs:element name="user-home-directory" type="xs:string" />

            <xs:element name="recent-files" type="recentFilesType" minOccurs="0" />
         </xs:sequence>
      </xs:complexType>

      <xs:element name="configuration" type="configurationType" />

   </xs:schema>
```

Generating JAXB Java Classes from Your Schema

New in JDK 6, the tools for generating JAXB classes from an XML schema are now included in the JDK. Previously they were available as part of the Java Web Services Development Pack (JWSDP). To use the XML schema compiler, you must make sure your `PATH` environment variable includes the `/<JDK 6 home>/bin` directory. The `xjc` command in the JDK `bin` directory invokes the schema compiler. You saved your schema to the file `configuration.xsd`. To compile the schema, simply type at the command prompt, in the same directory as the schema, the following:

```
xjc -d gen configuration.xsd
```

The `-d` option simply tells the compiler in which directory to put the generated Java source files. In this case, you have a directory under the main project specifically for generated source files, `gen`, so if you modify the schema, you can easily regenerate the files to this same location. After running the `xjc` compiler, the following Java source files were generated:

```
org\book\configuration\ColorType.java
org\book\configuration\ConfigurationType.java
org\book\configuration\ObjectFactory.java
org\book\configuration\PointType.java
org\book\configuration\RecentFilesType.java
org\book\configuration\UiSettingsType.java
org\book\configuration\UserSettingsType.java
org\book\configuration\package-info.java
```

JAXB 1.0-generated sources are not compatible with the JAXB 2.0 runtime. You can, however, simply include the JAXB 1.0 runtime on your classpath to use older generated classes (or simply generate new classes to use the 2.0 runtime). In 2.0, the older `impl` *package classes are no longer generated (the JAXB runtime includes all the necessary runtime support in 2.0).*

Generated JAXB Object Graphs

JAXB-generated classes follow certain conventions corresponding to how an XML schema is written. For every top-level XML schema complex type defined, JAXB generates a class corresponding to that type. In this configuration example, for instance, you defined the following complex types in your XML schema: `configurationType`, `recentFilesType`, `pointType`, `colorType`, `ui-settingsType`, and `user-settingsType`. Respectively corresponding to each of these XML schema types are the following generated JAXB classes: `ConfigurationType`, `RecentFilesType`, `PointType`, `ColorType`, `UiSettingsType`, and `UserSettingType`. The Java package where each of the generated classes is placed depends on the XML namespace assigned to its type. In this example, all of the complex types defined were in the `http://book.org/Configuration` XML namespace. JAXB placed all of your types, therefore, on the `org.book.configuration` package.

For each generated class, JAXB creates JavaBean properties based on the contents of the corresponding XML schema complex type. If a complex type contains complex sub-elements, JAXB will generate bean properties to other JAXB classes, representing those complex types. See Figure 5-10 for a class diagram of your generated classes.

Notice in `ConfigurationType` that there are two bean properties: `uiSettings` and `userSettings`. The types for these correspond to `UiSettingsType` and `UserSettingsType`, respectively, matching the definition in your XML schema. Looking into `ColorType` and `PointType` in more detail (see Figure 5-11) shows how JAXB maps a complex element definition to a generated Java class.

As you can see from Figure 5-11, JAXB maps XML elements and attributes of complex types to JavaBean properties. The complex type `colorType` in the schema had four sub-elements, `red`, `blue`, `green`, and `alpha`. These were all mapped to JavaBean `int` properties. They were mapped to `int` because that was the type specified in their element definitions. Note that this type could be another generated class type, if the element definition specified another XML schema complex type.

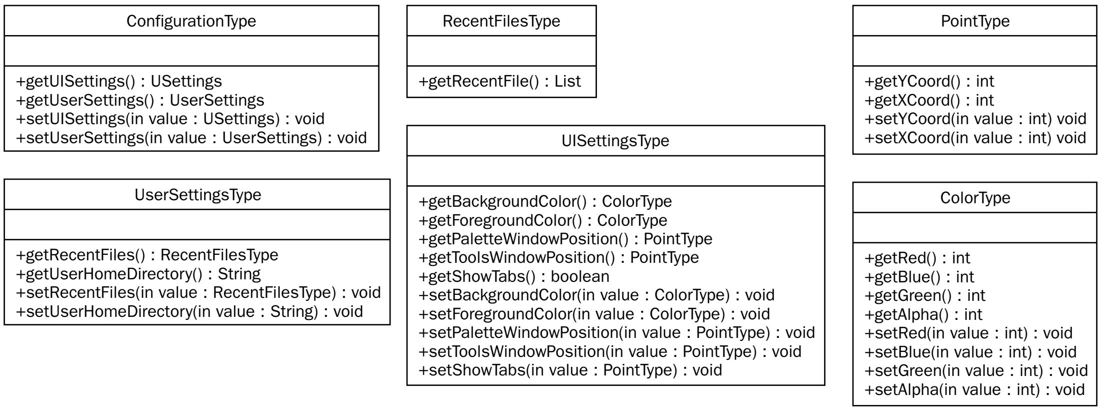
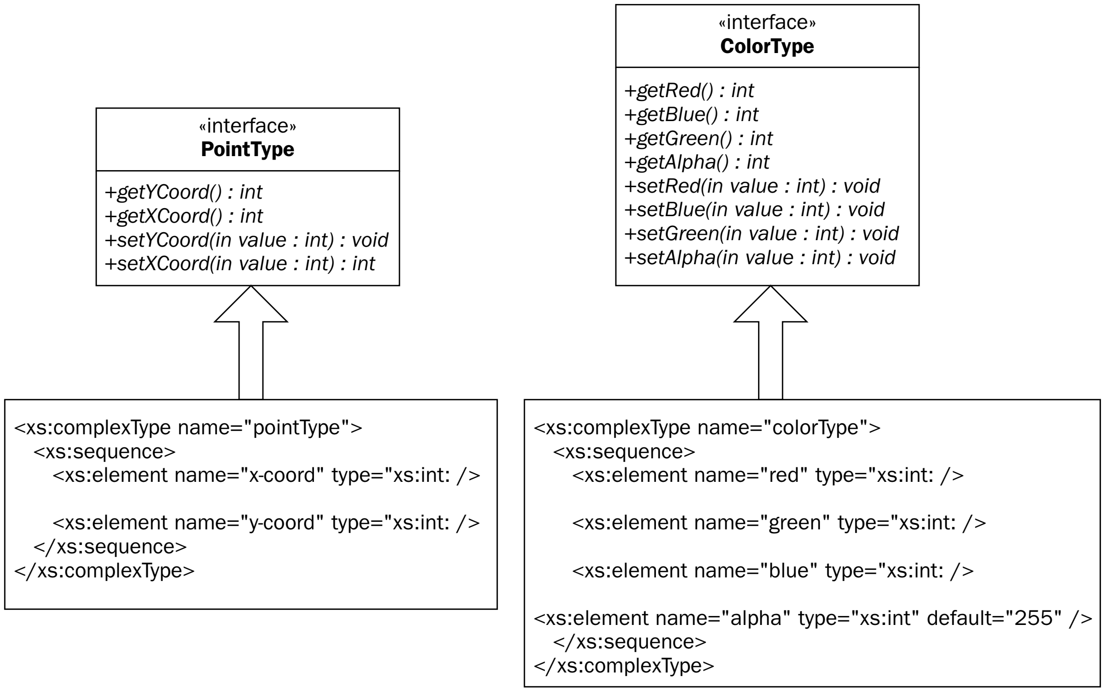

ConfigurationType
+getUISettings() : USettings
+getUserSettings() : UserSettings
+setUISettings(in value : USettings) : void
+setUserSettings(in value : UserSettings) : void
RecentFilesType
+getRecentFile() : List
PointType
+getYCoord() : int
+getXCoord() : int
+setYCoord(in value : int) void
+setXCoord(in value : int) void
UISettingsType
+getBackgroundColor() : ColorType
+getForegroundColor() : ColorType
+getPaletteWindowPosition() : PointType
+getToolsWindowPosition() : PointType
+getShowTabs() : boolean
+setBackgroundColor(in value : ColorType) : void
+setForegroundColor(in value : ColorType) : void
+setPaletteWindowPosition(in value : PointType) : void
+setToolsWindowPosition(in value : PointType) : void
+setShowTabs(in value : PointType) : void
UserSettingsType
+getRecentFiles() : RecentFilesType
+getUserHomeDirectory() : String
+setRecentFiles(in value : RecentFilesType) : void
+setUserHomeDirectory(in value : String) : void
ColorType
+getRed() : int
+getBlue() : int
+getGreen() : int
+getAlpha() : int
+setRed(in value : int) : void
+setBlue(in value : int) : void
+setGreen(in value : int) : void
+setAlpha(in value : int) : void

Figure 5-10


«interface»
PointType
+getYCoord() : int
+getXCoord() : int
+setYCoord(in value : int) : void
+setXCoord(in value : int) : int
«interface»
ColorType
+getRed() : int
+getBlue() : int
+getGreen() : int
+getAlpha() : int
+setRed(in value : int) : void
+setBlue(in value : int) : void
+setGreen(in value : int) : void
+setAlpha(in value : int) : void

```
<xs:complexType name="pointType">
  <xs:sequence>
    <xs:element name="x-coord" type="xs:int: />

    <xs:element name="y-coord" type="xs:int: />
  </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="colorType">
  <xs:sequence>
    <xs:element name="red" type="xs:int: />

