This section was originally published in the first edition of Learning Java in the JavaBeans chapter. The dynamic event adapter developed in this section is redundant with the new Java 1.4 `java.beans.EventHandler` API. However, the techniques involved are generally useful in JavaBeans and other application areas, so we have moved this information to the CD.

Putting Reflection to Work

In this section, we'll build a dynamic event adapter that can be configured at runtime.

In Chapter 15, "Swing," we saw how adapter classes could be built to connect event firings to arbitrary methods in our code, allowing us to cleanly separate GUI and logic in our applications. In this chapter, we have described how the BeanBox interposes adapters between Beans to do this for us. We have also described how the BeanBox uses adapters to bind and constrain properties between Beans.

One of the primary motivations behind the AWT/Swing event model was to reduce the need to subclass components to perform simple hookups. But if we start relying heavily on special adapter classes, we can quickly end up with as many adapters as objects. Anonymous inner classes let us hide the existence of these classes, but they're still there. A potential solution for large or specialized applications is to create *generic* event adapters that can serve a number of event sources and targets simultaneously.

The following example, `DynamicActionAdapter`, is a generic adapter for `ActionEvent`s. A single instance of `DynamicActionAdapter` can be used to hook up a number of `ActionEvent` sources. `DynamicActionAdapter` uses reflection on both the source and target objects. This enables us to direct each event on the source object to an arbitrary method of the target object.

Here's the code:

```
//file: DynamicActionAdapter.java
import java.awt.*;
import java.util.Hashtable;
import java.lang.reflect.Method;
import java.awt.event.*;

class DynamicActionAdapter implements ActionListener {
  Hashtable actions = new Hashtable( );

  public void hookup( Object sourceObject, Object targetObject,
                      String targetMethod ) {
    actions.put(sourceObject,
      new Target(targetObject, targetMethod));
    invokeReflectedMethod( sourceObject, "addActionListener",
      new Object[] {this}, new Class[] {ActionListener.class});
  }

  public void actionPerformed(ActionEvent e) {
    Target target = (Target)actions.get( e.getSource( ) );
    if ( target == null )
      throw new RuntimeException("unknown source");
    invokeReflectedMethod(target.object, target.methodName,
                          null, null);
  }

  private void invokeReflectedMethod(
    Object target, String methodName,
    Object [] args, Class [] argTypes ) {
```

```
    try {
      Method method =
        target.getClass( ).getMethod( methodName, argTypes );
      method.invoke( target, args );
    }
    catch ( Exception e ) {
      throw new RuntimeException("invocation problem: "+e);
    }
  }

  class Target {
    Object object;
    String methodName;

    Target( Object object, String methodName ) {
      this.object = object;
      this.methodName = methodName;
    }
  }
}
```

Once we have an instance of `DynamicActionAdapter`, we can use its `hookup( )` method to connect an `ActionEvent` source to some method of a target object. The target object doesn't have to be an `ActionListener`—or any other particular kind of object. The following application, `DynamicHookupTest`, uses an instance of our adapter to connect a button to its own `launchTheMissiles( )` method:

```
//file: DynamicHookupTest.java
import javax.swing.*;
import java.awt.event.*;

public class DynamicHookupTest extends JFrame {
  DynamicActionAdapter actionAdapter = new DynamicActionAdapter( );
  JLabel label = new JLabel( "Ready...", JLabel.CENTER );
  int count;

  public DynamicHookupTest( ) {
    JButton launchButton = new JButton("Launch!");
    getContentPane( ).add( launchButton, "South" );
    getContentPane( ).add( label, "Center" );
    actionAdapter.hookup(launchButton, this, "launchTheMissiles");
  }

  public void launchTheMissiles( ) {
    label.setText("Launched: "+ count++ );
  }

  public static void main(String[] args) {
    JFrame f = new DynamicHookupTest( );
    f.addWindowListener(new WindowAdapter( ) {
      public void windowClosing(WindowEvent we) { System.exit(0); }
    });
    f.setSize(150, 150);
    f.setVisible( true );
  }
}
```

Here we simply call the dynamic adapter's `hookup( )` method, passing it the `ActionEvent` source, the target object, and a string with the name of the method to invoke when the event arrives.

As for the code, it's pretty straightforward. `DynamicActionAdapter` implements the `ActionListener` interface. When `hookup( )` is called, it registers itself with the event source, using reflection on the source object to invoke its `addAction-Listener( )` method. It stores information

about the target object and method in a `Target` object, using an inner class. This object is stored in a `Hashtable` called `actions`.

When an action event arrives, the dynamic adapter looks up the target for the event source in the `Hashtable`. It then uses reflection on the target object, to look up and invoke the requested method. Our adapter can invoke only a method that takes no arguments. If the method doesn't exist, the adapter throws a `RuntimeException`.
The heart of the adapter is the `invokeReflectedMethod( )` method. This is a `private` method that uses reflection to look up and invoke an arbitrary method in an arbitrary class. First, it calls the target's `getClass( )` method to get the target's `Class` object. It uses this object to call `getMethod( )`, which returns a `Method` object. Once we have a `Method`, we can call `invoke( )` to invoke the method.

The dynamic adapter is important because it has almost no built-in knowledge. It doesn't know what kind of object will be the event source. Likewise, it doesn't know what object will receive the event, or what method it should call to deliver the event. All this information is provided at runtime, in the call to `hookup( )`. We use reflection to look up and invoke the event source's `addActionListener( )` method, and to look up and invoke the target's event handler. All this is done on the fly. Therefore, you can use this adapter in almost any situation requiring an `ActionEvent`.

Safety Implications

If the target's event-handling method isn't found, the adapter throws a `RuntimeException`. Therein lies the problem with this technique. By using reflection to locate and invoke methods, we abandon Java's strong typing and head off in the direction of scripting languages. We add power at the expense of safety.

Runtime Event Hookups with Reflection

Our dynamic event adapter is limited to handling `ActionEvent`s. What if we want to build something like the BeanBox that can hook up arbitrary event sources to arbitrary destinations? Can we build an adapter that can listen to any kind of event?

The answer is "yes." We can use the powerful new interface proxy reflection feature introduced in SDK 1.3. The `java.lang.reflect.Proxy` class is a factory that can generate adapters implementing any type of interface at runtime. By specifying one or more event listener interface (e.g., `ActionListener`) we get an adapter that implements those listener interfaces generated for us on the fly. The adapter is a specially created class that delegates all of the method calls on its interfaces to a designated `InvocationHandler` object. See Chapter 7, "Working with Objects and Classes" for more information about the reflection interface proxy.
We should repeat that this is a feature that was introduced in the SDK 1.3. In versions of Java prior to 1.3, there is no standard way to create an object that implements a specified interface at runtime. That is, there is no way to make a specified "kind" of event listener without creating the class yourself. The BeanBox and other utilities which generated adapters dynamically prior to the SDK 1.3 did so by creating Java byte code on the fly and loading it through a class loader.