This section was originally published in the first edition of the book, Learning Java as part of the utilities chapter. Although the SecurityManager API is still an important piece of the Java security infrastructure, its programmatic use by developers has largely been replaced by the declarative security offered through Java Security Policy files. We have therefore moved this material to the CD.

The Security Manager

A Java application's access to system resources, such as the display, the filesystem, threads, external processes, and the network, can be controlled at a single point with a security manager. The class that implements this functionality in the Java API is the `java.lang.SecurityManager` class.

As you saw in Chapter 3, "Tools of the Trade," the Java 2 platform provides a default security manager that you can use with the Java interpreter. For many applications, this default security manager is sufficient; for some types of applications, such as those that do custom class loading, you may need to write your own security manager.

An instance of the `SecurityManager` class can be installed once, and only once, in the life of the Java runtime environment. Thereafter, every access to a fundamental system resource is filtered through specific methods of the `SecurityManager` object by the core Java packages. By installing a specialized `SecurityManager`, we can implement arbitrarily complex (or simple) security policies for allowing access to individual resources.

When the Java runtime system starts executing, it's in a wide-open state until a `SecurityManager` is installed. The "null" security manager grants all requests, so the Java runtime system can perform any activity with the same level of access as other programs running under the user's authority. If the application that is running needs to ensure a secure environment, it can install a `SecurityManager` with the static `System.setSecurityManager( )` method. For example, a Java-enabled web browser such as Netscape Navigator installs a `SecurityManager` before it runs any Java applets.

`java.lang.SecurityManager` must be subclassed to be used. This class does not actually contain any abstract methods; it's `abstract` as an indication that its default implementation is not very useful. By default, each security method in `SecurityManager` is implemented to provide the strictest level of security. In other words, the default `SecurityManager` simply rejects all requests.

The following example, MyApp, installs a trivial subclass of SecurityManager as one of its first activities:

```
class FascistSecurityManager extends SecurityManager { }

public class MyApp {
    public static void main( String [] args ) {
        System.setSecurityManager( new FascistSecurityManager( ) );
        // no access to files, network, windows, etc.
        ...
    }
}
```

In this scenario, MyApp does little aside from reading from System.in and writing to System.out. Any attempt to read or write files, access the network, or even open a window results in a SecurityException being thrown.

After this draconian SecurityManager is installed, it's impossible to change the SecurityManager in any way. The security of this feature is not dependent on the SecurityManager itself; it's built into the Java runtime system. You can't replace or modify the SecurityManager under any circumstances. The upshot of this is that you have to install one that handles all your needs up front.

To do something more useful, we can override the methods that are consulted for access to various kinds of resources. Table 9b-8 lists some of the more important access methods. You should not normally have to call these methods yourself, although you could. They are called by the core Java classes before granting particular types of access.

SecurityManager Methods

Method	Can I . . .
CheckAccess(Thread g)	Access this thread?
CheckExit(int status)	Execute a System.exit( )?
CheckExec(String cmd)	exec( ) this process?
CheckRead(String file)	Read a file?

Method	Can I . . .
CheckWrite(String file)	Write a file?
CheckDelete(String file)	Delete a file?
CheckConnect(String host, int port)	Connect a socket to a host?
CheckListen(int port)	Create a server socket?
CheckAccept(String host, int port)	Accept this connection?
CheckPropertyAccess(String key)	Access this system property?
CheckTopLevelWindow(Object window)	Create this new top-level window?

Most of these methods simply return to grant access. If access is not granted, they throw a SecurityException. checkTopLevelWindow( ) is different; it returns a boolean value: true indicates the access is granted; false indicates the access is granted, but with the restriction that the new window should provide a warning border that serves to identify it as an untrusted window.

Let's implement a silly SecurityManager that allows only files whose names begin with *foo* to be read:

```
class  FooFileSecurityManager extends SecurityManager {
```

```
    public void checkRead( String s ) {
        if ( s.startsWith("foo") ) {
            return true;
        } else {
            throw new SecurityException("Access to non-foo file: "
                + s + " not allowed." );
        }
    }
}
```

Once the `FooFileSecurityManager` is installed, any attempt to read a badly named file from any class will fail and cause a `SecurityException` to be thrown. All other security methods are inherited from `SecurityManager`, so they are left at their default restrictiveness.

As we've shown, security managers can make their decisions about what to allow and disallow based on any kind of criterion. One very powerful facility that the `SecurityManager` class provides is the `classDepth( )` method. `classDepth( )` takes as an argument the name of a Java class; it returns an integer indicating the depth of that class if it is present on the Java stack. The depth indicates the number of nested method invocations that occurred between the call to `classDepth( )` and the last method invocation from the given class. This can be used to determine what class required the security check.

For example, if a class shows a depth of 1, the security check must have been caused by a method in that class—there are no method calls intervening between the last call to that class and the call requiring the check. You could allow or refuse an operation based on the knowledge that it came from a specific class.

All restrictions placed on applets by an applet-viewer application are enforced through a `SecurityManager`, including whether to allow untrusted code loaded from over the network to be executed. The `AppletSecurityManager` is responsible for applying the various rules for untrusted applets and allowing user configured access to trusted (signed) applets.