    <xs:element name="green" type="xs:int: />

    <xs:element name="blue" type="xs:int: />

<xs:element name="alpha" type="xs:int" default="255" />
  </xs:sequence>
</xs:complexType>
```

Figure 5-11

Because XML documents are hierarchical in nature, the structure of the generated JAXB classes is also hierarchical. JAXB serializes and deserializes root elements in an XML document. This root element is the beginning of an object graph. The XML complex type `pointType`, for instance, has sub-elements `x-coord` and `y-coord`; they are therefore properties of `pointType`. In the generated JAXB class, these coordinates become properties of the `PointType` interface. Figure 5-12 shows the generated JAXB object graph from the root element of the configuration data model, `configuration`.

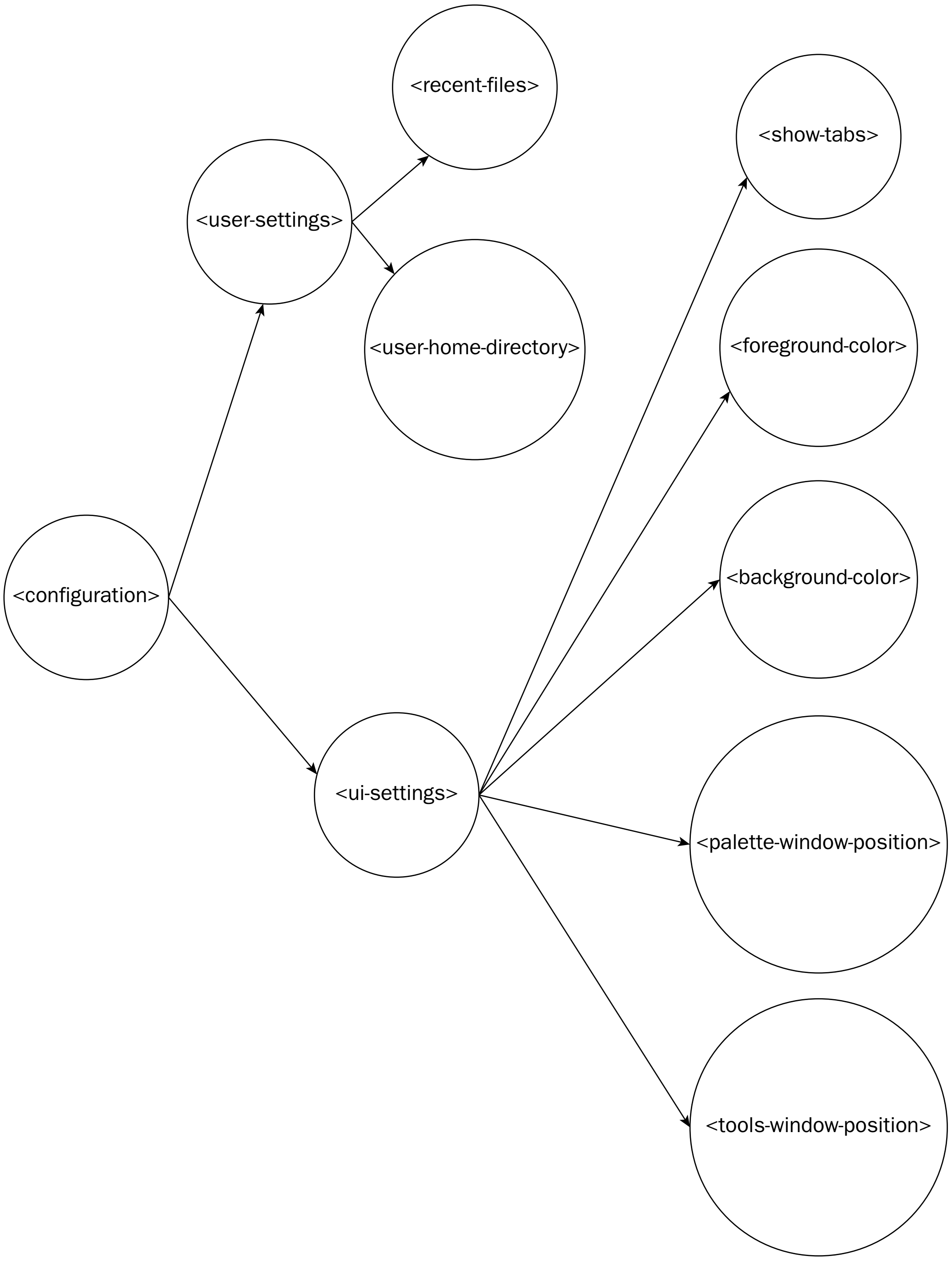


Figure 5-12

In XML Schema, elements that will be the root of an XML document must be specified as a top-level element definition in an XML schema. For this configuration example, the `configuration` element is the root element, and it is defined in XML schema as follows:

```
<xs:element name="configuration" type="configurationType" />
```

Because `configuration` is a top-level element and will be serialized as the root of an XML document, JAXB handles it a little differently. `ConfigurationType` is no different in structure than other JAXB-generated classes, but how it is serialized is different. This is where the `ObjectFactory` class generated by JAXB comes into play. JAXB uses the factory design pattern to create instances of its generated classes. For every set of classes it generates, it also generates a corresponding `ObjectFactory` class. This class contains methods to create instances of every Java type it generates. The `ObjectFactory` harkens back to the JAXB 1.0 days, when each complex type had both a Java `interface` and corresponding implementation class generated. The factory still serves a useful purpose, even when now in JAXB 2.0 all generated resources are actual concrete Java classes. This purpose is to wrap complex types *to be serialized as the root element of an XML document* in an instance of `javax.xml.bind.JAXBElement`.

Objects to be serialized as the root of an object graph *must* be wrapped in an instance of `javax.xml.bind.JAXBElement`. The generated `ObjectFactory` class makes this simple. When it comes time to serialize an instance of `ConfigurationType`, the `ObjectFactory` provides a clean and simple way to wrap the instance within a `JAXBElement` instance:

```
ConfigurationType ct = factory.createConfigurationType();
... set data properties...

JAXBElement<ConfigurationType> rootElement = factory.createConfiguration(ct);
```

The `rootElement` object is the one you will actually serialize and deserialize via JAXB. This is counter-intuitive when you're used to either the Java Serialization API or the XMLEncoder/Decoder API. With those APIs you just pass the actual Java object that is to be the root of the graph serialized — there is no need for a wrapper object. In the section "Annotating Existing Java Classes for Use with JAXB" you will see how wrapping an instance in `JAXBElement` can be circumvented to serialize it as root (through the use of the new annotations introduced with JAXB 2.0).

If you look at the source for any classes generated by JAXB, you will notice that all of the classes are annotated using the new annotations package in JAXB 2.0, `javax.xml.bind.annotation`*. These annotations are discussed in the "Annotating Existing Java Classes for Use with JAXB" section of this chapter.*

JAXB API Key Classes

While thus far this chapter has discussed the classes JAXB generates based on an XML schema, these generated classes are different from the classes used to actually serialize and deserialize them to and from XML. The JAXB runtime provides a couple of interfaces to perform the actual serialization operations, or conversion of Java class instances to their XML representation — `Marshaller` and `Unmarshaller`, both retrieved from the `JAXBContext`. `JAXBContext` instances must be set up using the Java class and package data of those classes to be serialized. From this information, the `JAXBContext` can create `Marshaller`s and `Unmarshaller`s that know the mapping between a particular set of Java classes and XML. `JAXBElement` must be used to wrap the root Java object being serialized or deserialized (unless you're using the `XmlRootElement` annotation — see the section "Annotating Existing Java Classes for Use with JAXB" for more information).

Class or Interface (from javax.xml.bind)	Function
`JAXBContext`	The `JAXBContext` is the initial class in which you create `Marshaller` and `Unmarshaller` classes for various JAXB-generated types
`JAXBElement<T>`	Used to represent a root element to be serialized, necessary to use if the root Java class of type `T` to be serialized is not marked with the annotation `XmlRootElement`
`Marshaller`	Interface that allows for the marshalling of JAXB-generated objects to XML in various formats (stream, DOM nodes, SAX events, and so forth)
`Unmarshaller`	Interface that allows for the unmarshalling of various XML representations (from a stream, a DOM tree, or SAX events), to populate instances of JAXB-generated classes

Marshalling and Unmarshalling XML Data

The process of marshalling and unmarshalling data to and from JAXB classes all occurs through three classes, `JAXBContext`, `Marshaller`, and `Unmarshaller`. Both `Marshaller` and `Unmarshaller` are created from an instance of `JAXBContext`, and they do the actual work of marshalling and unmarshalling the data. `JAXBContext` objects are configured upon creation, with the Java types that will be marshaled and unmarshaled. This allows the `JAXBContext` to set up the `Marshaller` and `Unmarshaller` objects with the particular schema rules and constraints for those classes that will be serialized. From the Java types passed in to configure the `JAXBContext`, the other dependent Java types will be automatically configured by the context (as long as they can be loaded by the same classloader) — therefore, only the root object of an object graph is necessary to configure the context. There are three steps to unmarshalling XML instance data conforming to a schema into your JAXB-generated object graph:

1. Retrieve an instance of `JAXBContext` specific the Java class(es) to be unmarshaled as the root of an XML document.
2. Create an `Unmarshaller` object from the `JAXBContext` instance.
3. Use the `Unmarshaller` to unmarshal XML data into instances of the generated JAXB classes.

Now you can unmarshal XML data conforming to the `configuration.xsd` schema into the generated JAXB classes. First, retrieve the `JAXBContext`, passing in `ConfigurationType.class` because that is the root Java object you are deserializing (meaning all other Java types being deserialized can be inferred from that type):

