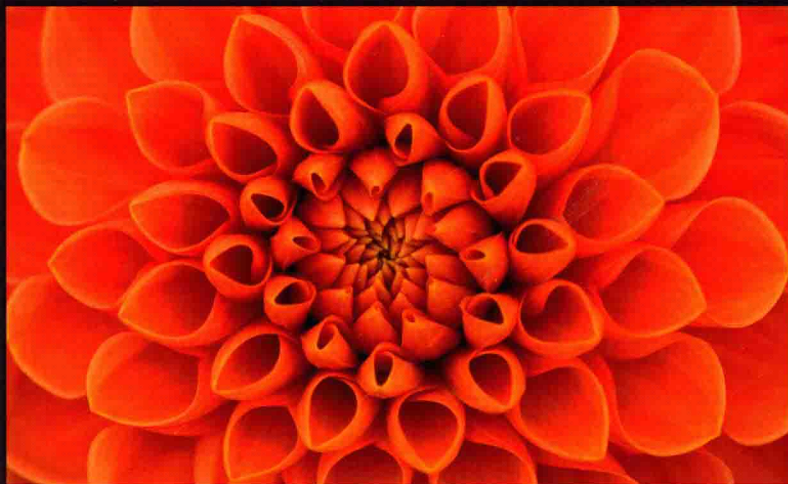


PEARSON



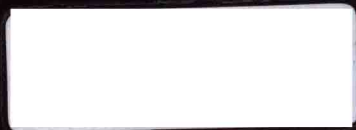
Java编码指南

编写安全可靠程序的75条建议 (英文版)

[美] Fred Long Dhruv Mohindra Robert C. Seacord 著
Dean F. Sutherland David Svoboda

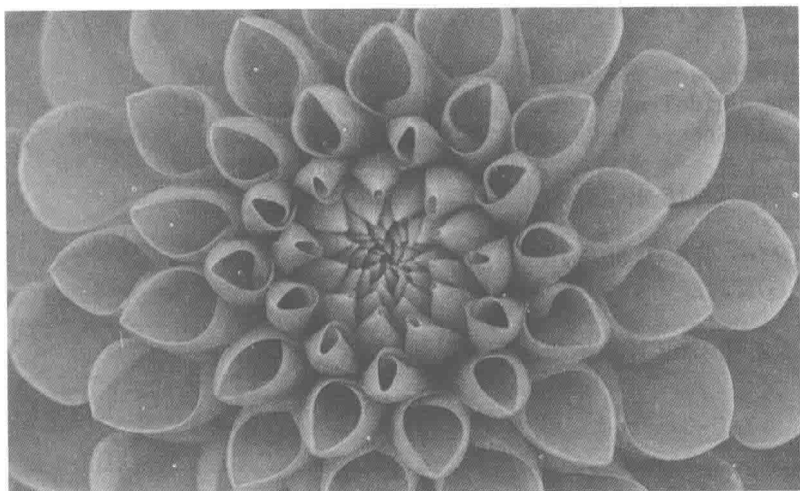
Java Coding Guidelines

75 Recommendations for Reliable and Secure Programs



 中国工信出版集团

 人民邮电出版社
POSTS & TELECOM PRESS



Java编码指南

编写安全可靠程序的75条建议 (英文版)

[美] Fred Long Dhruv Mohindra Robert C. Seacord 著
Dean F. Sutherland David Svoboda

Java Coding Guidelines

75 Recommendations for Reliable and Secure Programs

人民邮电出版社
北 京

图书在版编目 (C I P) 数据

Java编码指南 : 编写安全可靠程序的75条建议 =
Java Coding Guidelines: 75 Recommendations for
Reliable and Secure Programs : 英文 / (美) 朗
(Long, F.) 等著. — 北京 : 人民邮电出版社, 2015. 10
ISBN 978-7-115-40401-5

