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

JavaBeans, Introspection

Java Programming, Lecture Notes # 506, Revised 02/18/98.

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.

JDK 1.1 was released on February 18, 1997 and JDK 1.1.1 was released on March 27, 1997. This lesson was originally written on April 10, 1997 using the software and documentation in the JDK 1.1.1 download package along with the April 97 release of the BDK 1.0 download package.

Introduction

The JavaBeans APIs include the following class:

java.beans.Introspector.
This class provides a standard way for visual builder tools to learn about the properties, events, and methods of a target Bean's class.

The **Introspector** class contains two overloaded versions of a single method 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. Once the object of type BeanInfo is available, a variety of methods are available to extract specific information about the bean from that object.

The programmer can elect provide explicit information about a bean or can rely on automatic low-level reflection and the recognition of *design patterns.* The explicit approach will be briefly covered here, and discussed in more 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.

The Need for Introspection

Builder environments, and some runtime situations need to identify the properties, events, and methods that a bean supports. This process is called *introspection*.

One of the goals of the Java designers was to avoid the requirement for the use of a separate specification language for defining the behavior of a bean. Their goal was to make its behavior completely specifiable in Java.

According to JavaSoft:

> "A key goal of Java Beans is to make it very easy to write simple components and to provide default implementations for most common tasks."

Therefore, they have worked to make it possible to introspect on simple beans without requiring a lot of extra effort on the part of the component developer. At the same time, they have also worked to provide the component developer with an alternative approach that provides full and

precise control over which *properties*, *events*, and *methods* are exposed for more sophisticated components.

Remember these three: *properties*, *events*, and *methods*. The object of introspection is to gather information about these the exposed *properties*, *events*, and *methods* of a bean.

This has resulted in a composite mechanism. The default case is to use low-level *reflection* to analyze the methods supported by a bean and then to apply *design patterns* to determine from the methods the specific *events*, *properties*, and *public methods* that are supported.

However, they have also made it possible for a component developer to provide a class that implements the **BeanInfo** interface and to use that class to explicitly describe the bean. This **BeanInfo** class is then used to discover the beans behavior.

The **Introspector** class is provided to allow application builders and other tools to analyze beans in a uniform manner. The **Introspector** class understands the various *design patterns* and standard interfaces and extracts the pertinent information from the bean.

Design Patterns, General

The JavsSoft meaning of *design patterns* as used in this lesson is:

"conventional names and type signatures for sets of methods and/or interfaces that are used for standard purposes."

A common example of design patterns as used in the **Introspector** class is the use of the

```
public void set<PropertyName>(<PropertyType> a);
public <PropertyType> get<PropertyName>();
```

methods to *set* and *get* the value of the property with the specified name and the specified type.

JavaSoft has settled on the use of *design patterns* for at least two reasons.

- First, they provide a useful programming standard and documentation hint for human programmers. By using *design patterns* in their programming style, the programmer who defines the class and other programmers who read it can more quickly understand and use new classes.
- Second, the use of *design patterns* makes it possible for JavaSoft and others to write tools and libraries that recognize the *design patterns* and use them to analyze and understand components. *Design patterns* are used for Java Beans as a way to implement automatic identification of *properties*, *events*, and *exported methods*.

Again, however, the use of *design patterns* is entirely optional within Java Beans. Programmers who desire to do so can explicitly specify their *properties*, *methods*, and *events* using the

BeanInfo interface. By doing that, the programmer can use whatever names they please, provided of course that they satisfy the general requirements of the Java language.

Design Patterns for Properties

Properties may be

- simple
- indexed
- bound
- constrained

In this lesson, we will deal primarily with *simple* and *indexed* properties and defer the other two to a subsequent lesson.

Simple properties

The **Introspector** uses *design patterns* to locate properties by looking for methods having signatures of the form

```
public void set<PropertyName>(<PropertyType> a);
public <PropertyType> get<PropertyName>();
```

The existence of a matching pair of such methods is regarded as defining a *read-write* property whose name will be <propertyName>. (Note the change in case of the first letter in the property name. This will be explained more fully later.)