```
JAXBContext ctx = JAXBContext.newInstance(ConfigurationType.class);
```

Then create the `Unmarshaller` from the context:

```
Unmarshaller u = ctx.createUnmarshaller();
```

Now that you have an `Unmarshaller`, you can pass various streams, readers, and so forth, to access XML data to transform into an instance of the JAXB-generated object graph. In this example, you will pass it a `FileInputStream` corresponding to an XML file saved on disk that conforms to your schema. Notice how you cast the resulting `java.lang.Object` returned from the call to `unmarshal()` to `JAXBElement`. Using that object you can then get the `ConfigurationType` instance:

```
JAXBElement rootElement = (JAXBElement)
                            u.unmarshal(new FileInputStream("configuration.xml"));

org.book.configuration.ConfigurationType conf =
              (org.book.configuration.ConfigurationType)
                   rootElement.getValue();
```

Now you have the JAXB-generated `ConfigurationType` object, which represents the root node of your XML file, and is the root of your object graph. You can now use the data as necessary in your application. Marshalling data back into XML is just as simple as unmarshalling. The three steps to marshal the data mirror the three steps to unmarshal it:

1. Retrieve an instance of `JAXBContext` specific the Java class or classes to be marshaled as the root of an XML document.
2. Create a `Marshaller` object from the `JAXBContext` instance.
3. Use the `Marshaller` to marshal XML data into instances of the generated JAXB classes.

Now you can marshal instances of `org.book.configuration.ConfigurationType` back to disk (or to DOM representations or SAX events). Just like before, you get the `JAXBContext` particular for your package of JAXB-generated classes:

```
JAXBContext ctx = JAXBContext.newInstance(ConfigurationType.class);
```

Then you create the `Marshaller` from the context:

```
Marshaller m = ctx.createMarshaller();
```

Now you can use the `Marshaller` instance to serialize the information in the `conf` instance of `org .book.configuration.Configuration` to a `FileOutputStream`. Use the generated `ObjectFactory` class in `org.book.configuration` to create the `JAXBElement` wrapper for the root `ConfigurationType` object:

```
m.marshal(factory.createConfiguration(conf),
          new FileOutputStream("configuration.xml");
```

That's all there is to marshalling and unmarshalling data. As you can see, the difficult part is writing the schema.

If the `org.book.configuration.Configuration` *type is not populated with all the data the schema requires, the instance might be marshaled incorrectly to XML, meaning it will not be able to be unmarshaled back by JAXB. Exceptions will be thrown and the XML document will have to be fixed.*

Creating New XML Content with JAXB-Generated Classes

You have looked at how to load XML data into a JAXB object graph. You have looked into saving an existing JAXB object graph back into XML. How would you create a new JAXB graph and populate it programmatically? Unfortunately, this is one area where JAXB can become a little unwieldy. In JAXB, every set of generated classes comes with an `ObjectFactory` class at the root package of the generated classes. You may have noticed the class `org.book.configuration.ObjectFactory` back when you generated the set of classes for the `configuration.xsd` schema. This is the class you would use to create every JAXB object. The example that follows shows the creation and population of an `org.book.configuration.Configuration` instance:

```
ObjectFactory factory = new ObjectFactory();

ConfigurationType configType = factory.createConfiguration();
UiSettingsType uiSettingsType = factory.createUiSettingsType();
UserSettingsType userSettingsType = factory.createUserSettingsType();

configType.setUiSettings(uiSettingsType);
configType.setUserSettings(userSettingsType);
ColorType fgColorType = factory.createColorType();
fgColorType.setRed(255);
fgColorType.setBlue(255);
fgColorType.setGreen(0);

uiSettingsType.setForegroundColor(fgColorType);

uiSettingsType.setShowTabs(true);

userSettingsType.setUserHomeDirectory(conf.getUserHomeDirectory());

... // continue on as such, populating the entire object graph
```

One thing to take into consideration when manually populating JAXB object graphs, though, is completeness and conformance to the schema. Although it is easy to populate JAXB objects, and use the data in a Java application, if you want to save the data you are populating out to disk (or somewhere else) as XML, every schema-required piece of data must exist in your newly created object graph. In the preceding example, if you did not create a `UserSettingsType` instance and set it on the `Configuration` instance, you would get JAXB exceptions thrown if, after marshalling the data, you tried to unmarshal it back into your Java object.

Using JAXB-Generated Classes in Your Application

One of the potential issues that arise whenever information is saved and loaded from a file is that the information must be turned into objects used by the application. The nice thing about the Java Serialization API and XMLEncoder/Decoder is that they save the actual Java class instances used by an application, so there is no need to transform the data loaded into a format used by the application — it is already in the format used by the application. The classes that JAXB generates can be used as the in-memory data model for your application, but generally there is a need to perform at least some transformations. The Java classes in the JDK are rich and full of functionality — it would be wasteful to ignore it. Why store `URL`s as `String`s? Why store `File` objects as `String`s? Why not represent a color with a `java.awt.Color` object? Because it makes sense to use the classes in the JDK, a lot of the time you will find yourself taking data from the

JavaBeans generated by JAXB, and putting them into your own data structures. You'll find yourself adding JAXB classes to your own lists, maps, trees, and other data structures. This is the added burden of using JAXB with an existing schema over using Java Serialization or XMLEncoder/Decoder. In the example configuration data model used throughout this chapter, you used an instance of `book.Configuration` to represent the model. It contained Java representations of points and colors. To use the JAXB-generated configuration data model in your application, you will have to transform it to and from the `book.Configuration` data model. It's not a difficult task, but must be done for things like color and point representations to have any meaning to your application. The diagram in Figure 5-13 illustrates where transformations fit into the bigger picture of your application.

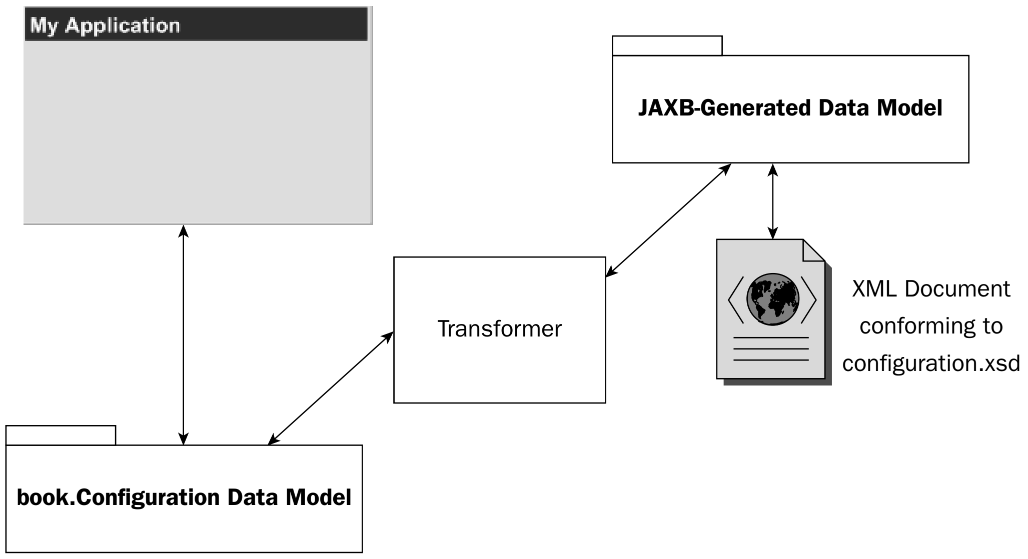

My Application
JAXB-Generated Data Model
Transformer
XML Document conforming to configuration.xsd
book.Configuration Data Model

Figure 5-13

In the original `Configuration` data model example, you wrapped your serialization code into Swing actions. This let you easily add code to save and load configuration data to menus and buttons in your application. You will do the same for code to save and load configuration data, this time with the XML format based on the `configuration.xsd` schema file. The key difference, though, will be that you need to integrate transformation functionality into these actions, because a conversion needs to be done between the JAXB-generated data model, and the original `Configuration` data model (as shown in Figure 5-13). Other than this transformation, the new XML save and load Swing actions will be very similar in structure and nature to the older actions.

Implementing the Save Action

The save action's `actionPerformed()` method will start out the same way as the original save action — by prompting the user for a file in which to save the configuration information:

```
package book;

...

import org.book.configuration.ColorType;
import org.book.configuration.ConfigurationType;
import org.book.configuration.ObjectFactory;
import org.book.configuration.PointType;
import org.book.configuration.RecentFilesType;
import org.book.configuration.UiSettingsType;
import org.book.configuration.UserSettingsType;

public class SaveXMLConfigurationAction extends AbstractAction {

  private Application myApp;

  public SaveXMLConfigurationAction(Application app) {
    super("Export XML Configuration");

    this.myApp = app;
  }

  public void actionPerformed(ActionEvent arg0) {
    JFileChooser fc = new JFileChooser();
    if (JFileChooser.APPROVE_OPTION == fc.showSaveDialog(myApp)) {
      try {
```

If the user chooses a file to save the configuration to, you get the application's `book.Configuration` object, and begin the process of transforming it to a `org.book.configuration.Configuration` object. Notice where the `JAXBContext` is created — it is created by programmatically getting the package name from `ConfigurationType` (which is in the `org.book.configuration` package). This is another method of creating a `JAXBContext`. By passing in the `String` for a Java package name, JAXB will keep all of the classes in the package in its context. Create the `ObjectFactory`, and then begin creating the `ConfigurationType` and mapping the information in the original `book.Configuration` object to the new JAXB `ConfigurationType`:

