EXHIBIT 10

ı.lang.reflect
ds. The Field
l for any given
s an invoke()
e Constructor
ıvokes the rep-
class. It does
nd write array

) represent not
y types. There
tive types, and
special Class
rimitive types.
example, and

ject that repre-
ther type with
g Class object.

you can write
PE.

ed quite a lot of
; "beans" as fol-
be manipulated
s and interfaces

people who are
g that a builder
e., to determine
d what events it
o display to the
of naming con-
e conventions, a
erties, methods,
ection to obtain
orm of a Bean-
ts describing the

ces intended for
Chapter 10, *Java*
ner. One of the
ss that all beans
n, beans should

follow certain naming conventions. The java.beans classes that a bean creator uses are generally auxiliary classes, used not by the bean, but by the builder tool that manipulates the bean. These auxiliary classes are shipped with a bean, and provide additional information or methods that a builder tool may use with the bean. These classes are not included in finished software built with the bean.

For example, one of the auxiliary classes a bean may define is a custom BeanInfo class to provide information to the builder tool that is not available through the Reflection API. This information might include a human-readable description of the bean's properties, methods, and events, for example. Or, if a bean does not follow the standard naming conventions, this custom BeanInfo class must also provide more basic information about the bean's properties, methods, and events.

Besides a BeanInfo class, complex beans may also provide a Customizer class and one or more PropertyEditor classes. A Customizer class is a kind of configuration tool or "wizard" for a bean. It is instantiated by the builder tool in order to guide the user through bean customization. A PropertyEditor class is used to allow the user to edit the value of bean properties of a particular class. Builder tools have built-in property editors for common types such as strings, colors, and fonts, but a bean that has properties of some unusual or custom type may want to provide a Property-Editor subclass to allow the user to easily specify values for those properties.

The third level at which the JavaBeans API can be used is by programmers who are assembling an application using beans. Some programmers may do this through a builder tool, while others may do it "by hand", the old-fashioned way. Programmers using beans do not typically have to use the java.beans package. At this level, it is more a matter of reading the documentation for the particular beans being used and following those instructions. Nevertheless, a programmer using beans does need to be familiar with the event model used by beans, which is the same as the Java 1.1 event model for AWT. Also, programmers using beans "by hand" should be familiar with the naming conventions for bean properties, methods, and events, in order to more easily understand how a given bean can be used. In Java 1.1, all AWT components are beans and follow these naming conventions.

Enterprise APIs: JDBC, RMI, and Security

Java 1.1 provides a number of important new features that are loosely grouped under the name "Enterprise APIs." These include JDBC (Java DataBase Connectivity), RMI (Remote Method Invocation) and Java Security. With release 1.1, Java has grown too big for all of it to be documented, even in quick-reference format, in a single volume. Therefore, the JDBC, RMI, and Security packages may be documented, along with other, forthcoming Enterprise APIs, in a separate volume. Note, however, that while this volume does not cover the Java Security API, it *does* cover applet security, signed applets, and the *javakey* program that is used to create digital signatures, generate key pairs, and manage a database of entities and their keys.

Applet Changes

There are several new features in Java 1.1 that affect applets. The first is the introduction of JAR files. "JAR" stands for Java ARchive, and a JAR file is just that: an archive of files used by a Java applet. An applet often requires multiple class files, as well as images, sounds, and other resources, to be loaded over the the network. Prior to Java 1.1, each of these files was loaded through a separate HTTP request, which is fairly inefficient. With Java 1.1, all (or many) of the files an applet needs can be combined into a single JAR file, which an applet viewer or Web browser can download with a single HTTP request. Chapter 6, *Applets*, demonstrates the use of JAR files.

JAR files are stored in the ZIP file format. A JAR archive can be created with the *jar* tool shipped with the JDK. Once you have created a JAR file, you refer to it in a `<APPLET>` tag with the `ARCHIVE` attribute. This `ARCHIVE` attribute may actually be set to a comma-separated list of archive files to be downloaded. Note that specifying an `ARCHIVE` attribute simply tells the applet viewer or browser the name of a JAR file or files to load; it does not tell the browser the name of the applet that is to be run. Thus, you still must specify the `CODE` attribute (or the new `OBJECT` attribute, as we'll see below). For example, you might use an `<APPLET>` tag like the following to tell the browser to download the *animation.jar* file and start the applet contained in the file *Animator.class*:

```
<APPLET CODE="Animator.class" ARCHIVE="animation.jar" WIDTH=500 HEIGHT=200>
</APPLET>
```

There is another advantage to the use of JAR files. Every JAR file contains a "manifest" file, which you either specify explicitly when you create the archive, or which is created for you by the *jar* tool. The manifest is stored in a file named `META-INF/MANIFEST.MF` and contains meta-information about the files in the archive. By default, the *jar* tool creates a manifest file that contains MD5 and SHA message digests for each file in the archive. This information can be used by the applet viewer or Web browser to verify that the files in the archive have not been corrupted since the JAR file was created.

The main reason to include message digests in the manifest file, however, is so that a JAR file can have digital signatures added to it. An archive can be signed with the *javakey* tool. What a digital signature allows you to do is verify that the files in a JAR file have not been modified since the digital signature was added to the archive. If you trust the person or entity who signed the file, then you ought to trust the applet contained in the JAR file. (The *javakey* tool allows you to specify whether or not you trust any given entity.) Chapter 6 also describes how you might use digital signatures and *javakey*.

In JDK 1.1, the *appletviewer* tool understands digitally signed JAR files. When it loads an applet that has been signed by a trusted entity, it runs that applet without subjecting it to the usual security restrictions—the applet can read and write files, and do anything that a standalone Java application can do. Common Web browsers are likely to follow suit and give special privileges to trusted applets. One refinement we may see in the future is the ability to specify varying levels of trust, and to assign different sets of privileges to applets at those varying trust levels.

Besides the introduction of JAR files and trusted applets, Java 1.1 also supports "serialized applets." In an `<APPLET>` tag, you can specify the `OBJECT` attribute instead of the `CODE` attribute. If you do this, the value of the `OBJECT` attribute should be the name of a file that contains a serialized representation of the applet to be run. Graphical application-builder tools may prefer to output applets as pre-initialized object trees, rather than generating custom Java code to perform the initializations. See Chapter 9 for more information on serialized applets.

New JDK U

JDK 1.1 includes
already seen *jar*
JAR archives. In
for managing a
digital signatures

serialver is a new
is deserialized, it
matches the ver
unique identifie
When an incom
puted, and the n
is used to compu

native2ascii is a
file encoding. T
Unicode charact
convert its inpu
file that uses the
locally-encoded

In addition to t
rmic and *rmireg*
They will be do