|  |
| --- |
| jedit  Version 1.0  Code analysis |

|  |
| --- |
| By: default  2020-01-13 |

# Content

[Content 1](#_Toc7772423)

[Introduction 2](#_Toc7772424)

[Configuration 2](#_Toc7772425)

[Synthesis 2](#_Toc7772426)

[Metrics 2](#_Toc7772427)

[Volume 3](#_Toc7772428)

[Issues count by severity and type 3](#_Toc7772429)

[Charts 3](#_Toc7772430)

[Issues 3](#_Toc7772431)

# Introduction

This document contains results of the code analysis of jedit.

# Configuration

* Quality Profiles
  + Names: Sonar way [Java]; Sonar way [HTML]; Sonar way [XML];
  + Files: AW-PBFvdC6WUNCuIjV0t.json; AW-PBF2NC6WUNCuIjV7S.json; AW-PBF7bC6WUNCuIjV70.json;
* Quality Gate
  + Name: Sonar way
  + File: Sonar way.xml

# Synthesis

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Quality Gate | Reliability | Security | Maintainability | Coverage | Duplication |
| ERROR | E | E | A | 0.0 % | 2.1 % |

# Metrics

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Cyclomatic  Complexity | Cognitive  Complexity | Lines of code per file | Comment  density (%) | Coverage | Duplication (%) |
| Min | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| Max | 22373.0 | 23616.0 | 115362.0 | 84.6 | 0.0 | 39.0 |

# Volume

|  |  |
| --- | --- |
| Language | Number |
| Java | 112890 |
| HTML | 107 |
| XML | 2365 |
| Total | 115362 |

# Issues count by severity and type

|  |  |  |
| --- | --- | --- |
| Type | Severity | Number |
| VULNERABILITY | BLOCKER | 3 |
| VULNERABILITY | CRITICAL | 0 |
| VULNERABILITY | MAJOR | 0 |
| VULNERABILITY | MINOR | 255 |
| VULNERABILITY | INFO | 0 |
| BUG | BLOCKER | 14 |
| BUG | CRITICAL | 4 |
| BUG | MAJOR | 151 |
| BUG | MINOR | 48 |
| BUG | INFO | 0 |
| CODE\_SMELL | BLOCKER | 26 |
| CODE\_SMELL | CRITICAL | 810 |
| CODE\_SMELL | MAJOR | 6082 |
| CODE\_SMELL | MINOR | 2504 |
| CODE\_SMELL | INFO | 68 |
| SECURITY\_HOTSPOT | BLOCKER | 0 |
| SECURITY\_HOTSPOT | CRITICAL | 0 |
| SECURITY\_HOTSPOT | MAJOR | 0 |
| SECURITY\_HOTSPOT | MINOR | 0 |
| SECURITY\_HOTSPOT | INFO | 0 |