The two methods are used to *get* and *set* the property values as the names of the method imply.

Both methods in the pair may be located either in the same class, or one may be in a subclass and the other may be in a superclass.

If only one of the methods from the pair exists, then it is regarded either as a *read-only* or a *write-only* property.

The default assumption is that the property is neither *bound* nor *constrained*. As mentioned earlier, this will be discussed in more detail in a subsequent lesson.

Reflecting the above general description in more concrete terms might result in the following

```
public void setMyProperty(int a){//...}
public int getMyProperty(){//...}
```

pair of methods for a property named **myProperty** of type **int**.

Boolean Properties

As a special case for **boolean** properties, the **Introspector** will recognize the following form either in place of or in addition to the *get* method.

```
public boolean is<PropertyName>(){//...}
```

In either case, if the " is<PropertyName>" method is present for a boolean property then it will be used to read the property value.

An example for a boolean property named **ready** might be:

```
public boolean isReady(){//...}
public void setReady(boolean m){//...}
```

It is important to remember that the instance variable used to maintain the value of the property is not required to have the same name as the property, but it may have the same name.

Indexed Properties

An indexed property is a property having multiple values stored in an array. The following design patterns are regarded as indicating a property of this type.

```
public <PropertyElementType>
                            get<PropertyName>(int a){//...}
public void set<PropertyName>(
                    int a, < PropertyElementType> b){//...}
```

where the value passed to the integer parameter is the index of the element of interest. It is also possible to have accessor methods which read and/or write the entire array. This results in design patterns which look like the following:

```
public <PropertyType>[] get<PropertyName>(){//...}
public void set<PropertyName>(<PropertyType> a[])
```

Taking all of this into account might lead to the following four methods in the design pattern for an indexed property of type **MyType** named **myProperty**.

```
//return an element
public MyType getMyProperty(int a){//...}

//set an element
public void setMyProperty(int a, MyType b){//...}
public MyType[] getMyProperty(){//...} //return an array
public void setMyProperty(MyType a[]){//...} //set an array
```

Design Patterns for Events

Events can be exposed as *multicast* or *unicast* events. A *multicast* event notifies one or more **Listeners** of the occurrence of an event. A *unicast* event can support only one **Listener**.

Multicast Events

The *design pattern* that is used to identify the events that are *multicast* by a bean consists of a pair of methods of the form:

```
public void add<EventListenerType>(<EventListenerType> a)
public void
         remove<EventListenerType>(<EventListenerType> a)
```

where

- both methods take the same " <EventListenerType>" type argument,
- the " <EventListenerType>" type implements the **java.util.EventListener** interface,
- the first method starts with " add" ,
- the second method starts with " remove" , and
- the " <EventListenerType>" type name ends with " Listener" .

This design pattern is based on an assumption that the bean is acting as a *multicast* event source for the events specified in the " <EventListenerType>" interface.

A pair of example methods that define a multicast event source might look like the following:

```
public void addMyTypeOfListener(MyTypeOfListener t){//...}
public void
          removeMyTypeOfListener(MyTypeOfListener t){//...}
```

Unicast Events

Unicast events comprise a special case. If the add method in the above design pattern throws the **java.util.TooManyListenersException**, it is assumed that the event source is *unicast* and can only tolerate a single event listener being registered at any given time. Converting the above example to a *unicast* source gives us

```
public void addMyTypeOfListener(MyTypeOfListener t)
     throws java.util.TooManyListenersException{//...}
public void
         removeMyTypeOfListener(MyTypeOfListener t){//...}
```

Design Patterns for Methods

The default assumption is that all public methods of a bean should be exposed as external methods. This makes them accessible by other components or by scripting languages. This

includes all property accessor methods and all event listener registry methods. The exception to this assumption occurs when the programmer explicitly identifies the methods.

Explicit Specification

As an alternative to the use of *design patterns*, a bean can explicitly specify which *properties*, *events*, and *methods* it supports by providing a class that implements the **BeanInfo** interface.