I. ①J… II. ①朗… III. ①JAVA语言—程序设计—
英文 IV. ①TP312

中国版本图书馆CIP数据核字 (2015) 第222143号

内 容 提 要

本书是《Java 安全编码标准》一书的扩展, 书中把那些不必列入 Java 安全编码标准但是同样会导致系统不可靠或不安全的 Java 编码实践整理出来, 并为这些糟糕的实践提供了相应的文档和警告, 以及合规的解决方案。读者可以将本书作为 Java 安全方面的工具书, 根据自己的需要, 找到自己感兴趣的规则进行阅读和理解, 或者在实际开发中遇到安全问题时, 根据书中列出的大致分类对规则进行索引和阅读, 也可以通读全书的所有规则, 系统地了解 Java 安全规则, 增强对 Java 安全特性、语言使用、运行环境特性的理解。

本书给出了帮助 Java 软件工程师设计出高质量的、安全的、可靠的、强大的、有弹性的、可用性和可维护性高的软件系统的 75 条编码指南, 适合所有 Java 开发人员阅读, 也适合高等院校教师和学生学习和参考。

◆ 著 [美] Fred Long Dhruv Mohindra Robert C. Seacord
Dean F. Sutherland David Svoboda

责任编辑 杨海玲

责任印制 张佳莹 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

固安县铭成印刷有限公司印刷

◆ 开本: 720×960 1/16

印张: 18

字数: 298 千字

2015 年 10 月第 1 版

印数: 1—2 000 册

2015 年 10 月河北第 1 次印刷

著作权合同登记号 图字: 01-2015-6171 号

定价: 59.00 元

读者服务热线: (010)81055410 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京崇工商广字第 0021 号

目录

Chapter 1	Security / 安全	1
1.	Limit the lifetime of sensitive data	2
	限制敏感数据的生命周期	
2.	Do not store unencrypted sensitive information on the client side	5
	不要在客户端存储未经加密的敏感数据	
3.	Provide sensitive mutable classes with unmodifiable wrappers	9
	为敏感可变类提供不可修改的包装器	
4.	Ensure that security-sensitive methods are called with validated arguments	11
	确保安全敏感方法被调用时参数经过验证	
5.	Prevent arbitrary file upload	13
	防止任意文件上传	
6.	Properly encode or escape output	16
	正确地编码或转义输出	
7.	Prevent code injection	20
	防止代码注入	
8.	Prevent XPath injection	23
	防止 XPath 注入	

9. Prevent LDAP injection	27
防止 LDAP 注入	
10. Do not use the clone() method to copy untrusted method parameters	31
不要使用 clone() 方法来复制不可信的方法参数	
11. Do not use Object.equals() to compare cryptographic keys	34
不要使用 Object.equals() 来比较密钥	
12. Do not use insecure or weak cryptographic algorithms	36
不要使用不安全的弱加密算法	
13. Store passwords using a hash function	37
使用散列函数存储密码	
14. Ensure that SecureRandom is properly seeded	42
确保 SecureRandom 正确地选择随机数种子	
15. Do not rely on methods that can be overridden by untrusted code	44
不要依赖可以被不可信代码覆写的方法	
16. Avoid granting excess privileges	50
避免授予过多特权	
17. Minimize privileged code	54
最小化特权代码	
18. Do not expose methods that use reduced-security checks to untrusted code	56
不要将使用降低安全性检查的方法暴露给不可信代码	
19. Define custom security permissions for fine-grained security	64
对细粒度的安全定义自定义安全权限	
20. Create a secure sandbox using a security manager	67
使用安全管理器创建一个安全的沙盒	
21. Do not let untrusted code misuse privileges of callback methods	72
不要让不可信代码误用回调方法的特权	
Chapter 2 Defensive Programming / 防御式编程	79
22. Minimize the scope of variables	80
最小化变量的作用域	

