informIT

Covert Java: Hacking Non-Public Methods and Variables of a Class

Date: Jun 4, 2004 By Alex Kalinovsky. Sample Chapter is provided courtesy of Sams.

Learn to hack non-public methods and variables of a class. Alex Kalinovsky covers the problem of encapsulation, accessing packages and protected class members, and accessing private class members.

"Anything can be made to work if you fiddle with it. If you fiddle with something long enough, you'll break it."

Murphy's Technology Laws

In this chapter

The Problem of Encapsulation

Encapsulation is one of the pillars of object-oriented programming. The purpose of encapsulation is separation of the interface from implementation and modularity of application components. It is generally recommended that you make data members private or protected and provide public accessor and mutator functions (also known as *getter* and *setter* functions). It is also sometimes recommended that you make internal implementation methods private or public to protect a class from being used incorrectly. Following the principle of encapsulation helps create a better application, but occasionally it can prove to be an obstacle for usage that was not foreseen by the class developer.

We will use `java.awt.BorderLayout` in our experiments. Maybe at some point this will encourage JavaSoft engineers to add public methods. We will obtain the source code for `BorderLayout` from `src.jar` in the JDK installation directory.

Accessing Packages and Protected Class Members

We will start by demonstrating how to easily access package-visible variables and methods. Our example uses a package-visible variable, but the technique works equally well for protected visibility. A variable or method is *package visible* when no specific visibility keyword such as `public`, `protected`, or `private` is used for declaration. `BorderLayout` stores the component that was added using the `BorderLayout.CENTER` constraint in a `center` variable declared as follows:

```
package java.awt;

public class BorderLayout implements LayoutManager2, java.io.Serializable {
  ...
  Component center;
  ....
}
```

Recall that package-visible members are accessible to the class that declared them and all classes within the same package. In our example, any class in the `java.awt` package can access the `center` variable directly. A simple solution therefore is to create a helper class, `AwtHelper`, in the `java.awt` package and use it to access package-visible members of `BorderLayout` instances. `AwtHelper` has a public function that takes in an instance of `BorderLayout` and returns the component for a given layout constraint:

```
package java.awt;

public class AwtHelper {

  public static Component getChild(BorderLayout layout, String key) {
```

```
    Component result = null;

    if (key == BorderLayout.NORTH)
      result = layout.north;
    else if (key == BorderLayout.SOUTH)
      result = layout.south;
    else if (key == BorderLayout.WEST)
      result = layout.west;
    else if (key == BorderLayout.EAST)
      result = layout.east;
    else if (key == BorderLayout.CENTER)
      result = layout.center;
    return result;
  }
}
```

Let's write a test class called `covertjava.visibility.PackageAccessTest` that uses `AwtHelper` to obtain the split pane instance from Chat's `MainFrame`. The following source code excerpt is what we are mostly interested in:

```
Container container = createTestContainer();
if (container.getLayout() instanceof BorderLayout) {
  BorderLayout layout = (BorderLayout)container.getLayout();
  Component center = AwtHelper.getChild(layout, BorderLayout.CENTER);
  System.out.println("Center component = " + center);
}
```

We obtain the layout for the container and, if it is `BorderLayout`, we use `AwtHelper` to get the center component. Chat's `MainFrame` has the split pane in the center; therefore, if the code is written correctly, we should see an instance of `JSplitPane` on the system console. Running `PackageAccessTest`, we get the following exception:

```
java.lang.SecurityException: Prohibited package name: java.awt
```

The exception is thrown because `java.awt` is considered to be a system name space that should not be used by regular classes. This would not have happened if we were trying to hack a package-visible member of a third-party class, but we have intentionally picked a system class to illustrate a real-life example. The only potential problem with using this technique for a nonsystem name space such as `com.mycompany.mypackage` occurs if the package is sealed. Adding a helper class to a sealed package requires the same technique as is explained for adding a patched class in Chapter 5, "Replacing and Patching Application Classes."

Adding system classes is a little trickier because they are loaded and treated differently from application classes. Chapter 16, "Intercepting Control Flow," provides a comprehensive discussion of system classes. For now, though, it would suffice to say that to add a class to the system package, the class has to be placed on the boot class path. A directory or JAR file can be prepended or appended to the boot class path using the `-Xbootclasspath` parameter to the `java` command line. Because we already have a `patches` subdirectory for the Chat application, we will use it for system classes as well. We modify `build.xml` to move the `java.lang` directory with `AwtHelper` to `distrib/patches` and create a new script (`package_access_test.bat`) in `distrib/bin`, as follows:

```
@echo off
set CLASSPATH=..\lib\chat.jar
java -Xbootclasspath/p:..\patches covertjava.visibility.PackageAccessTest
```

Running `package_access_test.bat` produces the following output:

```
C:\Projects\CovertJava\distrib\bin>package_access_test.bat
Testing package-visible access
Center component = javax.swing.JSplitPane[,0,0,0x0,...]
```

Having to place classes on the system boot class path makes deployment a little more involved because it requires modification of the startup script. For example, a Web application that is deployed into a Web container, such as Tomcat or WebLogic, can no longer be simply deployed through a console or the application deployment directory. The script that starts the application server must be modified to include the `-Xbootclasspath` parameter. Another disadvantage of this technique is that it does not work for private members. Last, but not least, adding classes to packages can violate the license agreement. This is the case with `BorderLayout` because a section in Sun's Java license agreement explicitly prohibits adding classes to packages that begin with `java`. The next section presents another alternative that solves some of these problems.