As mentioned earlier, a more detailed discussion of this topic will be deferred to a subsequent lesson. For the meantime, suffice it to say that application tools should always use the **Introspector** interface which combines the information from a variety of potential sources, including explicit specifications, to construct a **BeanInfo** descriptor for a target bean.

Analyzing a Bean

The **java.beans.Introspector** will search out and identify exposed *properties*, *methods*, and *events* on a bean using both explicit specifications and by performing implicit analysis using *design patterns*. That information is encapsulated into a **BeanInfo** object that describes the bean class.

The **Introspector** examines each class in the inheritance chain of the target class. It checks at each level to determine if there is a matching **BeanInfo** class providing explicit information about the bean. If it finds such a class, it uses that explicit information.

If it does not find such a class it uses low-level *reflection* APIs to study the target class and then uses *design patterns* to analyze its behavior. It then moves on up the inheritance chain.

This multi-level analysis allows component developers to deliver complex beans with explicitly specified behavior. These beans can then be subclassed and extended by end-user customers without a requirement to provide explicit behavior information. The **Introspector** can then combine the explicit behavior information provided by the bean's developer with information gained by *reflection* and *design patterns* on the behavior introduced when the bean is subclassed.

Capitalization Rules

A set of capitalization rules are defined to be used whenever *design patterns* are used to infer a *property* or *event* name.

When the name is extracted from the middle of a normal *mixedCase* Java name, the first character in the name will normally be converted to a lower case letter. For example the property accessor method named **setMyProp()** would normally result in a property name of **myProp**.

However, since it is sometimes desirable to use all uppercase names, the case will not be changed if the first two characters of the name are both uppercase. For example, the property accessor method named **setRGB()** would result in a property name of **RGB**.

Sample Program

This program was designed to be compiled and executed under JDK 1.1.1. The purpose of this program is to illustrate **Introspection**, and in particular to illustrate the use of the static **getBeanInfo()** method of the **Introspector** class to encapsulate information about a bean in an object of type **BeanInfo**.

Once the information about the bean is encapsulated in the **BeanInfo** object, a variety of other methods are used to extract specific types of information about the bean from the object.

In order to illustrate the behavior of the different methods being used, the application was applied to the skeleton bean named **Beans01** (which we discussed in an earlier lesson) and the output from the program for each section of code was included as comments in that section.

To use the program, enter the command

```
java Introspect01 beanName
```

where beanName is the name of a beans class file without the .class extension.

As a sidelight, this program also illustrates the use of the **fileWriter** and **printWriter** classes which are new to JDK 1.1. However, since they are to be discussed in another lesson dealing with I/O upgrades in JDK 1.1, they are not discussed in this lesson.

The program was tested using JDK 1.1.1 and Win95.

Interesting Code Fragments

The first interesting code fragment is that portion of the constructor that creates an object of type **Class** that describes the class of the bean specified on the command line. Here, we create an object of type **Class** that describes the class of the bean. This object will be used later with the static **getBeanInfo()** method of the **Introspector** class. The **forName()** method of the **Class** class returns such an object, given the name of the class as a **String** parameter.

```
Class myBeanClassObject = Class.forName(myBeanClassName);
```

The next interesting code fragment uses the static **getBeanInfo()** method of the **Introspector** class to obtain information about the class of the bean and its superclasses. There are two overloaded versions of this method. One version which requires a single parameter extracts information about the entire inheritance tree.

The second version that has two parameters uses the second parameter to determine the point in the tree at and above which information is not needed. In this case, we make the second parameter be the superclass of the bean, thus causing the **getBeanInfo()** method to extract

information only on the class of the bean.

```
BeanInfo beanInfo = Introspector.getBeanInfo(
     myBeanClassObject,myBeanClassObject.getSuperclass());
```

The next interesting code fragment applies the **getBeanDescriptor()** method to the **BeanInfo** object to produce an object of type **BeanDescriptor**. Such an object provides global information about a bean, including its Java class, its displayName, etc.

Once the **BeanDescriptor** object is available, other methods are applied to the object to extract specific information from the object and write it to the output file.

