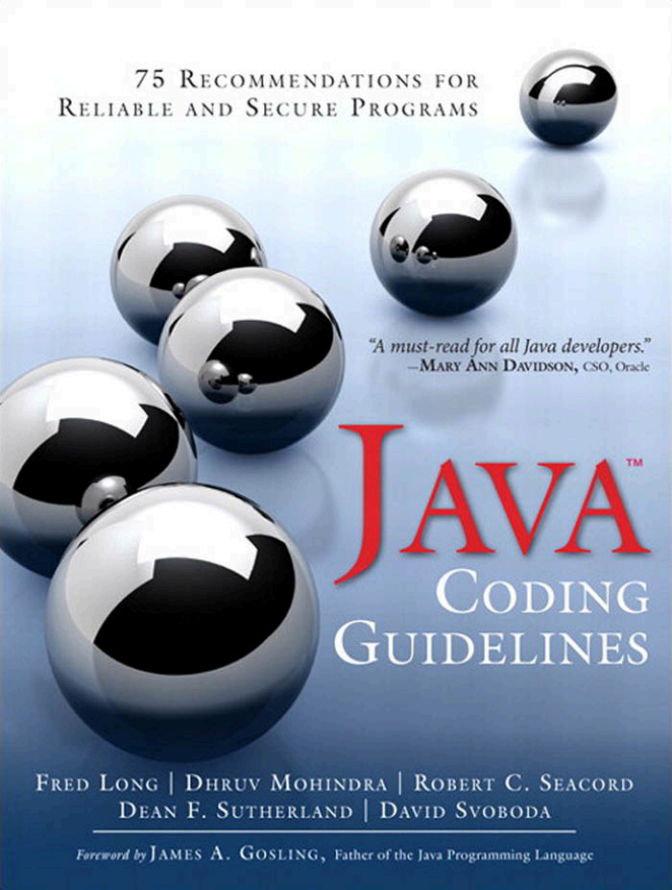


75 RECOMMENDATIONS FOR  
RELIABLE AND SECURE PROGRAMS



*"A must-read for all Java developers."*  
—MARY ANN DAVIDSON, CSO, Oracle

# JAVA<sup>TM</sup>

## CODING GUIDELINES

FRED LONG | DHURV MOHINDRA | ROBERT C. SEACORD  
DEAN F. SUTHERLAND | DAVID SVOBODA

---

Foreword by JAMES A. GOSLING, Father of the Java Programming Language

# Java™ Coding Guidelines

file assumes that the preceding sources reside in the `c:\package` directory on a Windows-based system.

```
grant codeBase "file:/c:/package" {  
    // For *nix, file:${user.home}/package/  
    permission ExceptionReporterPermission "exc.reporter";  
    permission java.lang.RuntimePermission "loadLibrary.myLib";  
};
```

By default, permissions cannot be defined to support actions using `BasicPermission`, but the actions can be freely implemented in the subclass `ExceptionReporterPermission` if required. `BasicPermission` is abstract even though it contains no abstract methods; it defines all the methods that it extends from the `Permission` class. The custom-defined subclass of the `BasicPermission` class must define two constructors to call the most appropriate (one- or two-argument) superclass constructor (because the superclass lacks a default constructor). The two-argument constructor also accepts an action, even though a basic permission does not use it. This behavior is required for constructing permission objects from the policy file. Note that the custom-defined subclass of the `BasicPermission` class is declared to be `final`.

## Applicability

Running Java code without defining custom permissions where default permissions are inapplicable can leave an application open to privilege escalation vulnerabilities.

## Bibliography

- |                |   |
|----------------|---|
| [API 2013]     | Class <code>FilePermission</code>                               |
|                | Class <code>SecurityManager</code>                              |
| [Oaks 2001]    | “Permissions” subsection of Chapter 5, “The Access Controller,” |
| [Oracle 2011d] | Permissions in the Java™ SE 6 Development Kit (JDK)             |
| [Oracle 2013c] | Java Platform Standard Edition 7 Documentation                  |
| [Policy 2010]  | “Permission Descriptions and Risks”                             |

## ■ 20. Create a secure sandbox using a security manager

---

According to the Java API Class `SecurityManager` documentation [API 2013],

The security manager is a class that allows applications to implement a security policy. It allows an application to determine, before performing a possibly unsafe or

sensitive operation, what the operation is and whether it is being attempted in a security context that allows the operation to be performed. The application can allow or disallow the operation.