```
        Configuration conf = this.myApp.getConfiguration();

        JAXBContext ctx = JAXBContext.newInstance(
                           ConfigurationType.class.getPackage().getName());

        Marshaller m = ctx.createMarshaller();
        ObjectFactory factory = new ObjectFactory();

        ConfigurationType configType = factory.createConfigurationType();
        UiSettingsType uiSettingsType = factory.createUiSettingsType();
        UserSettingsType userSettingsType = factory.createUserSettingsType();

        configType.setUiSettings(uiSettingsType);
        configType.setUserSettings(userSettingsType);

        Color fgColor = conf.getForegroundColor();
        if (fgColor != null) {
          ColorType fgColorType = factory.createColorType();
          fgColorType.setRed(fgColor.getRed());
          fgColorType.setBlue(fgColor.getBlue());
          fgColorType.setGreen(fgColor.getGreen());
```

```
            fgColorType.setAlpha(fgColor.getAlpha());

            uiSettingsType.setForegroundColor(fgColorType);
          }

          Color bgColor = conf.getBackgroundColor();
          if (bgColor != null) {
            ColorType bgColorType = factory.createColorType();
            bgColorType.setRed(bgColor.getRed());
            bgColorType.setBlue(bgColor.getBlue());
            bgColorType.setGreen(bgColor.getGreen());
            bgColorType.setAlpha(bgColor.getAlpha());

            uiSettingsType.setBackgroundColor(bgColorType);
          }

          Point ppPoint = conf.getPaletteWindowPosition();
          if (ppPoint != null) {
            PointType ppPointType = factory.createPointType();
            ppPointType.setXCoord(ppPoint.x);
            ppPointType.setYCoord(ppPoint.y);

            uiSettingsType.setPaletteWindowPosition(ppPointType);
          }

          Point tpPoint = conf.getToolsWindowPosition();
          if (ppPoint != null) {
            PointType tpPointType = factory.createPointType();
            tpPointType.setXCoord(tpPoint.x);
            tpPointType.setYCoord(tpPoint.y);

            uiSettingsType.setToolsWindowPosition(tpPointType);
          }

          uiSettingsType.setShowTabs(conf.isShowTabs());

          userSettingsType.setUserHomeDirectory(conf.getUserHomeDirectory());
          String[] recentFiles = conf.getRecentFiles();
          if (recentFiles != null) {
            RecentFilesType rFilesType = factory.createRecentFilesType();

            Collections.addAll(rFilesType.getRecentFile(), recentFiles);

            userSettingsType.setRecentFiles(rFilesType);
          }
```

Finally, after you finish mapping the data, marshal it to the file specified by the user:

```

          m.marshal(factory.createConfiguration(configType),
                                 new FileOutputStream(fc.getSelectedFile()));

        } catch (IOException ioe) {
          JOptionPane.showMessageDialog(this.myApp, ioe.getMessage(), "Error",
                                                 JOptionPane.ERROR_MESSAGE);

          ioe.printStackTrace();

        } catch (JAXBException jaxbEx) {
          JOptionPane.showMessageDialog(this.myApp, jaxbEx.getMessage(), "Error",
                                                 JOptionPane.ERROR_MESSAGE);

          jaxbEx.printStackTrace();
        }
      }
    }
  }
```

Note how you must catch `JAXBException` in the preceding code. Most JAXB operations can throw a `JAXBException` — when saving it can mean that you did not populate all the information that was required in the generated object structure.

Implementing the Load Action

The load action is, of course, similar to the original load action — and probably most actions that load files. The user is prompted for a file from which to load the configuration:

```
package book;

...

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;

import org.book.configuration.ColorType;
import org.book.configuration.ConfigurationType;
import org.book.configuration.PointType;
import org.book.configuration.RecentFilesType;

public class LoadXMLConfigurationAction extends AbstractAction {

  private Application myApp;

  public LoadXMLConfigurationAction(Application app) {
    super("Import XML Configuration");
    this.myApp = app;
  }

  public void actionPerformed(ActionEvent evt) {
    JFileChooser fc = new JFileChooser();
    if (JFileChooser.APPROVE_OPTION == fc.showOpenDialog(myApp)) {
      try {
```

Once the user has picked the file, begin the process of unmarshalling the XML data contained in the file to the JAXB-generated data model. Once the data has been unmarshaled, you can begin the process of mapping the data from the JAXB model to the `book.Configuration` model. This is essentially the reverse process of what you did in the save action. You are converting things like the JAXB ColorType back into a form you can display in the Swing user interface, the java.awt.Color object. Other mappings are not quite as important, because in theory if the application had originally been built to use the JAXB data model instead of the book.Configuration model, you could access the user's home directory and other Java-primitive–based properties directly from the JAXB model. Unfortunately, you would still lose some benefit — the java.io.File class would better represent files than a String, and so forth. Some degree of this type of mapping will almost always be required when using any sort of generated code:

```
            JAXBContext ctx = JAXBContext.newInstance(ConfigurationType.class
                                                          .getPackage().getName());

            Unmarshaller u = ctx.createUnmarshaller();

            JAXBElement rootElement = (JAXBElement)
                             (JAXBElement) u.unmarshal(fc.getSelectedFile());

            org.book.configuration.Configuration configType =
                                  (org.book.configuration.Configuration)
                                      rootElement.getValue();

            Configuration conf = new Configuration();

            ColorType bgColorType = configType.getUiSettings().getBackgroundColor();
            if (bgColorType != null) {
              Color bgColor = new Color(bgColorType.getRed(),
                                        bgColorType.getGreen(), bgColorType.getBlue(),
                                        bgColorType.getAlpha());

              conf.setBackgroundColor(bgColor);
            }

            ColorType fgColorType = configType.getUiSettings().getForegroundColor();
            if (fgColorType != null) {
              Color fgColor = new Color(fgColorType.getRed(),
                                        fgColorType.getGreen(), fgColorType.getBlue(),
                                        fgColorType.getAlpha());

              conf.setForegroundColor(fgColor);
            }

            PointType ppPointType = configType.getUiSettings()
                                      .getPaletteWindowPosition();

            if (ppPointType != null) {
              Point ppPoint = new Point(ppPointType.getXCoord(),
                                             ppPointType.getYCoord());

              conf.setPaletteWindowPosition(ppPoint);
            }

            PointType tpPointType = configType.getUiSettings()
```

```
                                    .getToolsWindowPosition();

          if (tpPointType != null) {
            Point tpPoint = new Point(tpPointType.getXCoord(),
                                        tpPointType.getYCoord());

            conf.setToolsWindowPosition(tpPoint);
          }

          conf.setShowTabs(configType.getUiSettings().isShowTabs());

          conf.setUserHomeDirectory(
                            configType.getUserSettings().getUserHomeDirectory());

          RecentFilesType rFilesType =
                                  configType.getUserSettings().getRecentFiles();

          if (rFilesType != null) {
            List recentFileList = rFilesType.getRecentFile();
            if (recentFileList != null) {
              String[] recentFiles = new String[recentFileList.size()];

              recentFileList.toArray(recentFiles);

              conf.setRecentFiles(recentFiles);
            }
          }


          myApp.setConfiguration(conf);
        } catch (JAXBException jaxb) {
          JOptionPane.showMessageDialog(this.myApp, jaxb.getMessage(), "Error",
                                         JOptionPane.ERROR_MESSAGE);

          jaxb.printStackTrace();

        }
      }
    }

  }
```

Similar to the save action, you must also catch `JAXBException`. If an error occurs while loading the file — for example, it does not conform to the `configuration.xsd` schema, or the file could not be found — the exception will be thrown.

The Swing actions you just developed get integrated into your application the same way the previous ones did. Your application now has two mechanisms for persisting its configuration data model. One is user-friendly to edit, the other one cannot be edited outside of the application. The updated application is shown in Figure 5-14.

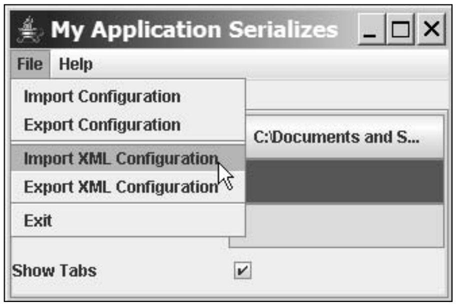


Figure 5-14

Annotating Existing Java Classes for Use with JAXB

As you can probably see from the previous example, the classes JAXB generates for you from an existing XML schema are not always that friendly to use within your application. Many times this is the case when working with a generated data model — there will always be some part of the data model generated that will need to be transformed into some more useful format, either for more efficient data traversal, or cases where conversion to some other type used in third-party libraries is necessary, like converting color values into the object actually used within AWT/Swing, `java.awt.Color`. These types of conversions can be tedious, and in the worst case, an application must work with two entirely separate in-memory object graphs, one to serialize and deserialize, and the other to actually use throughout the application. JAXB 2.0's annotations bridge this gap tremendously. By annotating existing Java objects, JAXB's scope widens significantly to include potentially any Java object for serialization. This is the best of both worlds in some respects — similar ease of serialization is like the Java Serialization API, but also the power to customize the output format.

A Simple Case

JAXB can handle the serialization of many existing Java classes, straight out of the box, without any annotation. For an object to be serialized as the root of an object graph, it must satisfy one of the following criteria:

- The class `javax.xml.bind.JAXBElement` must be used to wrap the object
- The class declaration must have the annotation `javax.xml.bind.annotation.XmlRootElement`

Looking back at the simple `MyPoint` class, both methods of serialization will be illustrated. Notice that the actual serialization code is identical to the previous JAXB classes, and similar to the API patterns found in the Java Serialization API and XMLEncoder/Decoder. The class `MyPoint`, without any JAXB annotations, is shown here:

```
package book;

public class MyPoint {
  public int x;
  public int y;
}
```

Because there is no `XmlRootElement` annotation on the class, it must be wrapped with `JAXBElement` to serialize properly. By simply creating a parameterized `JAXBElement`, you can assign the element name and namespace, and marshal the object to XML:

```
    JAXBContext ctx = JAXBContext.newInstance(MyPoint.class);

    Marshaller m = ctx.createMarshaller();
    m.setProperty("jaxb.formatted.output", true);

    MyPoint p = new MyPoint();
    p.x = 50;
    p.y = 75;

    JAXBElement<MyPoint> root = new JAXBElement<MyPoint>
                               (new QName("my-point"),
                                    MyPoint.class, p);

    m.marshal(root, System.out);
```

The second option to serialize the class would be to add the `XmlRootElement` annotation to the class declaration. Doing so allows instances of the class to be serialized using JAXB without using a `JAXBElement` wrapper if it is to be the root element. If you attempt to serialize a class that does not have the `XmlRootElement` annotation as the root element, JAXB will throw an exception and the serialization will not occur. Here is the slightly modified `MyPoint` class, annotated with `XmlRootElement`:

```
package book;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name="my-point")
public class MyPoint {
  public int x;
  public int y;
}
```

Serialization follows exactly as illustrated in the previous JAXB example, without the need to use an instance of `JAXBElement` for the root element:

```
    JAXBContext ctx = JAXBContext.newInstance(MyPoint.class);

    Marshaller m = ctx.createMarshaller();
    m.setProperty("jaxb.formatted.output", true);

    MyPoint p = new MyPoint();
    p.x = 50;
    p.y = 75;

    m.marshal(p, System.out);
```

If you wanted to serialize `MyPoint` *as part of another class — for example, it was a field or JavaBean property in another class — it would not have to be wrapped in* `JAXBElement` *nor have an* `XmlRootElement` *annotation. Also note the* `jaxb.formatted.output` *property; when set to* `true`*, the* `Marshaller` *object formats the XML with indentions and line breaks, making it more human-readable.*

The XML generated is the same for both simple examples and looks like the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<my-point>
    <x>50</x>
    <y>75</y>
</my-point>
```

JAXB API Key Annotations

With other JAXB annotations, the XML output can be customized. Some annotations can only be placed on a class declaration, such as `XmlType` and `XmlRootElement`. Annotations like `XmlAttribute` and `XmlElement` can only be placed on fields or JavaBean properties. The annotations map directly to XML Schema constructs, and an understanding of XML Schema is required to fully understand how all of the annotations in the `javax.xml.bind.annotation` package should be used. The annotations listed in the following table, though, can be understood enough without in-depth knowledge of XML Schema to be used in applications requiring simple XML serialization.

Annotations (from javax.xml.bind.annotation)	**Function**
`XmlAttribute`	Maps a field or JavaBean property to an XML attribute. Note that the Java type of the field or property being annotated must correspond to an XML schema simple type — a Java String, Boolean, integer, and so on — or a Java class annotated to make it map to a simple type.
`XmlElement`	Maps a non-static and non-transient field or JavaBean property to an XML element. If the Java type of the field or JavaBean property is another Java class, that class must be serializable by JAXB (annotated appropriately if necessary).
`XmlElementWrapper`	Can be used on any field or JavaBean property where `XmlElement` can be used. It works with `XmlElement` usually, but can also work with `XmlJavaTypeAdapter`. `XmlElementWrapper` puts another XML element around whatever element the field or property maps to (wrapping the element). It is mainly intended for use when serializing a field or property representing a collection of elements (such as a `List`).
`XmlID`	Can only be used on a field or bean property of type `java.lang.String`. It signifies the key field of an XML element, used when referring back to an element using the `XmlIDREF` annotation (XML schema ID and IDREF concept).
`XmlIDREF`	Placed on a field or bean property and changes how the Java type is serialized. Instead of serializing the whole type, only the key field (specified on that class via XmlID) is serialized — allowing the same instance of an object to be serialized multiple times in the document, but only listing the entire element once, with subsequent elements simply referring back to the original via the `XmlIDREF` annotation.

Annotations (from javax.xml.bind.annotation)	Function
`XmlJavaTypeAdapter`	Used on a class declaration, or a field or bean property declaration. It is used when the Java type to be serialized by JAXB does not meet the minimum requirements for serialization and must be adapted (by an adapter class). Every class serialized by JAXB must have a default no-arg constructor, or classes that do not match well to serialization will have to be adapted for use with JAXB.
`XmlRootElement`	Any class that is to be serialized as the root XML element must have the `XmlRootElement` annotation on its class declaration, or be serialized through the use of `JAXBElement`.
`XmlTransient`	By annotating a field or JavaBean property with `XmlTransient`, that field will not be serialized by JAXB (analogous to the Java `transient` keyword used in Java Serialization).
`XmlType`	Used to annotate a class declaration. It is used to annotate a class as being a complex type in XML Schema. Here the serialization order of fields can be specified (XML Schema's sequence), or simple anonymous types can be declared (when used in conjunction with the `XmlValue` annotation).
`XmlValue`	Used to represent the XML element content of an XML schema `simpleContent` type. Only one field or bean property can be annotated with `XmlValue` in a class, and the only other annotation allowed on other fields or bean properties is `XmlAttribute` (and any fields or bean properties not marked with `XmlAttribute` should be marked `XmlTransient`).

Annotating the Data Model

Using JAXB's powerful annotations, you can annotate the existing configuration data model. By annotating this data model, your application can avoid the problem of having two separate data models. The same data model used throughout the application can simply be serialized without the need to transform it to a different set of classes essentially used only for JAXB. Annotating existing Java classes is a great way to generate XML application configuration files. Just by annotating whatever Java classes hold the data for an application's configuration with JAXB annotations, you can easily create XML configuration files without the need for writing SAX handlers or traversing DOM trees with lower level XML APIs. Annotating the `book.Configuration` class yields the following:

```
... (imports) ...

@XmlRootElement(name="configuration", namespace="http://book.org/Configuration")
@XmlType(name="configurationType", namespace="http://book.org/Configuration",
    propOrder={"showTabs","backgroundColor","foregroundColor","recentFiles"})
@XmlAccessorType(value=XmlAccessType.PUBLIC_MEMBER)
```

```
public class Configuration {

  ... (un-annotated private fields)

  private List<File> recentFiles;

  @XmlAttribute(name="user-home-directory",
      namespace="http://book.org/Configuration")
  public String getUserHomeDirectory() {
    return userHomeDirectory;
  }

  public void setUserHomeDirectory(String userHomeDirectory) {
    this.userHomeDirectory = userHomeDirectory;
  }

  public boolean isShowTabs() {
    return showTabs;
  }

  @XmlElement(name="show-tabs",
      namespace="http://book.org/Configuration")
  public void setShowTabs(boolean showTabs) {
    this.showTabs = showTabs;
  }

  @XmlElementWrapper(name="recent-files",
      namespace="http://book.org/Configuration")
  @XmlElement(name="file",
      namespace="http://book.org/Configuration")
  public String[] getRecentFiles() {
...
  }

  public void setRecentFiles(String[] files) {
...
  }

  public Color getBackgroundColor() {
    return backgroundColor;
  }

  @XmlElement(name="background-color",
      namespace="http://book.org/Configuration")
  @XmlJavaTypeAdapter(value=ColorAdapter.class)
  public void setBackgroundColor(Color backgroundColor) {
    this.backgroundColor = backgroundColor;
  }

  @XmlElement(name="foreground-color",
      namespace="http://book.org/Configuration")
  @XmlJavaTypeAdapter(value=ColorAdapter.class)
  public Color getForegroundColor() {
    return foregroundColor;
    }

    public void setForegroundColor(Color foregroundColor) {
      this.foregroundColor = foregroundColor;
    }

    @XmlTransient
    public Point getPaletteWindowPosition() {
      return paletteWindowPosition;
    }

    public void setPaletteWindowPosition(Point paletteWindowPosition) {
      this.paletteWindowPosition = paletteWindowPosition;
    }


    @XmlTransient
    public Point getToolsWindowPosition() {
      return toolsWindowPosition;
    }

    public void setToolsWindowPosition(Point toolsWindowPosition) {
      this.toolsWindowPosition = toolsWindowPosition;
    }
  }
```

The Class Declaration: XmlRootElement, XmlType, and XmlAccessorType

Now after seeing the newly annotated `book.Configuration` class all at once, you can analyze the various annotations present. Start with the annotations on the class declaration. Previously, this chapter discussed the `XmlRootElement` attribute. It is necessary to annotate a class or interface declaration with `XmlRootElement` if you want to serialize instances of the class as the root object of the object graph. It is possible to marshal a class as the root of a serialization, without `XmlRootElement`, just by wrapping the instance to serialize as root within the parameterized-type `JAXBElement`:

```
@XmlRootElement(name="configuration", namespace="http://book.org/Configuration")
@XmlType(name="configurationType", namespace="http://book.org/Configuration",
    propOrder={"showTabs","backgroundColor","foregroundColor","recentFiles"})
@XmlAccessorType(value=XmlAccessType.PUBLIC_MEMBER)
public class Configuration {
```

Note the two values passed in to the `XmlRootElement` annotation, `name` and `namespace`. They represent the element name and its corresponding namespace for the class when it is serialized as the root element. If `name` and `namespace` are not specified, the default XML namespace will be used, and the name of the element will follow JavaBeans conventions and be generated from the Java class name. The name and namespace can change if the class is serialized in other places besides the root element, as you will see, because name and namespace can also be specified in the `XmlElement` attribute.