When this application was applied to the skeleton bean named **Beans01** that we studied in an earlier lesson, the output shown in the comments was produced by this section of code.

```
/*
Name of bean:  Beans01
Class of bean: class Beans01
*/
BeanDescriptor beanDescriptor =
                              beanInfo.getBeanDescriptor();
printWriter.println("Name of bean:  "
                                + beanDescriptor.getName());
printWriter.println("Class of bean: "
                            + beanDescriptor.getBeanClass());
printWriter.println("");
```

The next interesting code fragment uses the **getPropertyDescriptors()** method to produce an array of **PropertyDescriptor** objects.

Each object describes one property that a Java Bean exports via a pair of accessor methods. Once that array of objects is available, other methods are used to extract specific information about each of the properties and to write that information into the output file.

When this program was applied to the skeleton bean named **Beans01**, the output for this section of code was as shown in the comments. Note that the **preferredSize** property is a *read-only* property because its *set* method is *null* (there is no *set* method for this property). Note that manual breaks have been inserted to make the material fit the page.

```
/*
==== Properties: ====
Name: color
 Type:       class java.awt.Color
 Get method:
  public synchronized java.awt.Color Beans01.getColor()
```

```
 Set method:
  public synchronized void Beans01.setColor(java.awt.Color)
Name: preferredSize
 Type:       class java.awt.Dimension
 Get method: public synchronized
              java.awt.Dimension Beans01.getPreferredSize()
 Set method: null
Name: myBooleanProperty
 Type:       boolean
 Get method:
  public synchronized boolean Beans01.isMyBooleanProperty()
 Set method:
  public synchronized void
                     Beans01.setMyBooleanProperty(boolean)
*/
printWriter.println("==== Properties: ====");
PropertyDescriptor[] propertyDescriptor =
                          beanInfo.getPropertyDescriptors();
for (int i=0; i<propertyDescriptor.length; i++) {
  printWriter.println("Name: " +
                           propertyDescriptor[i].getName());
  printWriter.println(" Type:       "
                + propertyDescriptor[i].getPropertyType());
  printWriter.println(" Get method: "
                  + propertyDescriptor[i].getReadMethod());
  printWriter.println(" Set method: "
                 + propertyDescriptor[i].getWriteMethod());
}//end for-loop
printWriter.println("");
```

The next interesting code fragment uses the **getEventSetDescriptors()** method to produce an array of **EventSetDescriptor** objects. Other methods are then used to extract information from each of those objects.

When this program was applied to the skeleton bean named **Beans01**, the output for this section of code was as shown in the comments. (Note that the line breaks were inserted during the editing of this document to make it all fit on the screen.) In this case, only one *multicast* event was exposed by the bean.

```
    ==== Events: ====
    Event Name: action
     Add Method:    public synchronized void Beans01.
          addActionListener(java.awt.event.ActionListener)
     Remove Method: public synchronized void Beans01.
          removeActionListener(
                              java.awt.event.ActionListener)
     Event Type: actionPerformed
    */
    printWriter.println("==== Events: ====");
    EventSetDescriptor[] eventSetDescriptor =
                          beanInfo.getEventSetDescriptors();
    for (int i=0; i<eventSetDescriptor.length; i++) {
```

```
      printWriter.println("Event Name: "
                          + eventSetDescriptor[i].getName());
      printWriter.println(" Add Method:    " +
             eventSetDescriptor[i].getAddListenerMethod());
      printWriter.println(" Remove Method: " +
          eventSetDescriptor[i].getRemoveListenerMethod());
      MethodDescriptor[] methodDescriptor =
                         eventSetDescriptor[i].
                              getListenerMethodDescriptors();
      for (int j=0; j<methodDescriptor.length; j++) {
          printWriter.println(" Event Type: "
                            + methodDescriptor[j].getName());
      }//end for-loop
    }//end for-loop
    printWriter.println("");
```

The final interesting code fragment uses the **getMethodDescriptors()** method to produce an array of **MethodDescriptor** objects. Each such object describes a particular method that a Java Bean supports for external access from other methods.

