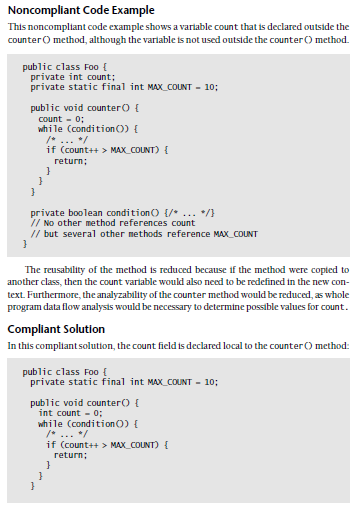
Writing secure Java code is very difficult. There is no magic bullet that will solve your security problems, All you can do is think hard, test your code and use good engineering practices to minimize risks.

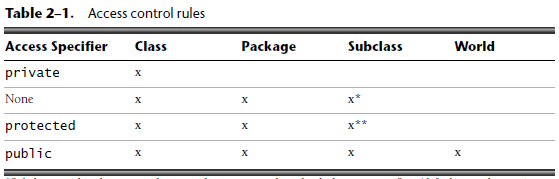
**Java™ Coding Guidelines: 75 Recommendations for Reliable and Secure Programs**

**Relibilility Examples**

* Minimize the scope of variables



* Minimize the accessibility of classes and their members



* Prefer user-defined exceptions over more general exception types

Because an exception is caught by its type, it is better to define exceptions for specific purposes than to use general exception types for multiple purposes. Throwing general exception types makes code hard to understand and maintain, and defeats much of the advantage of the Java exception-handling mechanism.

* Carefully design interfaces before releasing them

In this noncompliant code example, an interface User is frozen with two methods: authenticate() and subscribe(). Sometime later, the providers release a free service that does not rely on authentication.

*public interface User {*

*boolean authenticate(String username, char[] password);*

*void subscribe(int noOfDays);*

*// Introduced after the class is publicly released*

*void freeService();*

*}*

The addition of the *freeService()* method, unfortunately, breaks all the client code that implements the interface. Moreover, the implementers who wish to use only *freeService()* have to face the onus of also providing the other two methods, which pollute the API, for reasons discussed earlier.

* Do not declare more than one variable per declaration

Declaring multiple variables in a single declaration could cause confusion about the types of variables and their initial values. In particular, do not declare any of the following in a single declaration:

-Variables of different types

-A mixture of initialized and uninitialized variables

In general, you should declare each variable on its own line with an explanatory comment regarding its role.

**The CERT® Oracle® Secure Coding Standard for Java**

Extract from **Secure Coding Guidelines for Java SE**

<http://www.oracle.com/technetwork/java/seccodeguide-139067.html#8>  
  
Some Examples:

**Guideline 2-1 / CONFIDENTIAL-1: Purge sensitive information from exceptions**

Exception objects may convey sensitive information. For example, if a method calls the java.io.FileInputStream constructor to read an underlying configuration file and that file is not present, a java.io.FileNotFoundException containing the file path is thrown. Propagating this exception back to the method caller exposes the layout of the file system. Many forms of attack require knowing or guessing locations of files.

Internal exceptions should be caught and sanitized before propagating them to upstream callers. The type of an exception may reveal sensitive information, even if the message has been removed. For instance, FileNotFoundException reveals whether or not a given file exists.

It is sometimes also necessary to sanitize exceptions containing information derived from caller inputs. For example, exceptions related to file access could disclose whether a file exists. An attacker may be able gather useful information by providing various file names as input and analyzing the resulting exceptions.

**3 Injection and Inclusion**

A very common form of attack involves causing a particular program to interpret data crafted in such a way as to cause an unanticipated change of control. Typically, but not always, this involves text formats.

**Guideline 3-2 / INJECT-2: Avoid dynamic SQL**

It is well known that dynamically created SQL statements including untrusted input are subject to command injection. This often takes the form of supplying an input containing a quote character (') followed by SQL. Avoid dynamic SQL.

**Guideline 3-6 / INJECT-6: Care with BMP files**

BMP images files may contain references to local ICC (International Color Consortium) files. Whilst the contents of ICC files is unlikely to be interesting, the act of attempting to read files may be an issue. Either avoid BMP files, or reduce privileges

**Guideline 3-9 / INJECT-9: Prevent injection of exceptional floating point values**

Working with floating point numbers requires care when importing those from outside of a trust boundary, as the NaN (not a number) or infinite values can be injected into applications via untrusted input data

**4 Accessibility and Extensibility**

The task of securing a system is made easier by reducing the "attack surface" of the code.

**Guideline 4-1 / EXTEND-1: Limit the accessibility of classes, interfaces, methods, and fields**

A Java package comprises a grouping of related Java classes and interfaces. Declare any class or interface public if it is specified as part of a published API, otherwise, declare it package-private. Similarly, declare class members and constructors (nested classes, methods, or fields) public or protected as appropriate, if they are also part of the API. Otherwise, declare them private or package-private to avoid exposing the implementation. Note that members of interfaces are implicitly public.

**Guideline 4-5 / EXTEND-5: Limit the extensibility of classes and methods**

Design classes and methods for inheritance or declare them final. Left non-final, a class or method can be maliciously overridden by an attacker. A class that does not permit subclassing is easier to implement and verify that it is secure. Prefer composition to inheritance.

// Unsubclassable class with composed behavior.

public final class SensitiveClass {

private final Behavior behavior;

// Hide constructor.

private SensitiveClass(Behavior behavior) {

this.behavior = behavior;

}

// Guarded construction.

public static SensitiveClass newSensitiveClass(

Behavior behavior

) {

// ... validate any arguments ...

// ... perform security checks ...

return new SensitiveClass(behavior);

}

}

**Guideline 4-6 / EXTEND-6: Understand how a superclass can affect subclass behavior**

Subclasses do not have the ability to maintain absolute control over their own behavior. A superclass can affect subclass behavior by changing the implementation of an inherited method that is not overridden. If a subclass overrides all inherited methods, a superclass can still affect subclass behavior by introducing new methods. Such changes to a superclass can unintentionally break assumptions made in a subclass and lead to subtle security vulnerabilities.

**5 Input Validation**

A feature of the culture of Java is that rigorous method parameter checking is used to improve robustness. More generally, validating external inputs is an important part of security.

**Guideline 5-1 / INPUT-1: Validate inputs**

Input from untrusted sources must be validated before use. Maliciously crafted inputs may cause problems, whether coming through method arguments or external streams. Examples include overflow of integer values and directory traversal attacks by including "../" sequences in filenames.

**6 Mutability**

Mutability, whilst appearing innocuous, can cause a surprising variety of security problems.

**Guideline 6-1 / MUTABLE-1: Prefer immutability for value types**

Making classes immutable prevents the issues associated with mutable objects (described in subsequent guidelines) from arising in client code. Immutable classes should not be subclassable.

**Making a class ‘immutable’**

Don't provide "setter" methods — methods that modify fields or objects referred to by fields.

Make all fields final and private.

Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.

If the instance fields include references to mutable objects, don't allow those objects to be changed:

Don't provide methods that modify the mutable objects.

Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

**8 Serialization and Deserialization**

Java Serialization provides an interface to classes that sidesteps the field access control mechanisms of the Java language (by allowing data be incepted as bytes and thereby changed). As a result, care must be taken when performing serialization and deserialization.

**Guideline 8-1 / SERIAL-1: Avoid serialization for security-sensitive classes**

Security-sensitive classes that are not serializable will not have the problems detailed in this section. Making a class serializable effectively creates a public interface to all fields of that class.