# Charts

# Issues

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Type | Severity | Number |
| Resources should be closed | Connections, streams, files, and other classes that implement the Closeable interface or its super-interface, AutoCloseable, needs to be closed after use. Further, that close call must be made in a finally block otherwise an exception could keep the call from being made. Preferably, when class implements AutoCloseable, resource should be created using "try-with-resources" pattern and will be closed automatically. Failure to properly close resources will result in a resource leak which could bring first the application and then perhaps the box it's on to their knees. Noncompliant Code Example private void readTheFile() throws IOException { Path path = Paths.get(this.fileName); BufferedReader reader = Files.newBufferedReader(path, this.charset); // ... reader.close(); // Noncompliant // ... Files.lines("input.txt").forEach(System.out::println); // Noncompliant: The stream needs to be closed } private void doSomething() { OutputStream stream = null; try { for (String property : propertyList) { stream = new FileOutputStream("myfile.txt"); // Noncompliant // ... } } catch (Exception e) { // ... } finally { stream.close(); // Multiple streams were opened. Only the last is closed. } } Compliant Solution private void readTheFile(String fileName) throws IOException { Path path = Paths.get(fileName); try (BufferedReader reader = Files.newBufferedReader(path, StandardCharsets.UTF\_8)) { reader.readLine(); // ... } // .. try (Stream&lt;String&gt; input = Files.lines("input.txt")) { input.forEach(System.out::println); } } private void doSomething() { OutputStream stream = null; try { stream = new FileOutputStream("myfile.txt"); for (String property : propertyList) { // ... } } catch (Exception e) { // ... } finally { stream.close(); } } Exceptions Instances of the following classes are ignored by this rule because close has no effect: java.io.ByteArrayOutputStream java.io.ByteArrayInputStream java.io.CharArrayReader java.io.CharArrayWriter java.io.StringReader java.io.StringWriter Java 7 introduced the try-with-resources statement, which implicitly closes Closeables. All resources opened in a try-with-resources statement are ignored by this rule. try (BufferedReader br = new BufferedReader(new FileReader(fileName))) { //... } catch ( ... ) { //... } See MITRE, CWE-459 - Incomplete Cleanup CERT, FIO04-J. - Release resources when they are no longer needed CERT, FIO42-C. - Close files when they are no longer needed Try With Resources | BUG | BLOCKER | 13 |
| Loops should not be infinite | An infinite loop is one that will never end while the program is running, i.e., you have to kill the program to get out of the loop. Whether it is by meeting the loop's end condition or via a break, every loop should have an end condition. Noncompliant Code Example for (;;) { // Noncompliant; end condition omitted // ... } int j; while (true) { // Noncompliant; end condition omitted j++; } int k; boolean b = true; while (b) { // Noncompliant; b never written to in loop k++; } Compliant Solution int j; while (true) { // reachable end condition added j++; if (j == Integer.MIN\_VALUE) { // true at Integer.MAX\_VALUE +1 break; } } int k; boolean b = true; while (b) { k++; b = k &lt; Integer.MAX\_VALUE; } See CERT, MSC01-J. - Do not use an empty infinite loop | BUG | BLOCKER | 1 |
| Jump statements should not occur in "finally" blocks | Using return, break, throw, and so on from a finally block suppresses the propagation of any unhandled Throwable which was thrown in the try or catch block. This rule raises an issue when a jump statement (break, continue, return, throw, and goto) would force control flow to leave a finally block. Noncompliant Code Example public static void main(String[] args) { try { doSomethingWhichThrowsException(); System.out.println("OK"); // incorrect "OK" message is printed } catch (RuntimeException e) { System.out.println("ERROR"); // this message is not shown } } public static void doSomethingWhichThrowsException() { try { throw new RuntimeException(); } finally { for (int i = 0; i &lt; 10; i ++) { //... if (q == i) { break; // ignored } } /\* ... \*/ return; // Noncompliant - prevents the RuntimeException from being propagated } } Compliant Solution public static void main(String[] args) { try { doSomethingWhichThrowsException(); System.out.println("OK"); } catch (RuntimeException e) { System.out.println("ERROR"); // "ERROR" is printed as expected } } public static void doSomethingWhichThrowsException() { try { throw new RuntimeException(); } finally { for (int i = 0; i &lt; 10; i ++) { //... if (q == i) { break; // ignored } } /\* ... \*/ } } See MITRE, CWE-584 - Return Inside Finally Block CERT, ERR04-J. - Do not complete abruptly from a finally block | BUG | CRITICAL | 3 |
| Getters and setters should access the expected fields | Getters and setters provide a way to enforce encapsulation by providing public methods that give controlled access to private fields. However in classes with multiple fields it is not unusual that cut and paste is used to quickly create the needed getters and setters, which can result in the wrong field being accessed by a getter or setter. This rule raises an issue in any of these cases: A setter does not update the field with the corresponding name. A getter does not access the field with the corresponding name. Noncompliant Code Example class A { private int x; private int y; public void setX(int val) { // Noncompliant: field 'x' is not updated this.y = val; } public int getY() { // Noncompliant: field 'y' is not used in the return value return this.x; } } Compliant Solution class A { private int x; private int y; public void setX(int val) { this.x = val; } public int getY() { return this.y; } } | BUG | CRITICAL | 1 |
| "<!DOCTYPE>" declarations should appear before "<html>" tags |  | BUG | MAJOR | 21 |
| "<title>" should be present in all pages |  | BUG | MAJOR | 20 |
| "<html>" element should have a language attribute |  | BUG | MAJOR | 21 |
| Elements deprecated in HTML5 should not be used | With the advent of HTML5, many old elements were deprecated. To ensure the best user experience, deprecated elements should not be used. This rule checks for the following deprecated elements: Element Remediation Action basefont, big, blink, center, font, marquee, multicol, nobr, spacer, tt use CSS acronym use abbr applet use embed or object bgsound use audio frame, frameset, noframes restructure the page to remove frames isindex use form controls dir use ul hgroup use header or div listing use pre and code nextid use GUIDS noembed use object instead of embed when fallback is necessary plaintext use the "text/plain" MIME type strike use del or s xmp use pre or code, and escape "&lt;" and "&amp;" characters See W3C, Obsolete Features WHATWG, Obsolete Features | BUG | MAJOR | 1 |
| "equals" method overrides should accept "Object" parameters | "equals" as a method name should be used exclusively to override Object.equals(Object) to prevent any confusion. It is tempting to overload the method to take a specific class instead of Object as parameter, to save the class comparison check. However, this will not work as expected when that is the only override. Noncompliant Code Example class MyClass { private int foo = 1; public boolean equals(MyClass o) { // Noncompliant; does not override Object.equals(Object) return o != null &amp;&amp; o.foo == this.foo; } public static void main(String[] args) { MyClass o1 = new MyClass(); Object o2 = new MyClass(); System.out.println(o1.equals(o2)); // Prints "false" because o2 an Object not a MyClass } } class MyClass2 { public boolean equals(MyClass2 o) { // Ignored; `boolean equals(Object)` also present //.. } public boolean equals(Object o) { //... } } Compliant Solution class MyClass { private int foo = 1; @Override public boolean equals(Object o) { if (this == o) { return true; } if (o == null || getClass() != o.getClass()) { return false; } MyClass other = (MyClass)o; return this.foo == other.foo; } /\* ... \*/ } class MyClass2 { public boolean equals(MyClass2 o) { //.. } public boolean equals(Object o) { //... } } | BUG | MAJOR | 1 |
| Non-serializable classes should not be written | Nothing in a non-serializable class will be written out to file, and attempting to serialize such a class will result in an exception being thrown. Only a class that implements Serializable or one that extends such a class can successfully be serialized (or de-serialized). Noncompliant Code Example public class Vegetable { // neither implements Serializable nor extends a class that does //... } public class Menu { public void meal() throws IOException { Vegetable veg; //... FileOutputStream fout = new FileOutputStream(veg.getName()); ObjectOutputStream oos = new ObjectOutputStream(fout); oos.writeObject(veg); // Noncompliant. Nothing will be written } } Compliant Solution public class Vegetable implements Serializable { // can now be serialized //... } public class Menu { public void meal() throws IOException { Vegetable veg; //... FileOutputStream fout = new FileOutputStream(veg.getName()); ObjectOutputStream oos = new ObjectOutputStream(fout); oos.writeObject(veg); } } | BUG | MAJOR | 2 |
| "InterruptedException" should not be ignored | InterruptedExceptions should never be ignored in the code, and simply logging the exception counts in this case as "ignoring". The throwing of the InterruptedException clears the interrupted state of the Thread, so if the exception is not handled properly the fact that the thread was interrupted will be lost. Instead, InterruptedExceptions should either be rethrown - immediately or after cleaning up the method's state - or the thread should be re-interrupted by calling Thread.interrupt() even if this is supposed to be a single-threaded application. Any other course of action risks delaying thread shutdown and loses the information that the thread was interrupted - probably without finishing its task. Similarly, the ThreadDeath exception should also be propagated. According to its JavaDoc: If ThreadDeath is caught by a method, it is important that it be rethrown so that the thread actually dies. Noncompliant Code Example public void run () { try { while (true) { // do stuff } }catch (InterruptedException e) { // Noncompliant; logging is not enough LOGGER.log(Level.WARN, "Interrupted!", e); } } Compliant Solution public void run () { try { while (true) { // do stuff } }catch (InterruptedException e) { LOGGER.log(Level.WARN, "Interrupted!", e); // Restore interrupted state... Thread.currentThread().interrupt(); } } See MITRE, CWE-391 - Unchecked Error Condition Dealing with InterruptedException | BUG | MAJOR | 9 |
| Dissimilar primitive wrappers should not be used with the ternary operator without explicit casting | If wrapped primitive values (e.g. Integers and Floats) are used in a ternary operator (e.g. a?b:c), both values will be unboxed and coerced to a common type, potentially leading to unexpected results. To avoid this, add an explicit cast to a compatible type. Noncompliant Code Example Integer i = 123456789; Float f = 1.0f; Number n = condition ? i : f; // Noncompliant; i is coerced to float. n = 1.23456792E8 Compliant Solution Integer i = 123456789; Float f = 1.0f; Number n = condition ? (Number) i : f; // n = 123456789 | BUG | MAJOR | 2 |
| "toString()" and "clone()" methods should not return null | Calling toString() or clone() on an object should always return a string or an object. Returning null instead contravenes the method's implicit contract. Noncompliant Code Example public String toString () { if (this.collection.isEmpty()) { return null; // Noncompliant } else { // ... Compliant Solution public String toString () { if (this.collection.isEmpty()) { return ""; } else { // ... See MITRE CWE-476 - NULL Pointer Dereference CERT, EXP01-J. - Do not use a null in a case where an object is required | BUG | MAJOR | 2 |
| Null pointers should not be dereferenced | A reference to null should never be dereferenced/accessed. Doing so will cause a NullPointerException to be thrown. At best, such an exception will cause abrupt program termination. At worst, it could expose debugging information that would be useful to an attacker, or it could allow an attacker to bypass security measures. Note that when they are present, this rule takes advantage of @CheckForNull and @Nonnull annotations defined in JSR-305 to understand which values are and are not nullable except when @Nonnull is used on the parameter to equals, which by contract should always work with null. Noncompliant Code Example @CheckForNull String getName(){...} public boolean isNameEmpty() { return getName().length() == 0; // Noncompliant; the result of getName() could be null, but isn't null-checked } Connection conn = null; Statement stmt = null; try{ conn = DriverManager.getConnection(DB\_URL,USER,PASS); stmt = conn.createStatement(); // ... }catch(Exception e){ e.printStackTrace(); }finally{ stmt.close(); // Noncompliant; stmt could be null if an exception was thrown in the try{} block conn.close(); // Noncompliant; conn could be null if an exception was thrown } private void merge(@Nonnull Color firstColor, @Nonnull Color secondColor){...} public void append(@CheckForNull Color color) { merge(currentColor, color); // Noncompliant; color should be null-checked because merge(...) doesn't accept nullable parameters } void paint(Color color) { if(color == null) { System.out.println("Unable to apply color " + color.toString()); // Noncompliant; NullPointerException will be thrown return; } ... } See MITRE, CWE-476 - NULL Pointer Dereference CERT, EXP34-C. - Do not dereference null pointers CERT, EXP01-J. - Do not use a null in a case where an object is required | BUG | MAJOR | 26 |