A security manager may be associated with any Java code.

The applet security manager denies applets all but the most essential privileges. It is designed to protect against inadvertent system modification, information leakage, and user impersonation. The use of security managers is not limited to client-side protection. Web servers, such as Tomcat and WebSphere, use this facility to isolate trojan servlets and malicious Java Server Pages (JSP) as well as to protect sensitive system resources from inadvertent access.

Java applications that run from the command line can set a default or custom security manager using a command-line flag. Alternatively, it is possible to install a security manager programmatically. Installing a security manager programmatically helps create a default sandbox that allows or denies sensitive actions on the basis of the security policy in effect.

From Java 2 SE Platform onward, `SecurityManager` is a nonabstract class. As a result, there is no explicit requirement to override its methods. To create and use a security manager programmatically, the code must have the runtime permissions `createSecurityManager` (to instantiate `SecurityManager`) and `setSecurityManager` (to install it). These permissions are checked only if a security manager is already installed. This is useful for situations in which a default security manager is in place, such as on a virtual host, and individual hosts must be denied the requisite permissions for overriding the default security manager with a custom one.

The security manager is closely tied to the `AccessController` class. The former is used as a hub for access control, whereas the latter provides the actual implementation of the access control algorithm. The security manager supports

- Providing backward compatibility: Legacy code often contains custom implementations of the security manager class because it was originally abstract.
- Defining custom policies: Subclassing the security manager permits definition of custom security policies (for example, multilevel, coarse, or fine grain).

Regarding the implementation and use of custom security managers as opposed to default ones, the Java security architecture specification [SecuritySpec 2010] states:

We encourage the use of `AccessController` in application code, while customization of a security manager (via subclassing) should be the last resort and should be done with extreme care. Moreover, a customized security manager, such as one that always checks the time of the day before invoking standard security checks, could and should utilize the algorithm provided by `AccessController` whenever appropriate.

Many of the Java SE APIs perform security manager checks by default before performing sensitive operations. For example, the constructor of class `java.io`

`.FileInputStream` throws a `SecurityException` if the caller does not have the permission to read a file. Because `SecurityException` is a subclass of `RuntimeException`, the declarations of some API methods (for example, those of the `java.io.FileReader` class) may lack a `throws` clause that lists the `SecurityException`. Avoid depending on the presence or absence of security manager checks that are not specified in the API method's documentation.

## Noncompliant Code Example (Command-Line Installation)

This noncompliant code example fails to install any security manager from the command line. Consequently, the program runs with all permissions enabled; that is, there is no security manager to prevent any nefarious actions the program might perform.

```
java LocalJavaApp
```

## Compliant Solution (Default Policy File)

Any Java program can attempt to install a `SecurityManager` programmatically, although the currently active security manager may forbid this operation. Applications designed to run locally can specify a default security manager by use of a flag on the command line at invocation.

The command-line option is preferred when applications must be prohibited from installing custom security managers programmatically, and are required to abide by the default security policy under all circumstances. This compliant solution installs the default security manager using the appropriate command-line flags. The security policy file grants permissions to the application for its intended actions.

```
java -Djava.security.manager -Djava.security.policy=policyURL \
    LocalJavaApp
```

The command-line flag can specify a custom security manager whose policies are enforced globally. Use the `-Djava.security.manager` flag, as follows:

```
java -Djava.security.manager=my.security.CustomManager ...
```

If the current security policy enforced by the current security manager forbids replacements (by omitting the `RuntimePermission("setSecurityManager")`), any attempt to invoke `setSecurityManager()` will throw a `SecurityException`.

The default security policy file `java.policy`—found in the `/path/to/java.home/lib/security` directory on UNIX-like systems and its equivalent on Microsoft Windows systems—grants a few permissions (reading system properties, binding to unprivileged ports, and so forth). A user-specific policy file may also be located in the

user's home directory. The union of these policy files specifies the permissions granted to a program. The `java.security` file can specify which policy files are used. If either of the systemwide `java.policy` or `java.security` files is deleted, no permissions are granted to the executing Java program.