Stories From the Trenches - WebCream is a product that converts Java AWT and Swing applications into interactive HTML Web sites. It does it by

emulating a graphical environment for the graphical user interface (GUI) application running on the server side and capturing and converting the currently displayed top window to an HTML page. To generate the HTML, WebCream iterates all containers and tries to mimic Java layouts with HTML tables. One of the layouts WebCream needs to support is `BorderLayout`. For a container with `BorderLayout`, the HTML rendering module needs to know which child component has been added to the South section, which one to the North, and so on. `BorderLayout` stores this information in the member variables `south`, `north`, and so on, and it even has a `getChild()` method that can be used to obtain the component. The problem is that the variables are declared with package visibility and the `getChild` method is declared as private. To get around the absence of public access to `BorderLayout`'s child components, WebCream engineers had to rely on the hacking techniques described in this chapter.

Accessing Private Class Members

Private members are accessible only to the class that declares them. That is one of the ground rules of the Java language that ensures encapsulation. But is it really so? Is it really enforced all the time? If you said, "Well, this guy is writing about it, so there has to be a loophole of some sort," you'd be right. The Java compiler enforces the privacy of private members at compile time. Thus, there can be no static references by other classes to private methods and variables of a class. However, Java has a powerful mechanism of reflection that enables querying instance and class metadata and accessing fields and methods at runtime. Because reflection is dynamic, compile time checks are not applicable. Instead, Java runtime relies on a security manager—if one exists—to verify that the calling code has enough privileges for a particular type of access. The security manager provides enough protection because all the functions of the reflection API delegate to it before executing their logic. What undermines this protection is the fact that the security manager is often not set. By default, the security manager is not set, and unless the code explicitly installs a default or a custom security manager, the runtime access control checks are not in effect. Even if a security manager is set, it is typically configured through a policy file, which can be extended to allow access to the reflection API.

If you looked at the `BorderLayout` class carefully, you might have noticed that it already has a method that returns a child component based on the position key. Not surprisingly, it is called `getChild` and has the following signature:

```
private Component getChild(String key, boolean ltr)
```

This sounds like good news because you don't really have to write your own implementation. The problem is that the method is declared as private and there is no public method you can use to call it. To leverage the existing JDK code, you must call `BorderLayout.getChild()` using the reflection API. We will use the same test structure as in the previous section. This time, though, instead of using `AwtHelper`, the test class delegates to its own helper function (`getChild()`):

```
public class PrivateAccessTest {

  public static void main(String[] args) throws Exception {
    Container container = createTestContainer();
    if (container.getLayout() instanceof BorderLayout) {
      BorderLayout layout = (BorderLayout)container.getLayout();
      Component center = getChild(layout, BorderLayout.CENTER);
      System.out.println("Center component = " + center);
    }
    ...
  }

  public static Component getChild(BorderLayout layout, String key) throws Exception {
    Class[] paramTypes = new Class[]{String.class, boolean.class};
    Method method = layout.getClass().getDeclaredMethod("getChild", paramTypes);
    // Private methods are not accessible by default
    method.setAccessible(true);
    Object[] params = new Object[] {key, new Boolean(true)};
    Object result = method.invoke(layout, params);
    return (Component)result;
  }

  ...
}
```

The `getChild()` implementation is the core of the technique. It obtains the method object through reflection and then calls `setAccessible(true)`. A value of `true` is set to suppress

the access control checking and allow method invocation. The rest of the method is plain reflection API usage. Running `covertjava.visibility.PrivateAccessTest` produces the same output you saw in the previous section:

```
C:\Projects\CovertJava\distrib\bin>private_access_test.bat
Testing private access
Center component = javax.swing.JSplitPane[,0,0,0x0,...]
```

This was alarmingly easy. We might have to do a little more work if a security manager is set using `System.setSecurityManager` or via a command line, which is the case for most application servers and middleware products. If we run our test passing `-Djava.security.manager` to the `java` command line, we get the following exception:

```
java.security.AccessControlException: access denied
(java.lang.RuntimePermission accessDeclaredMembers)
```

For our code to work with a security manager installed, we have to grant the permissions to access declared members through reflection and to suppress access checks. We do so by adding a Java policy file that grants these two permissions to our code base:

```
grant {
  permission java.lang.RuntimePermission "accessDeclaredMembers";
  permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
};
```

Finally, we create a new test script (`private_access_test.bat`) in the `distrib\bin` directory that adds a command-line parameter (`java.security.policy`) to install our policy file:

```
set CLASSPATH=..\lib\chat.jar
set JAVA_ARGS=%JAVA_ARGS% -Djava.security.manager
set JAVA_ARGS=%JAVA_ARGS% -Djava.security.policy=../conf/java.policy

java %JAVA_ARGS% covertjava.visibility.PrivateAccessTest
```

If a policy file is already installed, our grant clause needs to be inserted into it. Java security files allow inclusion of additional policy files using the `policy.url.n` attribute. See Chapter 7, "Manipulating Java Security," for a detailed discussion of Java security and policy files.

The technique that relies on the reflection API can be used to access package and protected members as well. This makes inserting helper classes into third-party packages unnecessary. The drawback of the reflection API is that it is notoriously slow because it has to deal with runtime information and might have to go through a number of security checks. When speed is an issue, it is preferable to rely on the helper classes for package and protected members. Yet another alternative is serializing an instance into a byte array stream and then parsing the stream to obtain the values of the member variables. Obviously, this is a tedious process that does not work for transient fields.

Quick Quiz

1. Which technique can be used to obtain a value of a protected variable?
2. Which technique can be used to obtain a value of a private variable?
3. What are the advantages and disadvantages of each technique?

In Brief

- Methods and variables that are not declared `public` can still be accessed.
- A member with `package` or `protected` visibility can be accessed by inserting a helper class into its package or using the reflection API.
- A member with `private` visibility can be accessed using the reflection API.
- If a security manager is installed, the Java policy needs to be altered to allow unrestricted access for the reflection API.