| "notifyAll" should be used | notify and notifyAll both wake up sleeping threads, but notify only rouses one, while notifyAll rouses all of them. Since notify might not wake up the right thread, notifyAll should be used instead. Noncompliant Code Example class MyThread extends Thread{ @Override public void run(){ synchronized(this){ // ... notify(); // Noncompliant } } } Compliant Solution class MyThread extends Thread{ @Override public void run(){ synchronized(this){ // ... notifyAll(); } } } See CERT, THI02-J. - Notify all waiting threads rather than a single thread | BUG | MAJOR | 1 |
| Conditionally executed blocks should be reachable | Conditional expressions which are always true or false can lead to dead code. Such code is always buggy and should never be used in production. Noncompliant Code Example a = false; if (a) { // Noncompliant doSomething(); // never executed } if (!a || b) { // Noncompliant; "!a" is always "true", "b" is never evaluated doSomething(); } else { doSomethingElse(); // never executed } Exceptions This rule will not raise an issue in either of these cases: When the condition is a single final boolean final boolean debug = false; //... if (debug) { // Print something } When the condition is literally true or false. if (true) { // do something } In these cases it is obvious the code is as intended. See MITRE, CWE-570 - Expression is Always False MITRE, CWE-571 - Expression is Always True CERT, MSC12-C. - Detect and remove code that has no effect or is never executed | BUG | MAJOR | 28 |
| Non-thread-safe fields should not be static | Not all classes in the standard Java library were written to be thread-safe. Using them in a multi-threaded manner is highly likely to cause data problems or exceptions at runtime. This rule raises an issue when an instance of Calendar, DateFormat, javax.xml.xpath.XPath, or javax.xml.validation.SchemaFactory is marked static. Noncompliant Code Example public class MyClass { private static SimpleDateFormat format = new SimpleDateFormat("HH-mm-ss"); // Noncompliant private static Calendar calendar = Calendar.getInstance(); // Noncompliant Compliant Solution public class MyClass { private SimpleDateFormat format = new SimpleDateFormat("HH-mm-ss"); private Calendar calendar = Calendar.getInstance(); | BUG | MAJOR | 2 |
| Strings and Boxed types should be compared using "equals()" | It's almost always a mistake to compare two instances of java.lang.String or boxed types like java.lang.Integer using reference equality == or !=, because it is not comparing actual value but locations in memory. Noncompliant Code Example String firstName = getFirstName(); // String overrides equals String lastName = getLastName(); if (firstName == lastName) { ... }; // Non-compliant; false even if the strings have the same value Compliant Solution String firstName = getFirstName(); String lastName = getLastName(); if (firstName != null &amp;&amp; firstName.equals(lastName)) { ... }; See MITRE, CWE-595 - Comparison of Object References Instead of Object Contents MITRE, CWE-597 - Use of Wrong Operator in String Comparison CERT, EXP03-J. - Do not use the equality operators when comparing values of boxed primitives CERT, EXP50-J. - Do not confuse abstract object equality with reference equality | BUG | MAJOR | 15 |
| "equals(Object obj)" and "hashCode()" should be overridden in pairs | According to the Java Language Specification, there is a contract between equals(Object) and hashCode(): If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables. In order to comply with this contract, those methods should be either both inherited, or both overridden. Noncompliant Code Example class MyClass { // Noncompliant - should also override "hashCode()" @Override public boolean equals(Object obj) { /\* ... \*/ } } Compliant Solution class MyClass { // Compliant @Override public boolean equals(Object obj) { /\* ... \*/ } @Override public int hashCode() { /\* ... \*/ } } See MITRE, CWE-581 - Object Model Violation: Just One of Equals and Hashcode Defined CERT, MET09-J. - Classes that define an equals() method must also define a hashCode() method | BUG | MINOR | 10 |
| "Serializable" inner classes of non-serializable classes should be "static" | Serializing a non-static inner class will result in an attempt at serializing the outer class as well. If the outer class is not serializable, then serialization will fail, resulting in a runtime error. Making the inner class static (i.e. "nested") avoids this problem, therefore inner classes should be static if possible. However, you should be aware that there are semantic differences between an inner class and a nested one: an inner class can only be instantiated within the context of an instance of the outer class. a nested (static) class can be instantiated independently of the outer class. Noncompliant Code Example public class Pomegranate { // ... public class Seed implements Serializable { // Noncompliant; serialization will fail // ... } } Compliant Solution public class Pomegranate { // ... public static class Seed implements Serializable { // ... } } See CERT SER05-J. - Do not serialize instances of inner classes | BUG | MINOR | 1 |
| "equals(Object obj)" should test argument type | Because the equals method takes a generic Object as a parameter, any type of object may be passed to it. The method should not assume it will only be used to test objects of its class type. It must instead check the parameter's type. Noncompliant Code Example public boolean equals(Object obj) { MyClass mc = (MyClass)obj; // Noncompliant // ... } Compliant Solution public boolean equals(Object obj) { if (obj == null) return false; if (this.getClass() != obj.getClass()) return false; MyClass mc = (MyClass)obj; // ... } | BUG | MINOR | 1 |
| Boxing and unboxing should not be immediately reversed | Boxing is the process of putting a primitive value into an analogous object, such as creating an Integer to hold an int value. Unboxing is the process of retrieving the primitive value from such an object. Since the original value is unchanged during boxing and unboxing, there's no point in doing either when not needed. This also applies to autoboxing and auto-unboxing (when Java implicitly handles the primitive/object transition for you). Noncompliant Code Example public void examineInt(int a) { //... } public void examineInteger(Integer a) { // ... } public void func() { int i = 0; Integer iger1 = Integer.valueOf(0); double d = 1.0; int dIntValue = new Double(d).intValue(); // Noncompliant examineInt(new Integer(i).intValue()); // Noncompliant; explicit box/unbox examineInt(Integer.valueOf(i)); // Noncompliant; boxed int will be auto-unboxed examineInteger(i); // Compliant; value is boxed but not then unboxed examineInteger(iger1.intValue()); // Noncompliant; unboxed int will be autoboxed Integer iger2 = new Integer(iger1); // Noncompliant; unnecessary unboxing, value can be reused } Compliant Solution public void examineInt(int a) { //... } public void examineInteger(Integer a) { // ... } public void func() { int i = 0; Integer iger1 = Integer.valueOf(0); double d = 1.0; int dIntValue = (int) d; examineInt(i); examineInteger(i); examineInteger(iger1); } | BUG | MINOR | 26 |
| Math operands should be cast before assignment | When arithmetic is performed on integers, the result will always be an integer. You can assign that result to a long, double, or float with automatic type conversion, but having started as an int or long, the result will likely not be what you expect. For instance, if the result of int division is assigned to a floating-point variable, precision will have been lost before the assignment. Likewise, if the result of multiplication is assigned to a long, it may have already overflowed before the assignment. In either case, the result will not be what was expected. Instead, at least one operand should be cast or promoted to the final type before the operation takes place. Noncompliant Code Example float twoThirds = 2/3; // Noncompliant; int division. Yields 0.0 long millisInYear = 1\_000\*3\_600\*24\*365; // Noncompliant; int multiplication. Yields 1471228928 long bigNum = Integer.MAX\_VALUE + 2; // Noncompliant. Yields -2147483647 long bigNegNum = Integer.MIN\_VALUE-1; //Noncompliant, gives a positive result instead of a negative one. Date myDate = new Date(seconds \* 1\_000); //Noncompliant, won't produce the expected result if seconds &gt; 2\_147\_483 ... public long compute(int factor){ return factor \* 10\_000; //Noncompliant, won't produce the expected result if factor &gt; 214\_748 } public float compute2(long factor){ return factor / 123; //Noncompliant, will be rounded to closest long integer } Compliant Solution float twoThirds = 2f/3; // 2 promoted to float. Yields 0.6666667 long millisInYear = 1\_000L\*3\_600\*24\*365; // 1000 promoted to long. Yields 31\_536\_000\_000 long bigNum = Integer.MAX\_VALUE + 2L; // 2 promoted to long. Yields 2\_147\_483\_649 long bigNegNum = Integer.MIN\_VALUE-1L; // Yields -2\_147\_483\_649 Date myDate = new Date(seconds \* 1\_000L); ... public long compute(int factor){ return factor \* 10\_000L; } public float compute2(long factor){ return factor / 123f; } or float twoThirds = (float)2/3; // 2 cast to float long millisInYear = (long)1\_000\*3\_600\*24\*365; // 1\_000 cast to long long bigNum = (long)Integer.MAX\_VALUE + 2; long bigNegNum = (long)Integer.MIN\_VALUE-1; Date myDate = new Date((long)seconds \* 1\_000); ... public long compute(long factor){ return factor \* 10\_000; } public float compute2(float factor){ return factor / 123; } See MITRE, CWE-190 - Integer Overflow or Wraparound CERT, NUM50-J. - Convert integers to floating point for floating-point operations CERT, INT18-C. - Evaluate integer expressions in a larger size before comparing or assigning to that size SANS Top 25 - Risky Resource Management | BUG | MINOR | 6 |
| "@NonNull" values should not be set to null | Fields, parameters and return values marked @NotNull, @NonNull, or @Nonnull are assumed to have non-null values and are not typically null-checked before use. Therefore setting one of these values to null, or failing to set such a class field in a constructor, could cause NullPointerExceptions at runtime. Noncompliant Code Example public class MainClass { @Nonnull private String primary; private String secondary; public MainClass(String color) { if (color != null) { secondary = null; } primary = color; // Noncompliant; "primary" is Nonnull but could be set to null here } public MainClass() { // Noncompliant; "primary" Nonnull" but is not initialized } @Nonnull public String indirectMix() { String mix = null; return mix; // Noncompliant; return value is Nonnull, but null is returned.}} } See MITRE CWE-476 - NULL Pointer Dereference CERT, EXP01-J. - Do not use a null in a case where an object is required | BUG | MINOR | 2 |
| Non-primitive fields should not be "volatile" | Marking an array volatile means that the array itself will always be read fresh and never thread cached, but the items in the array will not be. Similarly, marking a mutable object field volatile means the object reference is volatile but the object itself is not, and other threads may not see updates to the object state. This can be salvaged with arrays by using the relevant AtomicArray class, such as AtomicIntegerArray, instead. For mutable objects, the volatile should be removed, and some other method should be used to ensure thread-safety, such as synchronization, or ThreadLocal storage. Noncompliant Code Example private volatile int [] vInts; // Noncompliant private volatile MyObj myObj; // Noncompliant Compliant Solution private AtomicIntegerArray vInts; private MyObj myObj; See CERT, CON50-J. - Do not assume that declaring a reference volatile guarantees safe publication of the members of the referenced object | BUG | MINOR | 2 |