Many annotations in JAXB are implicit if they are not specified. `XmlType` is one of them. Every Java type to be serialized is automatically assumed to be annotated as an `XmlType`. `XmlType` maps the Java type to the two different XML schema types, simple or complex. In this example, you specify the name of the complex XML schema type you are generating, its namespace, and its property order (via the `propOrder` value). The `propOrder` value takes an array of `String`s, representing the names of the field names and bean property names in the class. The order in which these names appear in the `propOrder` value dictates the order in which they appear in the output XML (creating a sequence in XML Schema). Sample XML output from serializing an instance of this class looks like the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration xmlns="http://book.org/Configuration" user-home-
directory="C:\Documents and Settings\Mark\My Documents">
    <show-tabs>true</show-tabs>
    <background-color>
        <red>204</red>
        <green>0</green>
        <blue>153</blue>
        <alpha>255</alpha>
    </background-color>
    <foreground-color>
        <red>51</red>
        <green>51</green>
        <blue>51</blue>
        <alpha>255</alpha>
    </foreground-color>
    <recent-files>
        <file>test.xml</file>
        <file>test2.xml</file>
    </recent-files>
</configuration>
```

As you can see, the order of the XML elements in this sample output matches the order specified in the `XmlType` annotation's `propOrder` value. The namespace of the entire document is the same, and JAXB gave it the default XML prefix for the document.

`XmlType` can represent both simple and complex XML schema types. When one field or bean property in the class has the `XmlValue` attribute, the `XmlType` declaration automatically assumes an XML schema simple content type (and the property marked with `XmlValue` is the content). When `XmlValue` is used, no other field or bean property can be marked as an `XmlElement`, only with `XmlAttribute`. `XmlType` can represent other complex element declarations — see its JavaDoc for more information.

Now to talk about the annotation you noticed on the class declaration, which was strangely absent from the previous table of key JAXB annotations — the `XmlAccessorType` annotation. This is an important annotation, but one you will probably find yourself rarely using. Its value takes the `XmlAccessType` enumeration. The values for this enumeration are `FIELD`, `NONE`, `PROPERTY`, and `PUBLIC_MEMBER`. These values define the scope of JAXB serialization for the class. `XmlAccessorType` is an optional annotation, and it defaults to `PUBLIC_MEMBER` if it is not specified. It is redundantly specified in `book.Configuration` class to aid in its explanation. For the `PUBLIC_MEMBER` `XmlAccessType`, JAXB serializes all public fields and JavaBean properties (even if they are not explicitly annotated). That is why the `MyPoint` class serialized the way it did; with no annotations whatsoever, it was using the default `PUBLIC_MEMBER` `XmlAccessType`. The other `XmlAccessType` values are straightforward. `FIELD` specifies that every non-static, non-transient field will be serialized (including private and protected fields). `NONE` indicates fields or bean properties to be serialized must be explicitly annotated. Finally, the PROPERTY value indicates that all JavaBean properties will be serialized by default. Note that the XmlAccessorType annotation merely tells JAXB which fields and properties to serialize by default — fields or bean properties explicitly marked for serialization will always be serialized.

XmlElement and XmlAttribute

The next couple annotations are the simplest, and probably the ones you will find yourself most often using. The following code segment shows XmlAttribute and XmlElement in action:

```
@XmlAttribute(name="user-home-directory",
    namespace="http://book.org/Configuration")
public String getUserHomeDirectory() {
  return userHomeDirectory;
}

public void setUserHomeDirectory(String userHomeDirectory) {
  this.userHomeDirectory = userHomeDirectory;
}

public boolean isShowTabs() {
  return showTabs;
}

  @XmlElement(name="show-tabs",
    namespace="http://book.org/Configuration")
public void setShowTabs(boolean showTabs) {
  this.showTabs = showTabs;
}
```

Notice how only one of the getter/setter pairs is actually annotated for each JavaBean property, userHomeDirectory and showTabs. To annotate JavaBean properties, either the getter or the setter can be annotated, but not both. XmlAttribute and XmlElement have the name and namespace values. These correspond to the XML element's local name and namespace (just as they did in the XmlRootElement attribute). Note that for XmlAttribute, the Java type of the field or bean property must be an XML schema simple type (because it must fit into an XML attribute). The XmlElement annotation does not have this limitation, and can be placed on any Java type valid for JAXB.

XmlElementWrapper

Java collection classes can also be serialized by JAXB. The most common case for collection class serialization is a list or array. In the book.Configuration class, you store a list of the most recent files the user has last accessed while using your application. Annotating this for serialization in JAXB looks like the following:

```
  @XmlElementWrapper(name="recent-files",
      namespace="http://book.org/Configuration")
  @XmlElement(name="file",
      namespace="http://book.org/Configuration")
  public String[] getRecentFiles() {
...
```

```
     }

     public void setRecentFiles(String[] files) {
   ...
     }
```

When `XmlElement` is applied to a collection, JAXB applies it to every object in the collection. When applied to the `String` array for example, each `String` in the array is represented as an XML element, with the name of `"file"`. Often when serializing a list of similar items in XML, convention is to wrap all of the similar elements in one larger element, increasing the readability of the document. The `XmlElementWrapper` is precisely for this purpose. It wraps the entire collection in one element; in this case, `recent-files`. Output XML from using the `XmlElementWrapper` looks like the following:

```
       <recent-files>
           <file>test.xml</file>
           <file>test2.xml</file>
       </recent-files>
```

XmlJavaTypeAdapter

Sometimes JAXB cannot serialize a particular Java type. Classes to be serialized must have a default constructor. Other times classes such as `java.util.HashMap` do not serialize naturally. The `XmlJavaTypeAdapter` annotation allows for custom marshalling and can be applied at the field or bean property level, the class level, or even the package level. `XmlJavaTypeAdapter` maps a Java class to another Java class, one that does serialize well under JAXB. Note how this process is similar to the data transformation discussed previously, when you converted the `book.Configuration` data model to an entirely separate data model, the one generated by JAXB. One advantage to `XmlJavaTypeAdpater` over such a strategy is that your data model transformations can be fine grained, at the object level (and not the entire data model). Maybe one class in an otherwise huge data model does not properly serialize. You would only have to use `XmlJavaTypeAdapter` for that one class, and not transform your entire model:

```
     public Color getBackgroundColor() {
       return backgroundColor;
     }

     @XmlElement(name="background-color",
         namespace="http://book.org/Configuration")
     @XmlJavaTypeAdapter(value=ColorAdapter.class)
     public void setBackgroundColor(Color backgroundColor) {
       this.backgroundColor = backgroundColor;
     }
```

The `XmlJavaTypeAdapter` annotation takes as its value a `Class` type. The class passed in must extend `javax.xml.bind.annotation.adapters.XmlAdapter`. The type passed in that extends `XmlAdapter` does the work of transforming an object of whatever type the `XmlJavaTypeAdapter` was placed on (in this example, `java.awt.Color`), to a type JAXB can serialize. In this example, you chose to use `XmlJavaTypeAdapter` on `java.awt.Color` because `Color` has no default constructor. The parameterized class `XmlAdapter<ValueType,BoundType>` has two abstract methods that must be overridden:

```
   ValueType marshal(BoundType b)
   BoundType unmarshal(ValueType v)
```

The BoundType refers to the Java class that cannot be serialized, the one that is being bound to XML — in this case it is java.awt.Color. The ValueType refers to the type you will transform the bound type to and from, the type that JAXB can understand and properly serialize. As you can see, these two methods you must implement give you a lot of freedom of how to serialize a particular Java type to XML. It is a more fine-grained method of data model transformation to a format more suitable for serialization. There will always be those times where a particular class has a data structure well suited to memory access that is simply incompatible with serialization. The code for the custom ColorAdapter class is as follows:

```
package jaxb2;

import java.awt.Color;
import javax.xml.bind.annotation.adapters.XmlAdapter;
import org.book.configuration.ColorType;

public class ColorAdapter extends XmlAdapter<ColorType,Color> {

  @Override
  public Color unmarshal(ColorType ct) throws Exception {
    return new Color(ct.getRed(), ct.getGreen(), ct.getBlue(), ct.getAlpha());
  }

  @Override
  public ColorType marshal(Color c) throws Exception {
    ColorType ct = new ColorType();

    ct.setAlpha(c.getAlpha());
    ct.setBlue(c.getBlue());
    ct.setGreen(c.getGreen());
    ct.setRed(c.getRed());

    return ct;
  }

}
```

Notice the use of org.book.configuration.ColorType for the JAXB-suitable ValueType. The code was generated for it from the previous example of generating JAXB classes from an XML schema. You could have just as easily created it yourself, though, for this one small use. The code for ColorType is properly annotated, and the class has a default constructor:

```
... (imports) ...

