Richard G Baldwin (512) 223-4758, baldwin@austin.cc.tx.us,
http://www2.austin.cc.tx.us/baldwin/

JavaBeans, Overview

Java Programming, Lecture Notes # 500, Revised 02/14/98.

Preface

Students in Prof. Baldwin's **Advanced Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

Introduction

A ***Java Bean*** is the name trademarked by Sun and given to a Java class that adheres to a specific and well-defined set of *interface specifications*.

According to JavaSoft,

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

The purpose of a Bean is to be installed in the toolbox of a visual builder tool (**VBT)** so that it can be incorporated into new applications and applets with no requirement to recompile the code for the Bean.

Furthermore, two or more Beans should be able to be installed in such a toolbox and caused to communicate with one another without the requirement to recompile either of them.

Depending on the capability of the VBT, it may be possible for the programmer to combine two or more Beans into a working application or applet without the requirement to write any new code (although the VBT will usually write new code on behalf of the programmer).

For example, the *BeanBox* that we will discuss later allows Beans to be wired together strictly using mouse actions on the Beans themselves and on menus.

Once a new application is created by combining individual Beans, it should be possible to save the new application or applet for later use. This includes saving the current state of all the Beans incorporated into the application.

If you are familiar with Microsoft's *Visual Basic* or Borland's *Delphi*, then you are familiar with reusable components similar to Java Beans. The VBX files and OCX files used in these two builder tools serve a purpose similar to that served by Java Beans in Java builder tools.

The Beans Interface

The following five attributes are common to Beans.

- **Properties**
- Customization
- Persistence
- **Events**
- Introspection

When we speak of the *interface* to a class in the context of a Bean, we are usually referring to the following three attributes of the class:

- **Properties**
- Methods
- **Events**

As you can see, with the exception of *Methods*, these two lists overlap. We will briefly discuss all five of these attributes in addition to **methods** in the sections which follow. We will discuss them all in detail in subsequent lessons.

Properties

In the simplest case, a property of a Bean consists of an instance variable whose value can be manipulated using a pair of *set* and *get* methods. For example, the following pair of methods, in conjunction with the instance variable to which they refer, constitute a property.

```
public void setDelay(int delayIn){myDelay = delayIn;}
public int getDelay(){return myDelay;}
```

(Some OOD books would refer to **setDelay()** as a *mutator* and would refer to **getDelay()** as a *field accessor*, or possibly simply as an *accessor*.)

In this case, according to Beans *design patterns*, the name of the property is **delay** and the name of the instance variable used to maintain the property is **myDelay**. In this case, the methodology used to establish that this is a property is based on *design patterns* (which will be explained in more detail later).

If the *set* method exists without a corresponding *get* method, then the property is a *write-only* property. Similarly, if the *get* method exists without a corresponding *set* method, then the property is a *read-only* property. (This is probably the more common of these two special cases.)

It is also possible to forego *design patterns* and provide explicit information regarding various aspects of the interface.

Note that the above methods are declared **public**. Common jargon has it that this property has been *exposed* to the builder tool.

Another design pattern used to identify properties has to do with **boolean** properties. If the underlying instance variable type is **boolean**, an *accessor* method of the form

```
public boolean is<PropertyName>()
```

would expose the property as a **boolean** property to a VBT.

There are four kinds of properties:

- Simple
- Indexed
- Bound
- Constrained

We will discuss the four types in detail in subsequent lessons.

Methods

All **public** methods are automatically exposed to the VBT, unless the Bean's methods are explicitly identified as described above.

Exposure of the methods means that the facilities of the VBT can link events generated by other Beans to those methods. Stated differently, this means that those methods can be invoked by the code generated by the VBT as a result of some event possibly involving that Bean or another Bean.

In the general case, if necessary to provide *thread-safe* operation, methods in a Bean should be *synchronized* to prevent them from being invoked from two or more threads at the same time.

Events

Java Beans use the *Delegation Event Model* of JDK 1.1 that we learned about in earlier lessons.

By default, a Bean can generate any type of event supported by its parent class.