| "switch" statements should not contain non-case labels | Even if it is legal, mixing case and non-case labels in the body of a switch statement is very confusing and can even be the result of a typing error. Noncompliant Code Example switch (day) { case MONDAY: case TUESDAY: WEDNESDAY: // Noncompliant; syntactically correct, but behavior is not what's expected doSomething(); break; ... } switch (day) { case MONDAY: break; case TUESDAY: foo:for(int i = 0 ; i &lt; X ; i++) { // Noncompliant; the code is correct and behaves as expected but is barely readable /\* ... \*/ break foo; // this break statement doesn't relate to the nesting case TUESDAY /\* ... \*/ } break; /\* ... \*/ } Compliant Solution switch (day) { case MONDAY: case TUESDAY: case WEDNESDAY: doSomething(); break; ... } switch (day) { case MONDAY: break; case TUESDAY: compute(args); // put the content of the labelled "for" statement in a dedicated method break; /\* ... \*/ } | CODE\_SMELL | BLOCKER | 3 |
| Switch cases should end with an unconditional "break" statement | When the execution is not explicitly terminated at the end of a switch case, it continues to execute the statements of the following case. While this is sometimes intentional, it often is a mistake which leads to unexpected behavior. Noncompliant Code Example switch (myVariable) { case 1: foo(); break; case 2: // Both 'doSomething()' and 'doSomethingElse()' will be executed. Is it on purpose ? doSomething(); default: doSomethingElse(); break; } Compliant Solution switch (myVariable) { case 1: foo(); break; case 2: doSomething(); break; default: doSomethingElse(); break; } Exceptions This rule is relaxed in the following cases: switch (myVariable) { case 0: // Empty case used to specify the same behavior for a group of cases. case 1: doSomething(); break; case 2: // Use of return statement return; case 3: // Use of throw statement throw new IllegalStateException(); case 4: // Use of continue statement continue; default: // For the last case, use of break statement is optional doSomethingElse(); } See MITRE, CWE-484 - Omitted Break Statement in Switch CERT, MSC17-C. - Finish every set of statements associated with a case label with a break statement CERT, MSC52-J. - Finish every set of statements associated with a case label with a break statement | CODE\_SMELL | BLOCKER | 3 |
| Methods and field names should not be the same or differ only by capitalization | Looking at the set of methods in a class, including superclass methods, and finding two methods or fields that differ only by capitalization is confusing to users of the class. It is similarly confusing to have a method and a field which differ only in capitalization or a method and a field with exactly the same name and visibility. In the case of methods, it may have been a mistake on the part of the original developer, who intended to override a superclass method, but instead added a new method with nearly the same name. Otherwise, this situation simply indicates poor naming. Method names should be action-oriented, and thus contain a verb, which is unlikely in the case where both a method and a member have the same name (with or without capitalization differences). However, renaming a public method could be disruptive to callers. Therefore renaming the member is the recommended action. Noncompliant Code Example public class Car{ public DriveTrain drive; public void tearDown(){...} public void drive() {...} // Noncompliant; duplicates field name } public class MyCar extends Car{ public void teardown(){...} // Noncompliant; not an override. It it really what's intended? public void drivefast(){...} public void driveFast(){...} //Huh? } Compliant Solution public class Car{ private DriveTrain drive; public void tearDown(){...} public void drive() {...} // field visibility reduced } public class MyCar extends Car{ @Override public void tearDown(){...} public void drivefast(){...} public void driveReallyFast(){...} } | CODE\_SMELL | BLOCKER | 7 |
| Child class fields should not shadow parent class fields | Having a variable with the same name in two unrelated classes is fine, but do the same thing within a class hierarchy and you'll get confusion at best, chaos at worst. Noncompliant Code Example public class Fruit { protected Season ripe; protected Color flesh; // ... } public class Raspberry extends Fruit { private boolean ripe; // Noncompliant private static Color FLESH; // Noncompliant } Compliant Solution public class Fruit { protected Season ripe; protected Color flesh; // ... } public class Raspberry extends Fruit { private boolean ripened; private static Color FLESH\_COLOR; } Exceptions This rule ignores same-name fields that are static in both the parent and child classes. This rule ignores private parent class fields, but in all other such cases, the child class field should be renamed. public class Fruit { private Season ripe; // ... } public class Raspberry extends Fruit { private Season ripe; // Compliant as parent field 'ripe' is anyway not visible from Raspberry // ... } | CODE\_SMELL | BLOCKER | 11 |
| "ThreadGroup" should not be used | There is little valid reason to use the methods of the ThreadGroup class. Some are deprecated (allowThreadSuspension(), resume(), stop(), and suspend()), some are obsolete, others aren't thread-safe, and still others are insecure (activeCount(), enumerate()) . For these reasons, any use of ThreadGroup is suspicious and should be avoided. Compliant Solution ThreadFactory threadFactory = Executors.defaultThreadFactory(); ThreadPoolExecutor executorPool = new ThreadPoolExecutor(3, 10, 5, TimeUnit.SECONDS, new ArrayBlockingQueue&lt;Runnable&gt;(2), threadFactory); for (int i = 0; i &lt; 10; i++) { executorPool.execute(new JobThread("Job: " + i)); } System.out.println(executorPool.getActiveCount()); // Compliant executorPool.shutdown(); See CERT, THI01-J. - Do not invoke ThreadGroup methods | CODE\_SMELL | BLOCKER | 1 |
| Methods returns should not be invariant | When a method is designed to return an invariant value, it may be poor design, but it shouldn't adversely affect the outcome of your program. However, when it happens on all paths through the logic, it is surely a bug. This rule raises an issue when a method contains several return statements that all return the same value. Noncompliant Code Example int foo(int a) { int b = 12; if (a == 1) { return b; } return b; // Noncompliant } | CODE\_SMELL | BLOCKER | 1 |
| The Object.finalize() method should not be overriden | The Object.finalize() method is called on an object by the garbage collector when it determines that there are no more references to the object. But there is absolutely no warranty that this method will be called AS SOON AS the last references to the object are removed. It can be few microseconds to few minutes later. So when system resources need to be disposed by an object, it's better to not rely on this asynchronous mechanism to dispose them. Noncompliant Code Example public class MyClass { ... protected void finalize() { releaseSomeResources(); // Noncompliant } ... } See CERT, MET12-J. - Do not use finalizers | CODE\_SMELL | CRITICAL | 3 |
| Constant names should comply with a naming convention | Shared coding conventions allow teams to collaborate efficiently. This rule checks that all constant names match a provided regular expression. Noncompliant Code Example With the default regular expression ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: public class MyClass { public static final int first = 1; } public enum MyEnum { first; } Compliant Solution public class MyClass { public static final int FIRST = 1; } public enum MyEnum { FIRST; } | CODE\_SMELL | CRITICAL | 26 |
| Methods should not be empty | There are several reasons for a method not to have a method body: It is an unintentional omission, and should be fixed to prevent an unexpected behavior in production. It is not yet, or never will be, supported. In this case an UnsupportedOperationException should be thrown. The method is an intentionally-blank override. In this case a nested comment should explain the reason for the blank override. Noncompliant Code Example public void doSomething() { } public void doSomethingElse() { } Compliant Solution @Override public void doSomething() { // Do nothing because of X and Y. } @Override public void doSomethingElse() { throw new UnsupportedOperationException(); } Exceptions Default (no-argument) constructors are ignored when there are other constructors in the class, as are empty methods in abstract classes. public abstract class Animal { void speak() { // default implementation ignored } } | CODE\_SMELL | CRITICAL | 79 |
| String literals should not be duplicated | Duplicated string literals make the process of refactoring error-prone, since you must be sure to update all occurrences. On the other hand, constants can be referenced from many places, but only need to be updated in a single place. Noncompliant Code Example With the default threshold of 3: public void run() { prepare("action1"); // Noncompliant - "action1" is duplicated 3 times execute("action1"); release("action1"); } @SuppressWarning("all") // Compliant - annotations are excluded private void method1() { /\* ... \*/ } @SuppressWarning("all") private void method2() { /\* ... \*/ } public String method3(String a) { System.out.println("'" + a + "'"); // Compliant - literal "'" has less than 5 characters and is excluded return ""; // Compliant - literal "" has less than 5 characters and is excluded } Compliant Solution private static final String ACTION\_1 = "action1"; // Compliant public void run() { prepare(ACTION\_1); // Compliant execute(ACTION\_1); release(ACTION\_1); } Exceptions To prevent generating some false-positives, literals having less than 5 characters are excluded. | CODE\_SMELL | CRITICAL | 186 |
| Constants should not be defined in interfaces | According to Joshua Bloch, author of "Effective Java": The constant interface pattern is a poor use of interfaces. That a class uses some constants internally is an implementation detail. Implementing a constant interface causes this implementation detail to leak into the class's exported API. It is of no consequence to the users of a class that the class implements a constant interface. In fact, it may even confuse them. Worse, it represents a commitment: if in a future release the class is modified so that it no longer needs to use the constants, it still must implement the interface to ensure binary compatibility. If a nonfinal class implements a constant interface, all of its subclasses will have their namespaces polluted by the constants in the interface. Noncompliant Code Example interface Status { // Noncompliant int OPEN = 1; int CLOSED = 2; } Compliant Solution public enum Status { // Compliant OPEN, CLOSED; } or public final class Status { // Compliant public static final int OPEN = 1; public static final int CLOSED = 2; } | CODE\_SMELL | CRITICAL | 5 |
| Execution of the Garbage Collector should be triggered only by the JVM | Calling System.gc() or Runtime.getRuntime().gc() is a bad idea for a simple reason: there is no way to know exactly what will be done under the hood by the JVM because the behavior will depend on its vendor, version and options: Will the whole application be frozen during the call? Is the -XX:DisableExplicitGC option activated? Will the JVM simply ignore the call? ... An application relying on these unpredictable methods is also unpredictable and therefore broken. The task of running the garbage collector should be left exclusively to the JVM. | CODE\_SMELL | CRITICAL | 1 |
| Fields in a "Serializable" class should either be transient or serializable | Fields in a Serializable class must themselves be either Serializable or transient even if the class is never explicitly serialized or deserialized. For instance, under load, most J2EE application frameworks flush objects to disk, and an allegedly Serializable object with non-transient, non-serializable data members could cause program crashes, and open the door to attackers. In general a Serializable class is expected to fulfil its contract and not have an unexpected behaviour when an instance is serialized. This rule raises an issue on non-Serializable fields, and on collection fields when they are not private (because they could be assigned non-Serializable values externally), and when they are assigned non-Serializable types within the class. Noncompliant Code Example public class Address { //... } public class Person implements Serializable { private static final long serialVersionUID = 1905122041950251207L; private String name; private Address address; // Noncompliant; Address isn't serializable } Compliant Solution public class Address implements Serializable { private static final long serialVersionUID = 2405172041950251807L; } public class Person implements Serializable { private static final long serialVersionUID = 1905122041950251207L; private String name; private Address address; } Exceptions The alternative to making all members serializable or transient is to implement special methods which take on the responsibility of properly serializing and de-serializing the object. This rule ignores classes which implement the following methods: private void writeObject(java.io.ObjectOutputStream out) throws IOException private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException; See MITRE, CWE-594 - Saving Unserializable Objects to Disk Oracle Java 6, Serializable Oracle Java 7, Serializable | CODE\_SMELL | CRITICAL | 69 |