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "colorType", propOrder = {
    "red",
    "green",
    "blue",
    "alpha"
})
public class ColorType {

    @XmlElement(namespace = "http://book.org/Configuration", type = Integer.class)
    protected int red;
    @XmlElement(namespace = "http://book.org/Configuration", type = Integer.class)
    protected int green;
    @XmlElement(namespace = "http://book.org/Configuration", type = Integer.class)
    protected int blue;
    @XmlElement(namespace = "http://book.org/Configuration", type = Integer.class,
defaultValue = "255")
    protected int alpha;

... (Java Bean properties for the above fields) ...

}
```

In the JAXB-suitable `ColorType` class you have defined all the attributes necessary to reproduce a `java.awt.Color` object — its RGBA values. When the `java.awt.Color` object is to be marshaled to XML, the `marshal(BoundType b)` method is called, and the `ColorAdapter` class creates a new, JAXB-suitable `ColorType` instance, and sets the appropriate RGBA values. The process is reversed during unmarshalling. The XML generated from the `ColorType` class looks like the following:

```
    <background-color>
        <red>204</red>
        <green>0</green>
        <blue>153</blue>
        <alpha>255</alpha>
    </background-color>
```

Notice how the name of the element is determined by the `XmlElement` annotation placed on the bean property (the `backgroundColor` property in this case). `XmlJavaTypeAdapter` does not determine the name of the element or the namespace — that is all determined by the `XmlElement` attribute, just as with any non-adapted bean property or field.

XmlTransient

`XmlTransient` is used much like the `transient` keyword in Java. Fields or bean properties marked with `XmlTransient` will be ignored by JAXB. In this example, you marked the `paletteWindowPosition` and the `toolsWindowPosition` bean properties as `XmlTransient`. In the output XML, these properties did not appear:

```
  @XmlTransient
  public Point getPaletteWindowPosition() {
    return paletteWindowPosition;
  }

  public void setPaletteWindowPosition(Point paletteWindowPosition) {
    this.paletteWindowPosition = paletteWindowPosition;
  }


  @XmlTransient
  public Point getToolsWindowPosition() {
    return toolsWindowPosition;
  }

  public void setToolsWindowPosition(Point toolsWindowPosition) {
    this.toolsWindowPosition = toolsWindowPosition;
  }
```

One of the reasons you chose to mark your properties of type `java.awt.Point` with `XmlTransient` is because `java.awt.Point` must be adapted to serialize properly. If you wanted to serialize `java.awt.Point` you would have had to create an `XmlJavaTypeAdapter` annotation on these properties, and create another subclass of `XmlAdapter` just like with `java.awt.Color`. The reason `java.awt.Point` cannot be serialized by JAXB is that one of its bean properties refers to itself (see `Point.getLocation()`), which creates an infinite loop during JAXB marshalling.

Generating an XML Schema from JAXB Annotated Classes

Now that the `book.Configuration` data model is annotated, you can generate its corresponding XML schema. You can distribute the generated schema to third parties who need to read the data you will be serializing to that format. Third parties then have a blueprint of the XML document type — they can either write an XML parser to parse it, or use a mechanism like JAXB for their programming language and platform (.NET, for instance, comes with an XML schema tool that generates classes from a schema). The JAXB `schemagen` tool generates XML schemas from a Java class and can be found in the `<JDK 6 Home>/bin` directory. Its usage is simple (assuming your classpath is correctly set up to find all the classes referenced by `book.Configuration`):

```
schemagen book.Configuration
```

Running this command creates two schemas:

```
schema1.xsd:

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" targetNamespace="http://book.org/Configuration"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:import schemaLocation="schema2.xsd"/>

  <xs:element name="alpha" type="xs:int"/>

  <xs:element name="blue" type="xs:int"/>

  <xs:element name="green" type="xs:int"/>

  <xs:element name="red" type="xs:int"/>

  <xs:element name="configuration" type="ns1:configurationType"
xmlns:ns1="http://book.org/Configuration"/>

  <xs:complexType name="configurationType">
    <xs:sequence>
      <xs:element name="show-tabs" type="xs:boolean" form="qualified"/>
      <xs:element name="background-color" type="colorType" form="qualified"
minOccurs="0"/>
      <xs:element name="foreground-color" type="colorType" form="qualified"
minOccurs="0"/>
      <xs:element form="qualified" name="recent-files" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="file" type="xs:string" form="qualified"
maxOccurs="unbounded" minOccurs="0"/>
```

```
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="user-home-directory" type="xs:string" form="qualified"/>
  </xs:complexType>

  <xs:attribute name="user-home-directory" type="xs:string"/>
</xs:schema>

schema2.xsd:

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:ns1="http://book.org/Configuration"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:import namespace="http://book.org/Configuration"
schemaLocation="schema1.xsd"/>

  <xs:complexType name="colorType">
    <xs:sequence>
      <xs:element ref="ns1:red" minOccurs="0"/>
      <xs:element ref="ns1:green" minOccurs="0"/>
      <xs:element ref="ns1:blue" minOccurs="0"/>
      <xs:element ref="ns1:alpha" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

These schemas could now be given to a third party to read the `book.Configuration` file format.

JAXB Pitfalls

There are a couple things to be wary of when annotating your classes with JAXB annotations. Usage of JAXB can be similar to usage of the Java Serialization API or the XMLEncoder/Decoder API, and this can lead to misconceptions of the capabilities of JAXB.

JAXB Serializes by Value

By default, all objects in an object graph are serialized by value. This is very different from the Java Serialization API and the XMLEncoder/Decoder API, which keep the referential integrity of an object graph being serialized by serializing multiple references to the same object only once. In these APIs, when two copies of the same object instance are saved, the object is only actually saved the first time, and all other references pointing back to that instance are saved as references, not as another duplicate copy of the object. When deserializing object graphs from the Java Serialization API or the XMLEncoder/Decoder API, multiple references to the same object instance will be returned as such and the original referential integrity of the object graph will be kept intact. Just as a simple demonstration of these concepts, serialize a slightly modified version of `MyPoint`, which has been changed to make its public fields private, and make them accessible via getters/setters to follow JavaBeans conventions, allowing XMLEncoder/Decoder to property serialize them. Serialize the following class, `PointContainer`, using both JAXB and XMLEncoder/Decoder:

```
    @XmlRootElement(name="point-container")
    public static class PointContainer {
      private MyPoint pointA;
      private MyPoint pointB;

      public MyPoint getPointA() {
        return pointA;
      }
      public void setPointA(MyPoint pointA) {
        this.pointA = pointA;
      }
      public MyPoint getPointB() {
        return pointB;
      }
      public void setPointB(MyPoint pointB) {
        this.pointB = pointB;
      }
    }
```

The serialization code will set the same `MyPoint` instance to both of `PointContainer`'s bean properties, `pointA` and `pointB`:

```
      JAXBContext ctx = JAXBContext.newInstance(MyPoint.class, PointContainer.class);

      Marshaller m = ctx.createMarshaller();
      m.setProperty("jaxb.formatted.output", true);

      MyPoint p = new MyPoint();
      p.setX(50);
      p.setY(75);

      PointContainer c = new PointContainer();
      c.setPointA(p);
      c.setPointB(p);

      m.marshal(c, System.out);

      XMLEncoder encoder = new XMLEncoder(System.out);
      encoder.writeObject(c);

      encoder.close();
```

The XML output for JAXB looks like the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<point-container>
    <pointA>
        <x>50</x>
        <y>75</y>
    </pointA>
    <pointB>
        <x>50</x>
        <y>75</y>
    </pointB>
</ point-container >
```

Notice how in JAXB the `MyPoint` instance is serialized by value twice in the output, when it is actually the same instance in the previous serialization code. If you were to deserialize this XML using JAXB, you would actually get two separate distinct instances of `MyPoint`, one for the `pointA` property and one for the `pointB` property. XMLEncoder/Decoder, on the other hand, produces the following XML output:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0-beta2" class="java.beans.XMLDecoder">
 <object class="book.MyPoint$PointContainer">
  <void property="pointA">
   <object id="MyPoint0" class="book.MyPoint">
    <void property="x">
     <int>50</int>
    </void>
    <void property="y">
     <int>75</int>
    </void>
   </object>
  </void>
  <void property="pointB">
   <object idref="MyPoint0"/>
  </void>
 </object>
</java>
```

In the XML output, the second reference to the same instance of `MyPoint`, `pointB`, simply refers to the already serialized `pointA` MyPoint instance (because they were the same instance in the serialization code). Both XMLEncoder/Decoder and the Java Serialization API handle this appropriately: they keep the object graph's referential integrity intact. JAXB does not do this by default, because it is not intended to exactly save Java object graphs, but merely bind them to and from an XML representation based on XML Schema. It is possible to keep the referential integrity intact, though, if your application requires it. To keep an object graph's referential integrity intact using JAXB requires some manual work, using the XML schema constructs of `XML ID` and `XML IDREF`. The main strategy with using `ID` and `IDREF` is to first serialize the full XML output for a given element, and then later on in the XML document, refer to that element by its ID. By manually forcing your JAXB classes into this model, you can enforce referential integrity for those areas of your object graph that require it. Modifying the `MyPoint` class and the `PointContainer` class, you can make the `pointA` and `pointB` fields of `PointContainer` serialize by reference and not by value. By annotating `pointA` and `pointB` with `XmlIDREF`, they will be serialized only by their key. The `XmlID` annotation identifies the key in the given type. In the code that follows, `MyPoint.key` is your key. Whatever key values `pointA` and `pointB` have must exist somewhere else in your XML document for proper deserialization to occur. Serialization will still occur if `pointA` and `pointB` have key values referring to nonexistent elements in your XML document. In this case though, `pointA` and `pointB` will not deserialize properly; they will only have their key field set (because the element they refer to does not exist in the document). In this code example, all points will first be added to be serialized by value in the `pointList` field. Then `pointA` and `pointB` will reference points in the list:

```
@XmlRootElement
public class MyPoint {
  public int x;
  public int y;

  @XmlID
  public String key;
  }

    @XmlRootElement(name="point-container")
    public class PointContainer {
      @XmlElementWrapper(name="all-points")
      @XmlElement(name="point")
      public List<MyPoint> pointList = new ArrayList<MyPoint>();

      @XmlIDREF
      public MyPoint pointA;

      @XmlIDREF
      public MyPoint pointB;
    }
```