23. Minimize the scope of the <code>@SuppressWarnings</code> annotation	82
最小化 <code>@SuppressWarnings</code> 注解的作用域	
24. Minimize the accessibility of classes and their members	84
最小化类及其成员的可访问性	
25. Document thread-safety and use annotations where applicable	89
文档化代码的线程安全性	
26. Always provide feedback about the resulting value of a method	96
为方法的结果值提供反馈	
27. Identify files using multiple file attributes	99
使用多个文件属性识别文件	
28. Do not attach significance to the ordinal associated with an enum	106
不要赋予枚举常量的序号任何特殊意义	
29. Be aware of numeric promotion behavior	108
注意数字提升行为	
30. Enable compile-time type checking of variable arity parameter types	112
对可变参数的类型做编译时类型检查	
31. Do not apply <code>public final</code> to constants whose value might change in later releases	115
不要把其值在以后版本里可能会发生变化的常量设置为 <code>public final</code>	
32. Avoid cyclic dependencies between packages	118
避免包之间的循环依赖	
33. Prefer user-defined exceptions over more general exception types	121
使用用户自定义的异常而非宽泛的异常类型	
34. Try to gracefully recover from system errors	123
尽量从系统错误中优雅恢复	
35. Carefully design interfaces before releasing them	125
布接口前请谨慎设计	
36. Write garbage collection-friendly code	128
编写对垃圾回收机制友好的代码	

Chapter 3 Reliability / 可靠性	131
37. Do not shadow or obscure identifiers in subscopes 不要在子作用域里遮蔽或者掩盖标识符	132
38. Do not declare more than one variable per declaration 不要在一个声明里声明多个变量	134
39. Use meaningful symbolic constants to represent literal values in program logic 在程序逻辑中用有意义的符号常量代表文字值	138
40. Properly encode relationships in constant definitions 在常量定义中恰当地表示相互之间的关系	142
41. Return an empty array or collection instead of a null value for methods that return an array or collection 对于返回数组或者集合的方法，用返回一个空数组或者 集合来替代返回一个空值	143
42. Use exceptions only for exceptional conditions 只在异常的情况下使用异常	146
43. Use a try-with-resources statement to safely handle closeable resources 使用 try-with-resources 语句安全处理可关闭的资源	148
44. Do not use assertions to verify the absence of runtime errors 不要使用断言来验证不存在的运行时错误	151
45. Use the same type for the second and third operands in conditional expressions 在条件表达式中，第二和第三个操作数应使用相同类型	153
46. Do not serialize direct handles to system resources 不要序列化直接指向系统资源的句柄	157
47. Prefer using iterators over enumerations 更倾向于使用迭代器而不是枚举	159
48. Do not use direct buffers for short-lived, infrequently used objects	162

对于短生存周期、不常用的对象不要使用直接缓冲区

49. Remove short-lived objects from long-lived
container objects 163
从长生存周期容器对象中移除短生存周期对象

Chapter 4 Program Understandability / 程序的可理解性 167

50. Be careful using visually misleading identifiers and literals 167
谨慎使用视觉上有误导性的标识符和文字
51. Avoid ambiguous overloading of variable arity methods 171
避免歧义重载变参方法
52. Avoid in-band error indicators 173
要避免使用带内错误指示器
53. Do not perform assignments in conditional expressions 175
不要在条件表达式中进行赋值
54. Use braces for the body of an if, for, or while statement 178
请使用大括号把 if、for 或 while 代码体括起来
55. Do not place a semicolon immediately following an if, for, or
while condition 180
不要直接在 if、for 或 while 条件语句后面加分号
56. Finish every set of statements associated with a case label with a
break statement 181
在每一个 case 分支的代码块中加上 break 语句
57. Avoid inadvertent wrapping of loop counters 183
避免不当的计算循环计数器
58. Use parentheses for precedence of operation 186
使用括号表示操作的优先级
59. Do not make assumptions about file creation 189
不要对文件的创建做任何假设
60. Convert integers to floating-point for floating-point operations 191
做浮点运算前把整数转换为浮点数
61. Ensure that the clone() method calls super.clone() 194
确保对象的 clone() 方法中有调用 super.clone()