| "for" loop increment clauses should modify the loops' counters | It can be extremely confusing when a for loop's counter is incremented outside of its increment clause. In such cases, the increment should be moved to the loop's increment clause if at all possible. Noncompliant Code Example for (i = 0; i &lt; 10; j++) { // Noncompliant // ... i++; } Compliant Solution for (i = 0; i &lt; 10; i++, j++) { // ... } Or for (i = 0; i &lt; 10; i++) { // ... j++; } | CODE\_SMELL | CRITICAL | 1 |
| Try-with-resources should be used | Java 7 introduced the try-with-resources statement, which guarantees that the resource in question will be closed. Since the new syntax is closer to bullet-proof, it should be preferred over the older try/catch/finally version. This rule checks that close-able resources are opened in a try-with-resources statement. Note that this rule is automatically disabled when the project's sonar.java.source is lower than 7. Noncompliant Code Example FileReader fr = null; BufferedReader br = null; try { fr = new FileReader(fileName); br = new BufferedReader(fr); return br.readLine(); } catch (...) { } finally { if (br != null) { try { br.close(); } catch(IOException e){...} } if (fr != null ) { try { br.close(); } catch(IOException e){...} } } Compliant Solution try ( FileReader fr = new FileReader(fileName); BufferedReader br = new BufferedReader(fr) ) { return br.readLine(); } catch (...) {} or try (BufferedReader br = new BufferedReader(new FileReader(fileName))) { // no need to name intermediate resources if you don't want to return br.readLine(); } catch (...) {} See CERT, ERR54-J. - Use a try-with-resources statement to safely handle closeable resources | CODE\_SMELL | CRITICAL | 28 |
| "Object.wait(...)" and "Condition.await(...)" should be called inside a "while" loop | According to the documentation of the Java Condition interface: When waiting upon a Condition, a "spurious wakeup" is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for. An implementation is free to remove the possibility of spurious wakeups but it is recommended that applications programmers always assume that they can occur and so always wait in a loop. The same advice is also found for the Object.wait(...) method: waits should always occur in loops, like this one: synchronized (obj) { while (&lt;condition does not hold&gt;){ obj.wait(timeout); } ... // Perform action appropriate to condition } Noncompliant Code Example synchronized (obj) { if (!suitableCondition()){ obj.wait(timeout); //the thread can wake up even if the condition is still false } ... // Perform action appropriate to condition } Compliant Solution synchronized (obj) { while (!suitableCondition()){ obj.wait(timeout); } ... // Perform action appropriate to condition } See CERT THI03-J. - Always invoke wait() and await() methods inside a loop | CODE\_SMELL | CRITICAL | 2 |
| "indexOf" checks should not be for positive numbers |  | CODE\_SMELL | CRITICAL | 4 |
| Instance methods should not write to "static" fields | Correctly updating a static field from a non-static method is tricky to get right and could easily lead to bugs if there are multiple class instances and/or multiple threads in play. Ideally, static fields are only updated from synchronized static methods. This rule raises an issue each time a static field is updated from a non-static method. Noncompliant Code Example public class MyClass { private static int count = 0; public void doSomething() { //... count++; // Noncompliant } } | CODE\_SMELL | CRITICAL | 29 |
| "static" base class members should not be accessed via derived types | In the interest of code clarity, static members of a base class should never be accessed using a derived type's name. Doing so is confusing and could create the illusion that two different static members exist. Noncompliant Code Example class Parent { public static int counter; } class Child extends Parent { public Child() { Child.counter++; // Noncompliant } } Compliant Solution class Parent { public static int counter; } class Child extends Parent { public Child() { Parent.counter++; } } | CODE\_SMELL | CRITICAL | 8 |
| Cognitive Complexity of methods should not be too high | Cognitive Complexity is a measure of how hard the control flow of a method is to understand. Methods with high Cognitive Complexity will be difficult to maintain. See Cognitive Complexity | CODE\_SMELL | CRITICAL | 284 |
| A conditionally executed single line should be denoted by indentation | In the absence of enclosing curly braces, the line immediately after a conditional is the one that is conditionally executed. By both convention and good practice, such lines are indented. In the absence of both curly braces and indentation the intent of the original programmer is entirely unclear and perhaps not actually what is executed. Additionally, such code is highly likely to be confusing to maintainers. Noncompliant Code Example if (condition) // Noncompliant doTheThing(); doTheOtherThing(); somethingElseEntirely(); foo(); Compliant Solution if (condition) doTheThing(); doTheOtherThing(); somethingElseEntirely(); foo(); | CODE\_SMELL | CRITICAL | 35 |
| "switch" statements should have "default" clauses | The requirement for a final default clause is defensive programming. The clause should either take appropriate action, or contain a suitable comment as to why no action is taken. Noncompliant Code Example switch (param) { //missing default clause case 0: doSomething(); break; case 1: doSomethingElse(); break; } switch (param) { default: // default clause should be the last one error(); break; case 0: doSomething(); break; case 1: doSomethingElse(); break; } Compliant Solution switch (param) { case 0: doSomething(); break; case 1: doSomethingElse(); break; default: error(); break; } Exceptions If the switch parameter is an Enum and if all the constants of this enum are used in the case statements, then no default clause is expected. Example: public enum Day { SUNDAY, MONDAY } ... switch(day) { case SUNDAY: doSomething(); break; case MONDAY: doSomethingElse(); break; } See MITRE, CWE-478 - Missing Default Case in Switch Statement CERT, MSC01-C. - Strive for logical completeness | CODE\_SMELL | CRITICAL | 50 |
| Deprecated code should be removed | This rule is meant to be used as a way to track code which is marked as being deprecated. Deprecated code should eventually be removed. Noncompliant Code Example class Foo { /\*\* \* @deprecated \*/ public void foo() { // Noncompliant } @Deprecated // Noncompliant public void bar() { } public void baz() { // Compliant } } | CODE\_SMELL | INFO | 36 |
| Track uses of "TODO" tags | TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed. Noncompliant Code Example void doSomething() { // TODO } See MITRE, CWE-546 - Suspicious Comment | CODE\_SMELL | INFO | 32 |
| Source files should not have any duplicated blocks | An issue is created on a file as soon as there is at least one block of duplicated code on this file | CODE\_SMELL | MAJOR | 55 |
| Assignments should not be made from within sub-expressions | Assignments within sub-expressions are hard to spot and therefore make the code less readable. Ideally, sub-expressions should not have side-effects. Noncompliant Code Example if ((str = cont.substring(pos1, pos2)).isEmpty()) { // Noncompliant //... Compliant Solution str = cont.substring(pos1, pos2); if (str.isEmpty()) { //... Exceptions Assignments in while statement conditions, and assignments enclosed in relational expressions are ignored. BufferedReader br = new BufferedReader(/\* ... \*/); String line; while ((line = br.readLine()) != null) {...} Chained assignments, including compound assignments, are ignored. int i = j = 0; int k = (j += 1); result = (bresult = new byte[len]); See MITRE, CWE-481 - Assigning instead of Comparing CERT, EXP45-C. - Do not perform assignments in selection statements CERT, EXP51-J. - Do not perform assignments in conditional expressions | CODE\_SMELL | MAJOR | 171 |
| Sections of code should not be commented out | Programmers should not comment out code as it bloats programs and reduces readability. Unused code should be deleted and can be retrieved from source control history if required. | CODE\_SMELL | MAJOR | 4220 |
| "for" loop stop conditions should be invariant | A for loop stop condition should test the loop counter against an invariant value (i.e. one that is true at both the beginning and ending of every loop iteration). Ideally, this means that the stop condition is set to a local variable just before the loop begins. Stop conditions that are not invariant are slightly less efficient, as well as being difficult to understand and maintain, and likely lead to the introduction of errors in the future. This rule tracks three types of non-invariant stop conditions: When the loop counters are updated in the body of the for loop When the stop condition depend upon a method call When the stop condition depends on an object property, since such properties could change during the execution of the loop. Noncompliant Code Example for (int i = 0; i &lt; 10; i++) { ... i = i - 1; // Noncompliant; counter updated in the body of the loop ... } Compliant Solution for (int i = 0; i &lt; 10; i++) {...} | CODE\_SMELL | MAJOR | 14 |
| Local variables should not shadow class fields | Overriding or shadowing a variable declared in an outer scope can strongly impact the readability, and therefore the maintainability, of a piece of code. Further, it could lead maintainers to introduce bugs because they think they're using one variable but are really using another. Noncompliant Code Example class Foo { public int myField; public void doSomething() { int myField = 0; ... } } See CERT, DCL01-C. - Do not reuse variable names in subscopes CERT, DCL51-J. - Do not shadow or obscure identifiers in subscopes | CODE\_SMELL | MAJOR | 113 |
| Labels should not be used | Labels are not commonly used in Java, and many developers do not understand how they work. Moreover, their usage makes the control flow harder to follow, which reduces the code's readability. Noncompliant Code Example int matrix[][] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} }; outer: for (int row = 0; row &lt; matrix.length; row++) { // Non-Compliant for (int col = 0; col &lt; matrix[row].length; col++) { if (col == row) { continue outer; } System.out.println(matrix[row][col]); // Prints the elements under the diagonal, i.e. 4, 7 and 8 } } Compliant Solution for (int row = 1; row &lt; matrix.length; row++) { // Compliant for (int col = 0; col &lt; row; col++) { System.out.println(matrix[row][col]); // Also prints 4, 7 and 8 } } | CODE\_SMELL | MAJOR | 44 |
| Inheritance tree of classes should not be too deep | Inheritance is certainly one of the most valuable concepts in object-oriented programming. It's a way to compartmentalize and reuse code by creating collections of attributes and behaviors called classes which can be based on previously created classes. But abusing this concept by creating a deep inheritance tree can lead to very complex and unmaintainable source code. Most of the time a too deep inheritance tree is due to bad object oriented design which has led to systematically use 'inheritance' when for instance 'composition' would suit better. This rule raises an issue when the inheritance tree, starting from Object has a greater depth than is allowed. | CODE\_SMELL | MAJOR | 61 |
| Deprecated elements should have both the annotation and the Javadoc tag | Deprecation should be marked with both the @Deprecated annotation and @deprecated Javadoc tag. The annotation enables tools such as IDEs to warn about referencing deprecated elements, and the tag can be used to explain when it was deprecated, why, and how references should be refactored. Further, Java 9 adds two additional arguments to the annotation: since allows you to describe when the deprecation took place forRemoval, indicates whether the deprecated element will be removed at some future date If your compile level is Java 9 or higher, you should be using one or both of these arguments. Noncompliant Code Example class MyClass { @Deprecated public void foo1() { } /\*\* \* @deprecated \*/ public void foo2() { // Noncompliant } } Compliant Solution class MyClass { /\*\* \* @deprecated (when, why, refactoring advice...) \*/ @Deprecated public void foo1() { } /\*\* \* Java &gt;= 9 \* @deprecated (when, why, refactoring advice...) \*/ @Deprecated(since="5.1") public void foo2() { } /\*\* \* Java &gt;= 9 \* @deprecated (when, why, refactoring advice...) \*/ @Deprecated(since="4.2", forRemoval=true) public void foo3() { } } Exceptions The members and methods of a deprecated class or interface are ignored by this rule. The classes and interfaces themselves are still subject to it. /\*\* \* @deprecated (when, why, etc...) \*/ @Deprecated class Qix { public void foo() {} // Compliant; class is deprecated } /\*\* \* @deprecated (when, why, etc...) \*/ @Deprecated interface Plop { void bar(); } | CODE\_SMELL | MAJOR | 26 |