This program was applied to the skeleton bean named **Beans01**, and the output for this section of code is shown in the comments. You should note that the list of methods includes property accessor methods, methods that expose multicast event support, and "ordinary" methods that are not intended to expose properties or events.

```
    /*
    ==== Methods: ====
    makeRed
    setMyBooleanProperty
    removeActionListener
    addActionListener
    setColor
    getColor
    getPreferredSize
    makeBlue
    isMyBooleanProperty
    */
    printWriter.println("==== Methods: ====");
    MethodDescriptor[] methodDescriptor =
                            beanInfo.getMethodDescriptors();
    for (int i=0; i<methodDescriptor.length; i++) {
        printWriter.println(methodDescriptor[i].getName());
    }//end for-loop
    printWriter.println("");.
```

The above code fragments represent only those parts of the program that are new and interesting in the context of Java Beans. A listing of the entire program with comments is contained in the next section.

Program Listing

A listing of the entire program with comments follows. See the previous sections for an operational description of the program.

```
/*File Introspect01.java Copyright 1997, R.G.Baldwin
This program was designed to be compiled and executed
under JDK 1.1.1.

The purpose of this program is to illustrate the use of the
static getBeanInfo() method of the Introspector class to
encapsulate information about a bean in an object of type
BeanInfo.

Once the information about the bean is encapsulated in the
BeanInfo object, a variety of other methods are used to
extract specific types of information about the bean from
the object.

In order to illustrate the behavior of the different
methods being used, the application was applied to the
skeleton bean named Beans01 (discussed in an earlier
lesson) and the output from the program for each section of
code was included as comments in that section.

The program was tested using JDK 1.1.1 and Win95.
**********************************************************/
import java.io.*;
import java.beans.*;
import java.lang.reflect.*;

public class Introspect01
{
  //name of bean class file to be analyzed
  static String myBeanClassName;
  FileWriter fileWriter;
  PrintWriter printWriter;

  //Start the program and get the name of the class file
  // for the bean in the String myBeanClassName
  public static void main(String args[]) throws Exception {
    myBeanClassName = args[0];
    Introspect01 x = new Introspect01();
  }//end main

  public Introspect01() throws Exception {//constructor
    //Open an output file to store the report in.
    fileWriter = new FileWriter("junk.txt");
    printWriter = new PrintWriter(fileWriter);

    //Create an object of type Class that describes the
    // class of the bean. The static method
    // Introspector.getBeanInfo() requires either one or
    // two objects of type Class as parameters.  The
    // forName() method of the Class class returns such an
    // object, given the name of a class as a parameter.
    Class myBeanClassObject = Class.forName(
```

```
                                              myBeanClassName);

    //Given the Class object that describes the bean's
    // class, use the static getBeanInfo() method of the
    // Introspector class to obtain information about the
    // class of the bean.  Save this information in an
    // object of type BeanInfo.  The second parameter
    // passed to getBeanInfo() prevents introspection from
    // going further up the inheritance hierarchy.
    BeanInfo beanInfo = Introspector.getBeanInfo(
      myBeanClassObject,myBeanClassObject.getSuperclass());

    //A BeanDescriptor object provides global information
    // about a bean, including its Java class, its
    // displayName, etc. Use the getBeanDescriptor()
    // method to extract information of that type from
    // the beanInfo object and store it in a new
    // BeanDescriptor object. Store the information in the
    // output file using methods designed to extract the
    // name and class of the bean from the beanDescriptor
    // object.  When this application was applied to the
    // skeleton bean named Beans01, the following output
    // was produced by this section of code.
    /*
    Name of bean:  Beans01
    Class of bean: class Beans01
    */
    BeanDescriptor beanDescriptor =
                             beanInfo.getBeanDescriptor();
    printWriter.println("Name of bean:  " +
                               beanDescriptor.getName());
    printWriter.println("Class of bean: " +
                           beanDescriptor.getBeanClass());
    printWriter.println("");