Note that the `XmlID` tag can only be applied to a field or property of the `String` data type. Now you can serialize `pointA` and `pointB` by reference with the following code:

```
      JAXBContext ctx = JAXBContext.newInstance(MyPoint.class, PointContainer.class);

      Marshaller m = ctx.createMarshaller();
      m.setProperty("jaxb.formatted.output", true);

      MyPoint p = new MyPoint();
      p.key = "MyPointKey:1";
      p.x = 50;
      p.y = 75;

      PointContainer c = new PointContainer();
      c.pointList.add(p);
      c.pointA = p;
      c.pointB = p;

      m.marshal(c, System.out);
```

The XML output uses the key field from `MyPoint` in the serialization of `pointA` and `pointB`:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<point-container>
    <all-points>
        <point>
            <key>MyPointKey:1</key>
            <x>50</x>
            <y>75</y>
        </point>
    </all-points>
    <pointA>MyPointKey:1</pointA>
    <pointB>MyPointKey:1</pointB>
</point-container>
```

There are still dangers in relying on `XmlID` *and* `XmlIDREF` *to enforce referential integrity of an object graph. JAXB will not check to make sure the object referred to by an* `XmlIDREF` *actually exists in the document. If it does not, when the object is deserialized, it will only have its key set! Because referential integrity is enforced via the* `XmlID` *tag (whatever it annotates becomes the key), it is still up to the client programmer to keep track of the keys and make sure they all match up correctly.*

XmlJavaTypeAdapter as the Root of Serialization

When XmlJavaTypeAdapter is applied to a class definition, JAXB serializes all instances of the class using the adapter, with one notable exception. When the root object to be serialized is adapted with XmlJavaTypeAdapter, JAXB will not use the adapter — and will error out if the object by default will not serialize with JAXB (if the class does not have a default no-arg constructor, for example). To serialize these objects as the root of a serialization graph, you must manually call the XmlAdapter subclass defined in the XmlJavaTypeAdapter, both for marshalling and unmarshalling. The following example illustrates this problem:

```
... (imports) ...

@XmlRootElement
@XmlJavaTypeAdapter(value=AdaptedExample.MyAdapter.class)
public class AdaptedExample {

  public String toAdapt = "Test";



  @XmlRootElement(name="root-element")
  public static class MyJAXBFriendlyType {
    @XmlAttribute(name="id")
    public String adapted;
  }

  public static class MyAdapter
            extends XmlAdapter<MyJAXBFriendlyType, AdaptedExample> {

    @Override
    public AdaptedExample unmarshal(MyJAXBFriendlyType v) throws Exception {
      AdaptedExample a = new AdaptedExample();
      a.toAdapt = v.adapted;

      return a;
    }

    @Override
    public MyJAXBFriendlyType marshal(AdaptedExample v) throws Exception {
      MyJAXBFriendlyType t = new MyJAXBFriendlyType();
      t.adapted = v.toAdapt;

      return t;
    }

  }
}
```

In this example, you want the serialization of AdaptedExample to be done by the XmlAdapter subclass, MyAdapter. MyAdapter adapts AdaptedExample to and from MyJAXBFriendlyType. However, when serializing an instance of AdaptedExample as the root of serialization, the adapter is ignored (had it been serialized anywhere else in the object graph it would have been used, but not at the root). Serializing as the root with the code:

```
m.marshal(new AdaptedExample(), System.out);

... yields the following output ...

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adaptedExample>
    <toAdapt>Test</toAdapt>
</adaptedExample>
```

The preceding XML has not been adapted. Therefore, whenever you serialize an object whose class is marked with `XmlJavaTypeAdapter` as root, you have to manually perform the transformation:

```
MyAdapter adapter = new MyAdapter();
m.marshal(adapter.marshal(new AdaptedExample()), System.out);

... yields the following output ...

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<root-element id="Test">
</root-element>
```

You will have to know which objects you are serializing as root are marked with `XmlJavaTypeAdapter`, and manually use the adapter on both ends of serialization — during marshalling as shown previously, as well as during unmarshalling.

When to Use JAXB

JAXB is fundamentally different from either the Java Serialization API or the XMLEncoder/Decoder API. In the Java Serialization and XMLEncoder/Decoder APIs, the developer designs Java classes and does not worry about the serialization file format — that is taken care of by the APIs. However, it has the unfortunate disadvantage of limiting the use of the serialized objects to Java-based applications. With JAXB, you can either generate Java data classes from an XML schema, or annotate existing Java classes to customize and create a new XML file format. JAXB is ideal for reading and writing a third-party XML schema, or creating an XML schema for other third parties to use. Advantages to using JAXB include:

- ❑ Existing objects can be annotated to quickly allow for serialization to a custom-defined XML format
- ❑ XML file formats defined by JAXB can by read by other applications written in any language
- ❑ The XML structure of serialized documents is completely customizable via XML Schema
- ❑ Fast way to read XML data based on an XML schema — uses less memory to represent an entire XML document in memory than a DOM tree

Its disadvantages are namely:

- ❑ Requires more development effort — sometimes you need to manage *two* data models, one your application can use, and the JAXB-generated data model
- ❑ Difficult to ensure referential integrity across an object graph
- ❑ Working with JAXB-generated objects can be unwieldy because they are generated — things like naming and object creation are more tedious to develop with than custom Java classes

JAXB should be used when you want a human-readable file format that can be edited by users. It should be used when you are developing a file format you wish non-Java–based applications to be able to read. It can be used in conjunction with other XML technologies, and to read third-party XML documents based on third-party XML schemas. Because of the ease of annotating classes, JAXB becomes ideal for quick application configuration files — simply annotate the config classes you already have and read and write to disk.

Where JAXB Fits in the JDK

JAXB is one of the new additions to JDK 6. It is one of the core technologies supporting another new JDK feature: Web Services. As discussed in Chapter 11, the JDK now includes tools to automatically publish and import Web Services. These tools use JAXB to generate classes based on the Web Services Definition Language (WSDL). JAXB now fits cleanly into the Web Services stack, and is an integral part of both publishing and importing Web Services. For publishing Web Services, JAXB is used to generate XML schemas to put into WSDL based on the methods of a class that are published as Web Services. JAXB generates these schemas based on the parameters and return types of the methods being published. As you have seen in this chapter, JAXB is useful from a file and third-party schema perspective, but in the bigger picture, it is also a key technology enabling Java Web Services. See Chapter 11 for more information on how Web Services are now integrated into JDK 6.

Summary

Saving the state of an application to a file is saving all of the pieces of its in-memory data model necessary to reconstruct it exactly as it was at a later point of time. Most object-oriented applications store their data model as a set of data storage classes. In Java, it is standard practice to have the data model represented as a series of classes following the JavaBeans conventions and utilizing collection classes where necessary (such as lists, maps, trees, sets, and so on). In applications that have graphical user interfaces, it is best to separate the in-memory data structure from the GUI toolkit classes as much as possible. The standard Java GUI toolkit, Swing, follows the Model-View-Controller design pattern to accomplish this separation. This way, to persist the state of an application, *only* the data model needs to be written to disk — the GUI is simply a transient aspect of the state of the application. Normally, when you say you want to be able to save an application's state, you are referring to saving some sort of file an application produces, whether an image file, a word processing document, or a spreadsheet. These types of files are simply a data model persisted to disk. By keeping your data model separate from your GUI classes, it is easier to save it off to a file. This chapter looked at the Java Serialization API and the XMLEncoder/Decoder API. These APIs take a set of Java classes, and persist enough information to disk to reconstruct the actual object instances as they used to look in memory. This methodology makes adding serialization capabilities to an application easy, but at the cost of limiting the use of the serialized information to Java-based applications.

The JAXB API takes a fundamentally different approach, and bases all of its serialization and deserialization on XML Schema. The file formats read and saved by JAXB are all based on XML Schema and can be completely customized, unlike XMLEncoder/Decoder. JAXB data models can be generated from existing XML schema documents or existing data models can be annotated with JAXB annotations to create custom XML formats. XML schemas can then be generated from these annotated data models to allow third parties access to the blueprints of the file format, making it easier for them to interoperate with your data. JAXB is ideal for data interoperability, and XMLEncoder/Decoder and the Java Serialization API are ideal for saving exact Java object graphs (they enforce object graph referential integrity).

Both the JAXB API and the XMLEncoder/Decoder API persist their information in XML — but the XML produced by the XMLEncoder/Decoder API can only be used by Java-based applications. The Java Serialization API serializes its information in a Java-specific binary format that is much more efficient than XML but also is only useful by Java applications and is not human-readable, not to mention presents versioning problems across different versions of the classes serialized. Persisting your applications using files can require as little design and development time as you give it. If you use JAXB it takes a little more time. Your application's in-memory data model is one of the most important aspects of your data design. Once that exists, the various serialization and persistence strategies found in this chapter can all be applied. The next chapter talks about how to serialize your application's data model using a database, which is usually necessary for multi-user systems.