| Methods should not have too many parameters | A long parameter list can indicate that a new structure should be created to wrap the numerous parameters or that the function is doing too many things. Noncompliant Code Example With a maximum number of 4 parameters: public void doSomething(int param1, int param2, int param3, String param4, long param5) { ... } Compliant Solution public void doSomething(int param1, int param2, int param3, String param4) { ... } Exceptions Methods annotated with Spring's @RequestMapping (and related shortcut annotations, like @GetRequest) or @JsonCreator may have a lot of parameters, encapsulation being possible. Such methods are therefore ignored. | CODE\_SMELL | MAJOR | 11 |
| Nested blocks of code should not be left empty | Most of the time a block of code is empty when a piece of code is really missing. So such empty block must be either filled or removed. Noncompliant Code Example for (int i = 0; i &lt; 42; i++){} // Empty on purpose or missing piece of code ? Exceptions When a block contains a comment, this block is not considered to be empty unless it is a synchronized block. synchronized blocks are still considered empty even with comments because they can still affect program flow. | CODE\_SMELL | MAJOR | 43 |
| Generic exceptions should never be thrown | Using such generic exceptions as Error, RuntimeException, Throwable, and Exception prevents calling methods from handling true, system-generated exceptions differently than application-generated errors. Noncompliant Code Example public void foo(String bar) throws Throwable { // Noncompliant throw new RuntimeException("My Message"); // Noncompliant } Compliant Solution public void foo(String bar) { throw new MyOwnRuntimeException("My Message"); } Exceptions Generic exceptions in the signatures of overriding methods are ignored, because overriding method has to follow signature of the throw declaration in the superclass. The issue will be raised on superclass declaration of the method (or won't be raised at all if superclass is not part of the analysis). @Override public void myMethod() throws Exception {...} Generic exceptions are also ignored in the signatures of methods that make calls to methods that throw generic exceptions. public void myOtherMethod throws Exception { doTheThing(); // this method throws Exception } See MITRE, CWE-397 - Declaration of Throws for Generic Exception CERT, ERR07-J. - Do not throw RuntimeException, Exception, or Throwable | CODE\_SMELL | MAJOR | 43 |
| Standard outputs should not be used directly to log anything | When logging a message there are several important requirements which must be fulfilled: The user must be able to easily retrieve the logs The format of all logged message must be uniform to allow the user to easily read the log Logged data must actually be recorded Sensitive data must only be logged securely If a program directly writes to the standard outputs, there is absolutely no way to comply with those requirements. That's why defining and using a dedicated logger is highly recommended. Noncompliant Code Example System.out.println("My Message"); // Noncompliant Compliant Solution logger.log("My Message"); See CERT, ERR02-J. - Prevent exceptions while logging data | CODE\_SMELL | MAJOR | 93 |
| Unused labels should be removed | If a label is declared but not used in the program, it can be considered as dead code and should therefore be removed. This will improve maintainability as developers will not wonder what this label is used for. Noncompliant Code Example void foo() { outer: //label is not used. for(int i = 0; i&lt;10; i++) { break; } } Compliant Solution void foo() { for(int i = 0; i&lt;10; i++) { break; } } See CERT, MSC12-C. - Detect and remove code that has no effect or is never executed | CODE\_SMELL | MAJOR | 3 |
| Collapsible "if" statements should be merged | Merging collapsible if statements increases the code's readability. Noncompliant Code Example if (file != null) { if (file.isFile() || file.isDirectory()) { /\* ... \*/ } } Compliant Solution if (file != null &amp;&amp; isFileOrDirectory(file)) { /\* ... \*/ } private static boolean isFileOrDirectory(File file) { return file.isFile() || file.isDirectory(); } | CODE\_SMELL | MAJOR | 52 |
| Unused "private" fields should be removed | If a private field is declared but not used in the program, it can be considered dead code and should therefore be removed. This will improve maintainability because developers will not wonder what the variable is used for. Note that this rule does not take reflection into account, which means that issues will be raised on private fields that are only accessed using the reflection API. Noncompliant Code Example public class MyClass { private int foo = 42; public int compute(int a) { return a \* 42; } } Compliant Solution public class MyClass { public int compute(int a) { return a \* 42; } } Exceptions The Java serialization runtime associates with each serializable class a version number, called serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. A serializable class can declare its own serialVersionUID explicitly by declaring a field named serialVersionUID that must be static, final, and of type long. By definition those serialVersionUID fields should not be reported by this rule: public class MyClass implements java.io.Serializable { private static final long serialVersionUID = 42L; } Moreover, this rule doesn't raise any issue on annotated fields. | CODE\_SMELL | MAJOR | 10 |
| Utility classes should not have public constructors | Utility classes, which are collections of static members, are not meant to be instantiated. Even abstract utility classes, which can be extended, should not have public constructors. Java adds an implicit public constructor to every class which does not define at least one explicitly. Hence, at least one non-public constructor should be defined. Noncompliant Code Example class StringUtils { // Noncompliant public static String concatenate(String s1, String s2) { return s1 + s2; } } Compliant Solution class StringUtils { // Compliant private StringUtils() { throw new IllegalStateException("Utility class"); } public static String concatenate(String s1, String s2) { return s1 + s2; } } Exceptions When class contains public static void main(String[] args) method it is not considered as utility class and will be ignored by this rule. | CODE\_SMELL | MAJOR | 27 |
| Track uses of "FIXME" tags | FIXME tags are commonly used to mark places where a bug is suspected, but which the developer wants to deal with later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed. Noncompliant Code Example int divide(int numerator, int denominator) { return numerator / denominator; // FIXME denominator value might be 0 } See MITRE, CWE-546 - Suspicious Comment | CODE\_SMELL | MAJOR | 3 |
| Try-catch blocks should not be nested | Nesting try/catch blocks severely impacts the readability of source code because it makes it too difficult to understand which block will catch which exception. | CODE\_SMELL | MAJOR | 14 |
| Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used | Early classes of the Java API, such as Vector, Hashtable and StringBuffer, were synchronized to make them thread-safe. Unfortunately, synchronization has a big negative impact on performance, even when using these collections from a single thread. It is better to use their new unsynchronized replacements: ArrayList or LinkedList instead of Vector Deque instead of Stack HashMap instead of Hashtable StringBuilder instead of StringBuffer Noncompliant Code Example Vector cats = new Vector(); Compliant Solution ArrayList cats = new ArrayList(); Exceptions Use of those synchronized classes is ignored in the signatures of overriding methods. @Override public Vector getCats() {...} | CODE\_SMELL | MAJOR | 140 |
| Enumeration should not be implemented | From the official Oracle Javadoc: NOTE: The functionality of this Enumeration interface is duplicated by the Iterator interface. In addition, Iterator adds an optional remove operation, and has shorter method names. New implementations should consider using Iterator in preference to Enumeration. Noncompliant Code Example public class MyClass implements Enumeration { // Non-Compliant /\* ... \*/ } Compliant Solution public class MyClass implements Iterator { // Compliant /\* ... \*/ } | CODE\_SMELL | MAJOR | 1 |
| "@Override" should be used on overriding and implementing methods | Using the @Override annotation is useful for two reasons : It elicits a warning from the compiler if the annotated method doesn't actually override anything, as in the case of a misspelling. It improves the readability of the source code by making it obvious that methods are overridden. Noncompliant Code Example class ParentClass { public boolean doSomething(){...} } class FirstChildClass extends ParentClass { public boolean doSomething(){...} // Noncompliant } Compliant Solution class ParentClass { public boolean doSomething(){...} } class FirstChildClass extends ParentClass { @Override public boolean doSomething(){...} // Compliant } Exceptions This rule is relaxed when overriding a method from the Object class like toString(), hashCode(), ... | CODE\_SMELL | MAJOR | 137 |
| Empty arrays and collections should be returned instead of null | Returning null instead of an actual array or collection forces callers of the method to explicitly test for nullity, making them more complex and less readable. Moreover, in many cases, null is used as a synonym for empty. Noncompliant Code Example public static List&lt;Result&gt; getResults() { return null; // Noncompliant } public static Result[] getResults() { return null; // Noncompliant } public static void main(String[] args) { Result[] results = getResults(); if (results != null) { // Nullity test required to prevent NPE for (Result result: results) { /\* ... \*/ } } } Compliant Solution public static List&lt;Result&gt; getResults() { return Collections.emptyList(); // Compliant } public static Result[] getResults() { return new Result[0]; } public static void main(String[] args) { for (Result result: getResults()) { /\* ... \*/ } } See CERT, MSC19-C. - For functions that return an array, prefer returning an empty array over a null value CERT, MET55-J. - Return an empty array or collection instead of a null value for methods that return an array or collection | CODE\_SMELL | MAJOR | 22 |
| Only static class initializers should be used |  | CODE\_SMELL | MAJOR | 1 |
| Unused method parameters should be removed | Unused parameters are misleading. Whatever the values passed to such parameters, the behavior will be the same. Noncompliant Code Example void doSomething(int a, int b) { // "b" is unused compute(a); } Compliant Solution void doSomething(int a) { compute(a); } Exceptions The rule will not raise issues for unused parameters: that are annotated with @javax.enterprise.event.Observes in overrides and implementation methods in interface default methods in non-private methods that only throw or that have empty bodies in annotated methods, unless the annotation is @SuppressWarning("unchecked") or @SuppressWarning("rawtypes"), in which case the annotation will be ignored in overridable methods (non-final, or not member of a final class, non-static, non-private), if the parameter is documented with a proper javadoc. @Override void doSomething(int a, int b) { // no issue reported on b compute(a); } public void foo(String s) { // designed to be extended but noop in standard case } protected void bar(String s) { //open-closed principle } public void qix(String s) { throw new UnsupportedOperationException("This method should be implemented in subclasses"); } /\*\* \* @param s This string may be use for further computation in overriding classes \*/ protected void foobar(int a, String s) { // no issue, method is overridable and unused parameter has proper javadoc compute(a); } See CERT, MSC12-C. - Detect and remove code that has no effect or is never executed | CODE\_SMELL | MAJOR | 23 |
| Throwable and Error should not be caught | Throwable is the superclass of all errors and exceptions in Java. Error is the superclass of all errors, which are not meant to be caught by applications. Catching either Throwable or Error will also catch OutOfMemoryError and InternalError, from which an application should not attempt to recover. Noncompliant Code Example try { /\* ... \*/ } catch (Throwable t) { /\* ... \*/ } try { /\* ... \*/ } catch (Error e) { /\* ... \*/ } Compliant Solution try { /\* ... \*/ } catch (RuntimeException e) { /\* ... \*/ } try { /\* ... \*/ } catch (MyException e) { /\* ... \*/ } See MITRE, CWE-396 - Declaration of Catch for Generic Exception CERT, ERR08-J. - Do not catch NullPointerException or any of its ancestors | CODE\_SMELL | MAJOR | 69 |
