Introduction to Java Beans

- ***Java Beans***

With the advent of Sun's Java Development Kit (JDK) 1.1, the Java community was introduced to the JavaBeans software component model. Formally, this model is defined by a published specification and a set of application programming interfaces (APIs) contained in the core Java platform. According to that specification, "A JavaBean is a reusable software component that can be manipulated visually in a builder tool." This notion should be very familiar to developers of Microsoft Windows-based software who have been using integrated development environments (IDEs) to build applications out of COM/ActiveX components. The intent of this style of application development is to reduce time to market by enabling rapid, visual integration and customization of reusable components. According to authors Cay Horstmann and Gary Cornell, the 1997 market for reusable Visual Basic and Delphi components exceeded $400,000,000. This certainly indicates a healthy interest in component-based software development.

The JavaBeans specification details a set of unifying features for Java's component model: properties, events, introspection, customization, and persistence.

Properties are named attributes representing a bean's current state. For instance, a button may have a property named font that represents the font used to display the button's label. Properties may be of any type, including primitives like int and class or interface types like java.awt.Font. Properties may be exposed in a number of ways. In scripting environments, field-style syntax may be used to get and set properties. For example, Object.Property = value, and value = Object.Property. IDEs may present a visual property sheet containing all of a bean's properties for quick and easy editing. Additionally, getter and setter

Our latest book

JavaSoft - the heart of it all

Visigenics - makers of Visibroker

Iona Technology - makers of ORBix

methods are available for programmatic access. Properties may be exposed as read/write, read-only, or write-only, depending on the requirements for a particular component. No matter what the environment, access to a bean’s properties eventually results in a method call to that bean (a getter or setter method). This allows for various programmatic side-effects, including event generation. For example, setting the font of a button may cause an event to be generated that, in turn, triggers a set of related components to change their fonts to match.

Events provide a convenient mechanism for notifications that something interesting has happened. In particular, a bean will use events to propagate state-change notifications between itself (the source) and one or more interested components (the listeners). This model is highly flexible in that it allows developers to extend the standard set of event types and propagation models to suit the requirements of their applications. Any bean that is the source of an event provides a registration mechanism to allow other components to add or remove themselves from a collection of interested listeners. Each listening component implements an associated interface that includes methods to handle generated events. In this way, IDEs can be used to visually create stimulus-response linkages among beans to exhibit a diverse range of application behavior.

Introspection allows for the dynamic discovery of a bean’s properties, events, and methods. This enables development environments to handle the integration of new beans, and their associated functionality, without modification to the environment. For the general case, "design patterns" are used to standardize the introspection process. Within the JavaBeans architecture, "design patterns" refers to a convention for naming and defining sets of methods and interfaces that are used for standard purposes. For example, getFont and setFont methods would be defined to get and set a property named font. The use of these design patterns is, however, entirely optional. Developers may associate a BeanInfo class with a bean to explicitly specify properties, events, and methods.

Customization addresses the ability to modify the appearance and/or behavior of the various beans used to build an application. As mentioned previously, one way this is enabled is through an IDE’s visual property sheet. In general, simple properties are exposed through an associated text field in the property sheet. For example, a user may enter Cancel to set a button's label. In some cases, however, getting and setting a property through a textual representation may be counter-intuitive. Consider the case of a font property. Instead of forcing the user to type in a textual specification for a font, a more effective alternative would be to use a font-chooser dialog that would display all the available fonts and allow the user to visually inspect and select an appropriate font. For just such a case, JavaBeans provides a mechanism for defining property editors. Some specialized property editors are built directly into the JDK, while others may be designed by developers as needed by their beans. While the combination of property sheets and editors will suffice for most beans, a more generalized and powerful mechanism is also provided to address the issue of customizing more complex beans. A developer-designed visual customizer may be associated with a bean to completely control the bean's customization. Normally, this customizer will be presented in a separate dialog window, and may redraw itself as the user performs the required steps to customize the bean. As is the case throughout JavaBeans, simple cases are handled simply yet support for complex behavior is always available.

Persistence allows an application's various component linkages and customizations to be "freeze dried" for later use. Once an application has been constructed, it can be saved to persistent media, a hard-drive for example, through Java's object serialization mechanism. At run-time, the application can then be reconstructed with the desired state and behavior. The familiar theme here is that simple, default cases are handled with next to no requirements on the developer. To the extent that greater control is required over the reading and writing of a bean's state, the JavaBeans architecture provides the framework to allow for very complex behavior.

The software development marketplace continues to testify to the success of component-based software development. With JavaBeans, this framework has been built into Java's core . As Java continues to grow, JavaBeans will play an increasingly important role in enabling enterprise-level application development. One need look no further than the current level of support and commitment of third-party vendors to the Enterprise JavaBeans architecture to read the writing on the wall. In the near future, Java components will provide multitier, distributed environments that improve scalability, performance, and reliability.