Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-ten)

10.7: The Java Beans API

One of the software buzzwords of recent years is *componentware:* software assembled from loosely coupled, general purpose components to meet a specific need. The classic example is the Windows Custom Controls. A Custom Control is a unit of functionality whose internal details are unimportant, but whose generally useful functionality is exposed via a well-known public interface. In the present case, this well-known interface is Microsoft's Object Linking and Embedding (OLE), a *component framework* that makes it easy to assemble applications from separately written and assembled pieces. You can buy custom controls from many different vendors, and quickly assemble a solution to a specific problem by stitching them together using Visual Basic or C++ code.

Microsoft Custom Controls are a great thing, but their use is limited to one operating system and hardware platform. Since Java is all about portability and interoperability, it seems natural to define a portable component framework for Java. Such a framework exists, in fact, and it is called the Java Beans API.

The Java Beans API uses the Reflection API to allow the components (now called "Beans") to discover information about each other's methods. You put a Bean-based application together by informing each Bean about the other Beans it should be "connected to." The Bean Development Environment can generate code that directly calls the appropriate methods, or the Beans can use Method.invoke() to call the methods by name.

Virtually any Java class can be a Bean. The JavaBeans API does not define a Bean via inheritancethere is no Bean class that all Beans must inherit from. Instead, a Bean must conform to certain conventions for naming its methods and responding to events. This makes Beans very "lighweight"--you do not need to implement a lot of code to create a Java Bean.

The Beans API is primarily a communications mechanism between Bean objects and between Beans and a development environment in which they are assembled. Beans can communicate in two ways. The first is by using the AWTEvent mechanism. This makes every AWT Component automatically a Bean. The other mechanism is simply to provide accessor and mutator methods for properties. An example here is worth a thousand words:

```
public class SimpleBean
  {
   long size;
   public long getSize() { return size; }
   public void setSize(long s) { size = s; }
  }
```

This is a perfectly valid Bean with one property named size. getSize( ) is an accessor method for size; setSize( ) is a mutator method. A property does not need to have both; read-only properties can have only an accessor method, for example. Bean development environments use Reflection to find these methods and to notify other Beans about the availability of various properties.

There is more to the Beans API than we've covered here. For example, there is a mechanism via which a Bean can present a customization dialog, and another mechanism for allowing Beans to share a MenuBar with other Beans. We've just looked at the basics. Refer to the Bean Development Kit (BDK) documentation for more information.

JavaBeans can interoperate with other component technologies. For example, an adapter which lets any JavaBean be plugged into a container designed for an ActiveX Control, one variation of Microsoft's Custom Controls. This lets any Bean masquerade as an ActiveX Control when appropriate. The Bean itself is still useful on other platforms, as well, which makes them an excellent choice for building portable components.