### Compliant Solution (Custom Policy File)

Use double equals (`==`) instead of the single equals (`=`) when overriding the global Java security policy file with a custom policy file:

```
java -Djava.security.manager \
    -Djava.security.policy==policyURL \
    LocalJavaApp
```

### Compliant Solution (Additional Policy Files)

The `appletviewer` automatically installs a security manager with the standard policy file. To specify additional policy files, use the `-J` flag.

```
appletviewer -J-Djava.security.manager \
    -J-Djava.security.policy==policyURL LocalJavaApp
```

Note that the policy file specified in the argument is ignored when the `policy.allowSystemProperty` property in the security properties file (`java.security`) is set to `false`; the default value of this property is `true`. Default Policy Implementation and Policy File Syntax [Policy 2010] discusses in depth the issues and syntax for writing policy files.

### Noncompliant Code Example (Programmatic Installation)

A `SecurityManager` can also be activated using the static `System.setSecurityManager()` method. Only one `SecurityManager` may be active at a time. This method replaces the currently active `SecurityManager` with the `SecurityManager` provided in its argument, or no `SecurityManager` if its argument is `null`.

This noncompliant code example deactivates any current `SecurityManager` but does not install another `SecurityManager` in its place. Consequently, subsequent code will run with all permissions enabled; there will be no restrictions on any nefarious action the program might perform.

```
try {
    System.setSecurityManager(null);
} catch (SecurityException se) {
    // Cannot set security manager, log to file
}
```

An active `SecurityManager` that enforces a sensible security policy will prevent the system from deactivating it, causing this code to throw a `SecurityException`.

### Compliant Solution (Default Security Manager)

This compliant solution instantiates and sets the default security manager.

```
try {
    System.setSecurityManager(new SecurityManager());
} catch (SecurityException se) {
    // Cannot set security manager, log appropriately
}
```

### Compliant Solution (Custom Security Manager)

This compliant solution demonstrates how to instantiate a custom `SecurityManager` class called `CustomSecurityManager` by invoking its constructor with a password; this custom security manager is then installed as the active security manager.

```
char password[] = /* initialize */
try {
    System.setSecurityManager(
        new CustomSecurityManager("password here")
    );
} catch (SecurityException se) {
    // Cannot set security manager, log appropriately
}
```

After this code executes, APIs that perform security checks will use the custom security manager. As noted earlier, custom security managers should be installed only when the default security manager lacks the required functionality.

### Applicability

Java security fundamentally depends on the existence of a security manager. In its absence, sensitive actions can execute without restriction.

Programmatic detection of the presence or absence of a `SecurityManager` at runtime is straightforward. Static analysis can address the presence or absence of code that would attempt to install a `SecurityManager` if the code were executed. Checking whether the `SecurityManager` is installed early enough, whether it

specifies the desired properties, or whether it is guaranteed to be installed may be possible in some special cases, but is generally undecidable.

Invocation of the `setSecurityManager()` method may be omitted in controlled environments in which it is known that a global-default security manager is *always* installed from the command line. This is difficult to enforce, and can result in vulnerabilities if the environment is incorrectly configured.

## Bibliography

[API 2013]	Class <code>SecurityManager</code> Class <code>AccessControlContext</code> Class <code>AccessController</code>
[Gong 2003]	§6.1, “Security Manager”
[Pistoia 2004]	§7.4, “The Security Manager”
[Policy 2010]	Default Policy Implementation and Policy File Syntax
[SecuritySpec 2010]	§6.2, “ <code>SecurityManager</code> versus <code>AccessController</code> ”

## ■ 21. Do not let untrusted code misuse privileges of callback methods

---

Callbacks provide a means to register a method to be invoked (or *called back*) when an interesting event occurs. Java uses callbacks for applet and servlet life-cycle events, AWT and Swing event notifications such as button clicks, asynchronous reads and writes to storage, and even in `Runnable.run()` wherein a new thread automatically executes the specified `run()` method.

In Java, callbacks are typically implemented using interfaces. The general structure of a callback is as follows:

```
public interface Callback {
    void callMethod();
}

class CallbackImpl implements Callback {
    public void callMethod() {
        System.out.println("Callback invoked");
    }
}

class CallbackAction {
    private Callback callback;
```