62. Use comments consistently and in a readable fashion	196
保持注释的一致性和可读性	
63. Detect and remove superfluous code and values	198
检测并移除冗余的代码和值	
64. Strive for logical completeness	202
尽量保证逻辑完备	
65. Avoid ambiguous or confusing uses of overloading	205
避免有歧义的重载或者误导性的重载	
 Chapter 5 Programmer Misconceptions / 程序员的常见误解	 209
66. Do not assume that declaring a reference <code>volatile</code> guarantees safe publication of the members of the referenced object	209
不要假设使用 <code>volatile</code> 关键字声明引用可以保证引用所 指对象的安全发布	
67. Do not assume that the <code>sleep()</code> , <code>yield()</code> , or <code>getState()</code> methods provide synchronization semantics	216
不要假设 <code>sleep()</code> 、 <code>yield()</code> 或 <code>getState()</code> 方法提供了同步语义	
68. Do not assume that the remainder operator always returns a nonnegative result for integral operands	220
不要假设对整数做取余运算总是返回正整数	
69. Do not confuse abstract object equality with reference equality	222
不要弄混抽象对象的相等性和引用的相等性	
70. Understand the differences between bitwise and logical operators	225
理解按位运算符和逻辑运算符之间的差异	
71. Understand how escape characters are interpreted when strings are loaded	228
理解加载字符串时如何做特殊字符转义	
72. Do not use overloaded methods to differentiate between runtime types	231
不要使用重载的方法来区分运行时类型	

73. Never confuse the immutability of a reference with that of the referenced object	234
不要弄混引用的不可变性和对象的不可变性	
74. Use the serialization methods <code>writeUnshared()</code> and <code>readUnshared()</code> with care	239
谨慎使用序列化方法 <code>writeUnshared()</code> 和 <code>readUnshared()</code>	
75. Do not attempt to help the garbage collector by setting local reference variables to <code>null</code>	243
不要试图通过把本地引用变量设置为 <code>null</code> 来帮助垃圾收集器	
Appendix A Android	245
Glossary / 术语表	249
References / 参考文献	255

Chapter 1

Security

The Java programming language and runtime system were designed with security in mind. For example, pointer manipulation is implicit and hidden from the programmer, and any attempt to reference a null pointer results in an exception being thrown. Similarly, an exception results from any attempt to access an array or a string outside of its bounds. Java is a strongly typed language, and all implicit type conversions are well defined and platform independent, as are the arithmetic types and conversions. The Java Virtual Machine (JVM) has a built-in bytecode verifier to ensure that the bytecode being run conforms to the Java Language Specification: Java SE 7 Edition (JLS) so that all the checks defined in the language are in place and cannot be circumvented.

The Java class loader mechanism identifies classes as they are loaded into the JVM, and can distinguish between trusted system classes and other classes that may not be trusted. Classes from external sources can be given privileges by digitally signing them; these digital signatures can also be examined by the class loader, and contribute to the class's identification. Java also provides an extensible fine-grained security mechanism that enables the programmer to control access to resources such as system information, files, sockets, and any other security-sensitive resources that the programmer wishes to use. This security mechanism can require that a runtime security manager be in place to enforce a security policy. A security manager and its security policy are usually specified by command-line arguments, but they may be installed programmatically, provided that such an action is not already disallowed by an existing security policy. Privileges to access resources may be extended to nonsystem Java classes by relying on the identification provided by the class loader mechanism.

Enterprise Java applications are susceptible to attacks because they accept untrusted input and interact with complex subsystems. Injection attacks (such as cross-site scripting [XSS], XPath, and LDAP injection) are possible when the components susceptible to these attacks are used in the application. An effective mitigation strategy is to whitelist input, and encode or escape output before it is processed for rendering.