| Exception types should not be tested using "instanceof" in catch blocks | Multiple catch blocks of the appropriate type should be used instead of catching a general exception, and then testing on the type. Noncompliant Code Example try { /\* ... \*/ } catch (Exception e) { if(e instanceof IOException) { /\* ... \*/ } // Noncompliant if(e instanceof NullPointerException{ /\* ... \*/ } // Noncompliant } Compliant Solution try { /\* ... \*/ } catch (IOException e) { /\* ... \*/ } // Compliant } catch (NullPointerException e) { /\* ... \*/ } // Compliant See CERT, ERR51-J. - Prefer user-defined exceptions over more general exception types | CODE\_SMELL | MAJOR | 72 |
| "switch" statements should not have too many "case" clauses | When switch statements have large sets of case clauses, it is usually an attempt to map two sets of data. A real map structure would be more readable and maintainable, and should be used instead. Exceptions This rule ignores switches over Enums and empty, fall-through cases. | CODE\_SMELL | MAJOR | 2 |
| Anonymous inner classes containing only one method should become lambdas |  | CODE\_SMELL | MAJOR | 151 |
| A field should not duplicate the name of its containing class | It's confusing to have a class member with the same name (case differences aside) as its enclosing class. This is particularly so when you consider the common practice of naming a class instance for the class itself. Best practice dictates that any field or member with the same name as the enclosing class be renamed to be more descriptive of the particular aspect of the class it represents or holds. Noncompliant Code Example public class Foo { private String foo; public String getFoo() { } } Foo foo = new Foo(); foo.getFoo() // what does this return? Compliant Solution public class Foo { private String name; public String getName() { } } //... Foo foo = new Foo(); foo.getName() Exceptions When the type of the field is the containing class and that field is static, no issue is raised to allow singletons named like the type. public class Foo { ... private static Foo foo; public Foo getInstance() { if(foo==null) { foo = new Foo(); } return foo; } ... } | CODE\_SMELL | MAJOR | 9 |
| Dead stores should be removed | A dead store happens when a local variable is assigned a value that is not read by any subsequent instruction. Calculating or retrieving a value only to then overwrite it or throw it away, could indicate a serious error in the code. Even if it's not an error, it is at best a waste of resources. Therefore all calculated values should be used. Noncompliant Code Example i = a + b; // Noncompliant; calculation result not used before value is overwritten i = compute(); Compliant Solution i = a + b; i += compute(); Exceptions This rule ignores initializations to -1, 0, 1, null, true, false and "". See MITRE, CWE-563 - Assignment to Variable without Use ('Unused Variable') CERT, MSC13-C. - Detect and remove unused values CERT, MSC56-J. - Detect and remove superfluous code and values | CODE\_SMELL | MAJOR | 42 |
| Two branches in a conditional structure should not have exactly the same implementation |  | CODE\_SMELL | MAJOR | 5 |
| "URL.hashCode" and "URL.equals" should be avoided |  | CODE\_SMELL | MAJOR | 1 |
| Constructors should not be used to instantiate "String", "BigInteger", "BigDecimal" and primitive-wrapper classes | Constructors for String, BigInteger, BigDecimal and the objects used to wrap primitives should never be used. Doing so is less clear and uses more memory than simply using the desired value in the case of strings, and using valueOf for everything else. Noncompliant Code Example String empty = new String(); // Noncompliant; yields essentially "", so just use that. String nonempty = new String("Hello world"); // Noncompliant Double myDouble = new Double(1.1); // Noncompliant; use valueOf Integer integer = new Integer(1); // Noncompliant Boolean bool = new Boolean(true); // Noncompliant BigInteger bigInteger1 = new BigInteger("3"); // Noncompliant BigInteger bigInteger2 = new BigInteger("9223372036854775807"); // Noncompliant BigInteger bigInteger3 = new BigInteger("111222333444555666777888999"); // Compliant, greater than Long.MAX\_VALUE Compliant Solution String empty = ""; String nonempty = "Hello world"; Double myDouble = Double.valueOf(1.1); Integer integer = Integer.valueOf(1); Boolean bool = Boolean.valueOf(true); BigInteger bigInteger1 = BigInteger.valueOf(3); BigInteger bigInteger2 = BigInteger.valueOf(9223372036854775807L); BigInteger bigInteger3 = new BigInteger("111222333444555666777888999"); Exceptions BigDecimal constructor with double argument is ignored as using valueOf instead might change resulting value. See S2111. | CODE\_SMELL | MAJOR | 129 |
| "static" members should be accessed statically | While it is possible to access static members from a class instance, it's bad form, and considered by most to be misleading because it implies to the readers of your code that there's an instance of the member per class instance. Noncompliant Code Example public class A { public static int counter = 0; } public class B { private A first = new A(); private A second = new A(); public void runUpTheCount() { first.counter ++; // Noncompliant second.counter ++; // Noncompliant. A.counter is now 2, which is perhaps contrary to expectations } } Compliant Solution public class A { public static int counter = 0; } public class B { private A first = new A(); private A second = new A(); public void runUpTheCount() { A.counter ++; // Compliant A.counter ++; // Compliant } } | CODE\_SMELL | MAJOR | 4 |
| Parameters should be passed in the correct order | When the names of parameters in a method call match the names of the method arguments, it contributes to clearer, more readable code. However, when the names match, but are passed in a different order than the method arguments, it indicates a mistake in the parameter order which will likely lead to unexpected results. Noncompliant Code Example public double divide(int divisor, int dividend) { return divisor/dividend; } public void doTheThing() { int divisor = 15; int dividend = 5; double result = divide(dividend, divisor); // Noncompliant; operation succeeds, but result is unexpected //... } Compliant Solution public double divide(int divisor, int dividend) { return divisor/dividend; } public void doTheThing() { int divisor = 15; int dividend = 5; double result = divide(divisor, dividend); //... } | CODE\_SMELL | MAJOR | 136 |
| Boolean expressions should not be gratuitous | If a boolean expression doesn't change the evaluation of the condition, then it is entirely unnecessary, and can be removed. If it is gratuitous because it does not match the programmer's intent, then it's a bug and the expression should be fixed. Noncompliant Code Example a = true; if (a) { // Noncompliant doSomething(); } if (b &amp;&amp; a) { // Noncompliant; "a" is always "true" doSomething(); } if (c || !a) { // Noncompliant; "!a" is always "false" doSomething(); } Compliant Solution a = true; if (foo(a)) { doSomething(); } if (b) { doSomething(); } if (c) { doSomething(); } See MISRA C:2004, 13.7 - Boolean operations whose results are invariant shall not be permitted. MISRA C:2012, 14.3 - Controlling expressions shall not be invariant MITRE, CWE-571 - Expression is Always True MITRE, CWE-570 - Expression is Always False MITRE, CWE-489 - Leftover Debug Code CERT, MSC12-C. - Detect and remove code that has no effect or is never executed | CODE\_SMELL | MAJOR | 9 |
| Multiline blocks should be enclosed in curly braces | Curly braces can be omitted from a one-line block, such as with an if statement or for loop, but doing so can be misleading and induce bugs. This rule raises an issue when the whitespacing of the lines after a one line block indicates an intent to include those lines in the block, but the omission of curly braces means the lines will be unconditionally executed once. Noncompliant Code Example if (condition) firstActionInBlock(); secondAction(); // Noncompliant; executed unconditionally thirdAction(); if (condition) firstActionInBlock(); secondAction(); // Noncompliant; secondAction executed unconditionally if (condition) firstActionInBlock(); // Noncompliant secondAction(); // Executed unconditionally if (condition); secondAction(); // Noncompliant; secondAction executed unconditionally String str = null; for (int i = 0; i &lt; array.length; i++) str = array[i]; doTheThing(str); // Noncompliant; executed only on last array element Compliant Solution if (condition) { firstActionInBlock(); secondAction(); } thirdAction(); String str = null; for (int i = 0; i &lt; array.length; i++) { str = array[i]; doTheThing(str); } See MITRE, CWE-483 - Incorrect Block Delimitation CERT, EXP52-J. - Use braces for the body of an if, for, or while statement | CODE\_SMELL | MAJOR | 2 |
| "entrySet()" should be iterated when both the key and value are needed |  | CODE\_SMELL | MAJOR | 2 |
| Static fields should not be updated in constructors | Assigning a value to a static field in a constructor could cause unreliable behavior at runtime since it will change the value for all instances of the class. Instead remove the field's static modifier, or initialize it statically. Noncompliant Code Example public class Person { static Date dateOfBirth; static int expectedFingers; public Person(date birthday) { dateOfBirth = birthday; // Noncompliant; now everyone has this birthday expectedFingers = 10; // Noncompliant } } Compliant Solution public class Person { Date dateOfBirth; static int expectedFingers = 10; public Person(date birthday) { dateOfBirth = birthday; } } | CODE\_SMELL | MAJOR | 5 |
| String function use should be optimized for single characters | An indexOf or lastIndexOf call with a single letter String can be made more performant by switching to a call with a char argument. Noncompliant Code Example String myStr = "Hello World"; // ... int pos = myStr.indexOf("W"); // Noncompliant // ... int otherPos = myStr.lastIndexOf("r"); // Noncompliant // ... Compliant Solution String myStr = "Hello World"; // ... int pos = myStr.indexOf('W'); // ... int otherPos = myStr.lastIndexOf('r'); // ... | CODE\_SMELL | MAJOR | 8 |
| "writeObject" should not be the only "synchronized" code in a class | The purpose of synchronization is to ensure that only one thread executes a given block of code at a time. There's no real problem with marking writeObject synchronized, but it's highly suspicious if this serialization-related method is the only synchronized code in a class. Noncompliant Code Example public class RubberBall { private Color color; private int diameter; public RubberBall(Color color, int diameter) { // ... } public void bounce(float angle, float velocity) { // ... } private synchronized void writeObject(ObjectOutputStream stream) throws IOException { // Noncompliant // ... } } Compliant Solution public class RubberBall { private Color color; private int diameter; public RubberBall(Color color, int diameter) { // ... } public void bounce(float angle, float velocity) { // ... } private void writeObject(ObjectOutputStream stream) throws IOException { // ... } } | CODE\_SMELL | MAJOR | 1 |
| Ternary operators should not be nested | Just because you can do something, doesn't mean you should, and that's the case with nested ternary operations. Nesting ternary operators results in the kind of code that may seem clear as day when you write it, but six months later will leave maintainers (or worse - future you) scratching their heads and cursing. Instead, err on the side of clarity, and use another line to express the nested operation as a separate statement. Noncompliant Code Example public String getTitle(Person p) { return p.gender == Person.MALE ? "Mr. " : p.isMarried() ? "Mrs. " : "Miss "; // Noncompliant } Compliant Solution public String getTitle(Person p) { if (p.gender == Person.MALE) { return "Mr. "; } return p.isMarried() ? "Mrs. " : "Miss "; } | CODE\_SMELL | MAJOR | 12 |
| "Map.get" and value test should be replaced with single method call | It's a common pattern to test the result of a java.util.Map.get() against null before proceeding with adding or changing the value in the map. However the java.util.Map API offers a significantly better alternative in the form of the computeIfPresent() and computeIfAbsent() methods. Using these instead leads to cleaner and more readable code. Note that this rule is automatically disabled when the project's sonar.java.source is not 8. Noncompliant Code Example V value = map.get(key); if (value == null) { // Noncompliant value = V.createFor(key); if (value != null) { map.put(key, value); } } return value; Compliant Solution return map.computeIfAbsent(key, k -&gt; V.createFor(k)); | CODE\_SMELL | MAJOR | 8 |
| "java.nio.Files#delete" should be preferred | When java.io.File#delete fails, this boolean method simply returns false with no indication of the cause. On the other hand, when java.nio.Files#delete fails, this void method returns one of a series of exception types to better indicate the cause of the failure. And since more information is generally better in a debugging situation, java.nio.Files#delete is the preferred option. Noncompliant Code Example public void cleanUp(Path path) { File file = new File(path); if (!file.delete()) { // Noncompliant //... } } Compliant Solution public void cleanUp(Path path) throws NoSuchFileException, DirectoryNotEmptyException, IOException{ Files.delete(path); } | CODE\_SMELL | MAJOR | 25 |