    //A PropertyDescriptor object describes one property
    // that a Java Bean exports via a pair of accessor
    // methods. Use the getPropertyDescriptors() method
    // to create an array of PropertyDescriptor objects,
    // one for each exported property.  Then store that
    // information in the output file using methods
    // designed to extract the name of the property, the
    // type of the property, the name of the get method,
    // and the name of the set method. When this
    // application was applied to Beans01, the following
    // output was produced by this section of code.  Manual
    // line breaks were inserted to make it fit in the
    // available space.
    /*
    ==== Properties: ====
    Name: color
     Type:       class java.awt.Color
     Get method: public synchronized java.awt.Color
                                         Beans01.getColor()
     Set method: public synchronized void
                           Beans01.setColor(java.awt.Color)
```

```
    Name: preferredSize
     Type:       class java.awt.Dimension
     Get method: public synchronized java.awt.Dimension
                                 Beans01.getPreferredSize()
     Set method: null
    Name: myBooleanProperty
     Type:       boolean
     Get method: public synchronized boolean
                             Beans01.isMyBooleanProperty()
     Set method: public synchronized void
                     Beans01.setMyBooleanProperty(boolean)
    */
    printWriter.println("==== Properties: ====");
    PropertyDescriptor[] propertyDescriptor =
                         beanInfo.getPropertyDescriptors();
    for (int i=0; i<propertyDescriptor.length; i++) {
      printWriter.println("Name: " +
                          propertyDescriptor[i].getName());
      printWriter.println(" Type:       " +
                  propertyDescriptor[i].getPropertyType());
      printWriter.println(" Get method: " +
                    propertyDescriptor[i].getReadMethod());
      printWriter.println(" Set method: " +
                   propertyDescriptor[i].getWriteMethod());
    }//end for-loop
    printWriter.println("");


    //An EventSetDescriptor object describes a group of
    // events that a given Java bean fires. Information can
    // be extracted from each object of the type.
    // When this application was applied to the Beans01
    // bean, the following output was produced by this
    // section of code (note that line breaks were
    // inserted during editing).
    /*
    ==== Events: ====
    Event Name: action
     Add Method:    public synchronized void Beans01.
           addActionListener(java.awt.event.ActionListener)
     Remove Method: public synchronized void Beans01.
        removeActionListener(java.awt.event.ActionListener)
     Event Type: actionPerformed
    */
    printWriter.println("==== Events: ====");
    EventSetDescriptor[] eventSetDescriptor =
                         beanInfo.getEventSetDescriptors();
    for (int i=0; i<eventSetDescriptor.length; i++) {
      printWriter.println("Event Name: " +
                          eventSetDescriptor[i].getName());
      printWriter.println(" Add Method:    " +
             eventSetDescriptor[i].getAddListenerMethod());
      printWriter.println(" Remove Method: " +
          eventSetDescriptor[i].getRemoveListenerMethod());
      MethodDescriptor[] methodDescriptor =
                          eventSetDescriptor[i].
```

```
                            getListenerMethodDescriptors();
      for (int j=0; j<methodDescriptor.length; j++) {
          printWriter.println(" Event Type: " +
                            methodDescriptor[j].getName());
      }//end for-loop
    }//end for-loop
    printWriter.println("");

    //A MethodDescriptor describes a particular method that
    // a Java Bean supports for external access from other
    // components.  The getMethodDescriptors() method
    // returns an array of MethodDescriptor objects where
    // each object describes one of the methods.  When this
    // application was applied to the Beans01 bean, the
    // following output was produced by this section
    // of code.
    /*
    ==== Methods: ====
    makeRed
    setMyBooleanProperty
    removeActionListener
    addActionListener
    setColor
    getColor
    getPreferredSize
    makeBlue
    isMyBooleanProperty
    */
    printWriter.println("==== Methods: ====");
    MethodDescriptor[] methodDescriptor =
                           beanInfo.getMethodDescriptors();
    for (int i=0; i<methodDescriptor.length; i++) {
        printWriter.println(methodDescriptor[i].getName());
    }//end for-loop
    printWriter.println("");

    printWriter.close();
  }//end constructor
}//end class Introspect01
```

-end-