This chapter contains guidelines that are concerned specifically with ensuring the security of Java-based applications. Guidelines dealing with the following security nuances are articulated.

1. Dealing with sensitive data
2. Avoiding common injection attacks
3. Language features that can be misused to compromise security
4. Details of Java's fine-grained security mechanism

■ 1. Limit the lifetime of sensitive data

Sensitive data in memory can be vulnerable to compromise. An adversary who can execute code on the same system as an application may be able to access such data if the application

- Uses objects to store sensitive data whose contents are not cleared or garbage-collected after use
- Has memory pages that can be swapped out to disk as required by the operating system (for example, to perform memory management tasks or to support hibernation)
- Holds sensitive data in a buffer (such as `BufferedReader`) that retains copies of the data in the OS cache or in memory
- Bases its control flow on reflection that allows countermeasures to circumvent the limiting of the lifetime of sensitive variables
- Reveals sensitive data in debugging messages, log files, environment variables, or through thread and core dumps

Sensitive data leaks become more likely if the memory containing the data is not cleared after using the data. To limit the risk of exposure, programs must minimize the lifetime of sensitive data.

Complete mitigation (that is, foolproof protection of data in memory) requires support from the underlying operating system and Java Virtual Machine. For example, if swapping sensitive data out to disk is an issue, a secure operating system that disables swapping and hibernation is required.

Noncompliant Code Example

This noncompliant code example reads user name and password information from the console and stores the password as a `String` object. The credentials remain exposed until the garbage collector reclaims the memory associated with this `String`.

```
class Password {
    public static void main (String args[]) throws IOException {
        Console c = System.console();
        if (c == null) {
            System.err.println("No console.");
            System.exit(1);
        }

        String username = c.readLine("Enter your user name: ");
        String password = c.readLine("Enter your password: ");

        if (!verify(username, password)) {
            throw new SecurityException("Invalid Credentials");
        }

        // ...
    }

    // Dummy verify method, always returns true
    private static final boolean verify(String username,
        String password) {
        return true;
    }
}
```

Compliant Solution

This compliant solution uses the `Console.readPassword()` method to obtain the password from the console.

```
class Password {
    public static void main (String args[]) throws IOException {
        Console c = System.console();

        if (c == null) {
            System.err.println("No console.");
            System.exit(1);
        }

        String username = c.readLine("Enter your user name: ");
        char[] password = c.readPassword("Enter your password: ");
    }
}
```

```
    if (!verify(username, password)) {
        throw new SecurityException("Invalid Credentials");
    }

    // Clear the password
    Arrays.fill(password, ' ');
}

// Dummy verify method, always returns true
private static final boolean verify(String username,
    char[] password) {
    return true;
}
}
```

The `Console.readPassword()` method allows the password to be returned as a sequence of characters rather than as a `String` object. Consequently, the programmer can clear the password from the array immediately after use. This method also disables echoing of the password to the console.

Noncompliant Code Example

This noncompliant code example uses a `BufferedReader` to wrap an `InputStreamReader` object so that sensitive data can be read from a file:

```
void readData() throws IOException{
    BufferedReader br = new BufferedReader(new InputStreamReader(
        new FileInputStream("file")));
    // Read from the file
    String data = br.readLine();
}
```

The `BufferedReader.readLine()` method returns the sensitive data as a `String` object, which can persist long after the data is no longer needed. The `BufferedReader.read(char[], int, int)` method can read and populate a `char` array. However, it requires the programmer to manually clear the sensitive data in the array after use. Alternatively, even if the `BufferedReader` were to wrap a `FileReader` object, it would suffer from the same pitfalls.