In addition, a Bean can expose the fact that it can generate events in a *multicast* sense by providing a pair of public methods similar to the following (this pair of methods represents another *design pattern*):

```
public synchronized void addMouseListener(MouseListener e){...}
public synchronized void removeMouseListener(MouseListener ml)
  {...}
```

We have seen methods similar to these in previous lessons discussing event handling in JDK 1.1. The body of these two methods must be capable of maintaining a list of *Listener* objects which are to be notified whenever an event of the specified type occurs. Generally notification will involve invoking a specific method on each *Listener* object in the list and passing an object containing a description of the event as a parameter. We will see examples in subsequent lessons.

In keeping with the terminology of the JDK 1.1 Delegation Event Model, the Bean that maintains the list is the *Source* Bean and the objects added to the list (registered) are the *Listener* objects.

In this case, a VBT might have the ability to cause other Bean objects to be added to the list of registered *Listener* objects so that they would be automatically notified whenever an event of the specified type occurs on the *Source* object.

Introspection

The JavaBeans APIs include the following class:

- **java.beans.Introspector**.

This class provides a standard way for VBTs to learn about the properties, events, and methods of a target Bean's class.

The **Introspector** class contains methods that can be used to analyze the target Bean's class and superclasses looking either for explicit or implicit information. The information discovered is used to build and return an object of type **BeanInfo** that describes the target Bean.

As mentioned earlier, the programmer

- can provide explicit information about a Bean,
- can rely on the recognition of *design patterns*, or
- some combination of the two.

Both approaches will be discussed in detail in a subsequent lesson.

The methods of the **Introspector** class use low-level *reflection* techniques in the analysis of the Bean. Low-level reflection techniques were studied in an earlier lesson.

The primary method of the **Introspector** class used to analyze a Bean is the **getBeanInfo()** method. Simply put, this method takes a target **Class** object as a parameter and returns a **BeanInfo** object containing information about the target class. The **BeanInfo** class contains a number of methods that can be used to extract the different elements of information from the **BeanInfo** object.

There are two versions of the **getBeanInfo()** method. The method which accepts only one parameter returns information about the target class and all its superclasses.

Another version accepts a second **Class** object as a parameter and uses that class as a ceiling for introspection up the inheritance hierarchy. For example, if this second class is the direct superclass of the primary target class, only information about the primary target class is returned.

Customization

Customization is the ability of the VBT to modify the appearance or behavior of the Bean while integrating it into a larger overall application. Generally this is accomplished by modifying properties at design time. Most VBTs will provide a *property editor* as a part of the development environment for this purpose.

Sometimes it is also desirable for the bean to provide its own *property editor* to enable a VBT to modify properties whose type is otherwise unknown to the VBT.

Also, in some cases, the Bean may simply be too complex for the default tools in the VBT to handle. In those cases, the Bean can provide a *customizer*.

Persistence

Often it is necessary for the current state of a bean to be distributed in serialized form, or to be saved and later retrieved. This goes under the name *Persistence*. In order to accomplish this, classes for Beans must implement the **Serializable** interface so that the capabilities of *Object Serialization* can be used. *Object Serialization* was discussed in an earlier lesson.

The Beanbox

As of the original date of this writing (April 1997), the APIs needed to develop Beans were available for downloading from JavaSoft in the form of the Beans Development Kit (BDK). When you download the BDK, you also receive a Java application known as the *BeanBox*. The *BeanBox* is an application written in Java that can be used to test your Java Beans.

The BeanBox is not intended to be a Visual Builder Tool. However, it does resemble VBTs in that it presents a *toolbox*, a *property inspector*, and a *main form* for assembling an application. It also provides menus which allow you to hook Beans together and also to allow you to apply introspection and report on a Bean.

In order to place a new Bean in the toolbox for the BeanBox, you need to create a Java Archive Tool (JAR) file and store that file in a specific directory. Subsequent lessons will contain discussions regarding the use of the BeanBox for testing Beans as well as procedures for creating JAR files and installing them in the toolbox.

-end-