Compliant Solution

This compliant solution uses a directly allocated NIO (new I/O) buffer to read sensitive data from the file. The data can be cleared immediately after use and is not cached or buffered in multiple locations. It exists only in the system memory.

```
void readData(){
    ByteBuffer buffer = ByteBuffer.allocateDirect(16 * 1024);
    try (FileChannel rdr =
        (new FileInputStream("file")).getChannel()) {
        while (rdr.read(buffer) > 0) {
            // Do something with the buffer
            buffer.clear();
        }
    } catch (Throwable e) {
        // Handle error
    }
}
```

Note that manual clearing of the buffer data is mandatory because direct buffers are not garbage collected.

Applicability

Failure to limit the lifetime of sensitive data can lead to information leaks.

Bibliography

[API 2013]	Class <code>ByteBuffer</code>
[Oracle 2013b]	“Reading ASCII Passwords from an <code>InputStream</code> Example” from the Java Cryptography Architecture [JCA] Reference Guide
[Tutorials 2013]	I/O from the Command Line

■ 2. Do not store unencrypted sensitive information on the client side

When building an application that uses a client–server model, storing sensitive information, such as user credentials, on the client side may result in its unauthorized disclosure if the client is vulnerable to attack.

For web applications, the most common mitigation to this problem is to provide the client with a cookie and store the sensitive information on the server. Cookies are created by a web server, and are stored for a period of time on the client. When the client reconnects to the server, it provides the cookie, which identifies the client to the server, and the server then provides the sensitive information.

Cookies do not protect sensitive information against cross-site scripting (XSS) attacks. An attacker who is able to obtain a cookie either through an XSS attack, or directly by attacking the client, can obtain the sensitive information from the server

using the cookie. This risk is timeboxed if the server invalidates the session after a limited time has elapsed, such as 15 minutes.

A cookie is typically a short string. If it contains sensitive information, that information should be encrypted. Sensitive information includes user names, passwords, credit card numbers, social security numbers, and any other personally identifiable information about the user. For more details about managing passwords, see Guideline 13, “Store passwords using a hash function.” For more information about securing the memory that holds sensitive information, see Guideline 1, “Limit the lifetime of sensitive data.”

Noncompliant Code Example

In this noncompliant code example, the login servlet stores the user name and password in the cookie to identify the user for subsequent requests:

```
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) {

    // Validate input (omitted)

    String username = request.getParameter("username");
    char[] password =
        request.getParameter("password").toCharArray();
    boolean rememberMe =
        Boolean.valueOf(request.getParameter("rememberme"));

    LoginService loginService = new LoginServiceImpl();

    if (rememberMe) {
        if (request.getCookies()[0] != null &&
            request.getCookies()[0].getValue() != null) {
            String[] value =
                request.getCookies()[0].getValue().split(";");

            if (!loginService.isUserValid(value[0],
                value[1].toCharArray())) {
                // Set error and return
            } else {
                // Forward to welcome page
            }
        } else {
            boolean validated =
                loginService.isUserValid(username, password);
```



```
        if (validated) {
            Cookie loginCookie = new Cookie("rememberme", username +
                                           ";" + new String(password));
            response.addCookie(loginCookie);
            // ... forward to welcome page
        } else {
            // Set error and return
        }
    }
} else {
    // No remember-me functionality selected
    // Proceed with regular authentication;
    // if it fails set error and return
}

Arrays.fill(password, ' ');
}
```

However, the attempt to implement the remember-me functionality is insecure because an attacker with access to the client machine can obtain this information directly on the client. This code also violates Guideline 13, “Store passwords using a hash function.”

Compliant Solution (Session)

This compliant solution implements the remember-me functionality by storing the user name and a secure random string in the cookie. It also maintains state in the session using `HttpSession`:

```
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) {

    // Validate input (omitted)

    String username = request.getParameter("username");
    char[] password =
        request.getParameter("password").toCharArray();
    boolean rememberMe =
        Boolean.valueOf(request.getParameter("rememberme"));
    LoginService loginService = new LoginServiceImpl();
    boolean validated = false;
    if (rememberMe) {
        if (request.getCookies()[0] != null &&
            request.getCookies()[0].getValue() != null) {
```