| Methods should not have identical implementations | When two methods have the same implementation, either it was a mistake - something else was intended - or the duplication was intentional, but may be confusing to maintainers. In the latter case, one implementation should invoke the other. Numerical and string literals are not taken into account. Noncompliant Code Example private final static String CODE = "bounteous"; public String calculateCode() { doTheThing(); return CODE; } public String getName() { // Noncompliant doTheThing(); return CODE; } Compliant Solution private final static String CODE = "bounteous"; public String getCode() { doTheThing(); return CODE; } public String getName() { return getCode(); } Exceptions Methods that are not accessors (getters and setters), with fewer than 2 statements are ignored. | CODE\_SMELL | MAJOR | 21 |
| Assignments should not be redundant | The transitive property says that if a == b and b == c, then a == c. In such cases, there's no point in assigning a to c or vice versa because they're already equivalent. This rule raises an issue when an assignment is useless because the assigned-to variable already holds the value on all execution paths. Noncompliant Code Example a = b; c = a; b = c; // Noncompliant: c and b are already the same Compliant Solution a = b; c = a; | CODE\_SMELL | MAJOR | 1 |
| Asserts should not be used to check the parameters of a public method |  | CODE\_SMELL | MAJOR | 4 |
| "else" statements should be clearly matched with an "if" | The dangling else problem appears when nested if/else&nbsp;statements are written without curly braces. In this case, else is associated with the nearest&nbsp;if but that is not always obvious and sometimes the indentation can also be misleading. This rules reports else statements that are difficult to understand, because they are inside nested if statements without curly braces. Adding curly braces can generally make the code clearer (RSPEC-121), and in this situation of dangling else, it really clarifies the intention of the code. Noncompliant Code Example if (a) if (b) d++; else // Noncompliant, is the "else" associated with "if(a)" or "if (b)"? (the answer is "if(b)") e++; Compliant Solution if (a) { if (b) { d++; } } else { // Compliant, there is no doubt the "else" is associated with "if(a)" e++; } See \* https://en.wikipedia.org/wiki/Dangling\_else | CODE\_SMELL | MAJOR | 23 |
| 'List.remove()' should not be used in ascending 'for' loops |  | CODE\_SMELL | MAJOR | 2 |
| Unused "private" methods should be removed | private methods that are never executed are dead code: unnecessary, inoperative code that should be removed. Cleaning out dead code decreases the size of the maintained codebase, making it easier to understand the program and preventing bugs from being introduced. Note that this rule does not take reflection into account, which means that issues will be raised on private methods that are only accessed using the reflection API. Noncompliant Code Example public class Foo implements Serializable { private Foo(){} //Compliant, private empty constructor intentionally used to prevent any direct instantiation of a class. public static void doSomething(){ Foo foo = new Foo(); ... } private void unusedPrivateMethod(){...} private void writeObject(ObjectOutputStream s){...} //Compliant, relates to the java serialization mechanism private void readObject(ObjectInputStream in){...} //Compliant, relates to the java serialization mechanism } Compliant Solution public class Foo implements Serializable { private Foo(){} //Compliant, private empty constructor intentionally used to prevent any direct instantiation of a class. public static void doSomething(){ Foo foo = new Foo(); ... } private void writeObject(ObjectOutputStream s){...} //Compliant, relates to the java serialization mechanism private void readObject(ObjectInputStream in){...} //Compliant, relates to the java serialization mechanism } Exceptions This rule doesn't raise any issue on annotated methods. | CODE\_SMELL | MAJOR | 6 |
| Redundant pairs of parentheses should be removed |  | CODE\_SMELL | MAJOR | 6 |
| "@Deprecated" code should not be used | Once deprecated, classes, and interfaces, and their members should be avoided, rather than used, inherited or extended. Deprecation is a warning that the class or interface has been superseded, and will eventually be removed. The deprecation period allows you to make a smooth transition away from the aging, soon-to-be-retired technology. Noncompliant Code Example /\*\* \* @deprecated As of release 1.3, replaced by {@link #Fee} \*/ @Deprecated public class Fum { ... } public class Foo { /\*\* \* @deprecated As of release 1.7, replaced by {@link #doTheThingBetter()} \*/ @Deprecated public void doTheThing() { ... } public void doTheThingBetter() { ... } } public class Bar extends Foo { public void doTheThing() { ... } // Noncompliant; don't override a deprecated method or explicitly mark it as @Deprecated } public class Bar extends Fum { // Noncompliant; Fum is deprecated public void myMethod() { Foo foo = new Foo(); // okay; the class isn't deprecated foo.doTheThing(); // Noncompliant; doTheThing method is deprecated } } See MITRE, CWE-477 - Use of Obsolete Functions CERT, MET02-J. - Do not use deprecated or obsolete classes or methods | CODE\_SMELL | MINOR | 225 |
| Empty statements should be removed | Empty statements, i.e. ;, are usually introduced by mistake, for example because: It was meant to be replaced by an actual statement, but this was forgotten. There was a typo which lead the semicolon to be doubled, i.e. ;;. Noncompliant Code Example void doSomething() { ; // Noncompliant - was used as a kind of TODO marker } void doSomethingElse() { System.out.println("Hello, world!");; // Noncompliant - double ; ... } Compliant Solution void doSomething() {} void doSomethingElse() { System.out.println("Hello, world!"); ... for (int i = 0; i &lt; 3; i++) ; // compliant if unique statement of a loop ... } See CERT, MSC12-C. - Detect and remove code that has no effect or is never executed CERT, MSC51-J. - Do not place a semicolon immediately following an if, for, or while condition CERT, EXP15-C. - Do not place a semicolon on the same line as an if, for, or while statement | CODE\_SMELL | MINOR | 58 |
| Modifiers should be declared in the correct order | The Java Language Specification recommends listing modifiers in the following order: 1. Annotations 2. public 3. protected 4. private 5. abstract 6. static 7. final 8. transient 9. volatile 10. synchronized 11. native 12. strictfp Not following this convention has no technical impact, but will reduce the code's readability because most developers are used to the standard order. Noncompliant Code Example static public void main(String[] args) { // Noncompliant } Compliant Solution public static void main(String[] args) { // Compliant } | CODE\_SMELL | MINOR | 403 |
| "throws" declarations should not be superfluous | An exception in a throws declaration in Java is superfluous if it is: listed multiple times a subclass of another listed exception a RuntimeException, or one of its descendants completely unnecessary because the declared exception type cannot actually be thrown Noncompliant Code Example void foo() throws MyException, MyException {} // Noncompliant; should be listed once void bar() throws Throwable, Exception {} // Noncompliant; Exception is a subclass of Throwable void baz() throws RuntimeException {} // Noncompliant; RuntimeException can always be thrown Compliant Solution void foo() throws MyException {} void bar() throws Throwable {} void baz() {} Exceptions The rule will not raise any issue for exceptions that cannot be thrown from the method body: in overriding and implementation methods in interface default methods in non-private methods that only throw, have empty bodies, or a single return statement . in overridable methods (non-final, or not member of a final class, non-static, non-private), if the exception is documented with a proper javadoc. class A extends B { @Override void doSomething() throws IOException { compute(a); } public void foo() throws IOException {} protected void bar() throws IOException { throw new UnsupportedOperationException("This method should be implemented in subclasses"); } Object foobar(String s) throws IOException { return null; } /\*\* \* @throws IOException Overriding classes may throw this exception if they print values into a file \*/ protected void print() throws IOException { // no issue, method is overridable and the exception has proper javadoc System.out.println("foo"); } } | CODE\_SMELL | MINOR | 22 |
| Method names should comply with a naming convention | Shared naming conventions allow teams to collaborate efficiently. This rule checks that all method names match a provided regular expression. Noncompliant Code Example With default provided regular expression ^[a-z][a-zA-Z0-9]\*$: public int DoSomething(){...} Compliant Solution public int doSomething(){...} Exceptions Overriding methods are excluded. @Override public int Do\_Something(){...} | CODE\_SMELL | MINOR | 348 |
| Class names should comply with a naming convention | Shared coding conventions allow teams to collaborate effectively. This rule allows to check that all class names match a provided regular expression. Noncompliant Code Example With default provided regular expression ^[A-Z][a-zA-Z0-9]\*$: class my\_class {...} Compliant Solution class MyClass {...} | CODE\_SMELL | MINOR | 7 |
| Field names should comply with a naming convention | Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression. Noncompliant Code Example With the default regular expression ^[a-z][a-zA-Z0-9]\*$: class MyClass { private int my\_field; } Compliant Solution class MyClass { private int myField; } | CODE\_SMELL | MINOR | 28 |
| Local variable and method parameter names should comply with a naming convention | Shared naming conventions allow teams to collaborate effectively. This rule raises an issue when a local variable or function parameter name does not match the provided regular expression. Noncompliant Code Example With the default regular expression ^[a-z][a-zA-Z0-9]\*$: public void doSomething(int my\_param) { int LOCAL; ... } Compliant Solution public void doSomething(int myParam) { int local; ... } Exceptions Loop counters are ignored by this rule. for (int i\_1 = 0; i\_1 &lt; limit; i\_1++) { // Compliant // ... } as well as one-character catch variables: try { //... } catch (Exception e) { // Compliant } | CODE\_SMELL | MINOR | 161 |
| URIs should not be hardcoded |  | CODE\_SMELL | MINOR | 5 |
| Boolean literals should not be redundant | Redundant Boolean literals should be removed from expressions to improve readability. Noncompliant Code Example if (booleanMethod() == true) { /\* ... \*/ } if (booleanMethod() == false) { /\* ... \*/ } if (booleanMethod() || false) { /\* ... \*/ } doSomething(!false); doSomething(booleanMethod() == true); booleanVariable = booleanMethod() ? true : false; booleanVariable = booleanMethod() ? true : exp; booleanVariable = booleanMethod() ? false : exp; booleanVariable = booleanMethod() ? exp : true; booleanVariable = booleanMethod() ? exp : false; Compliant Solution if (booleanMethod()) { /\* ... \*/ } if (!booleanMethod()) { /\* ... \*/ } if (booleanMethod()) { /\* ... \*/ } doSomething(true); doSomething(booleanMethod()); booleanVariable = booleanMethod(); booleanVariable = booleanMethod() || exp; booleanVariable = !booleanMethod() &amp;&amp; exp; booleanVariable = !booleanMethod() || exp; booleanVariable = booleanMethod() &amp;&amp; exp; | CODE\_SMELL | MINOR | 5 |
| Return of boolean expressions should not be wrapped into an "if-then-else" statement | Return of boolean literal statements wrapped into if-then-else ones should be simplified. Similarly, method invocations wrapped into if-then-else differing only from boolean literals should be simplified into a single invocation. Noncompliant Code Example boolean foo(Object param) { if (expression) { // Noncompliant bar(param, true, "qix"); } else { bar(param, false, "qix"); } if (expression) { // Noncompliant return true; } else { return false; } } Compliant Solution boolean foo(Object param) { bar(param, expression, "qix"); return expression; } | CODE\_SMELL | MINOR | 176 |
| Collection.isEmpty() should be used to test for emptiness | Using Collection.size() to test for emptiness works, but using Collection.isEmpty() makes the code more readable and can be more performant. The time complexity of any isEmpty() method implementation should be O(1) whereas some implementations of size() can be O(n). Noncompliant Code Example if (myCollection.size() == 0) { // Noncompliant /\* ... \*/ } Compliant Solution if (myCollection.isEmpty()) { /\* ... \*/ } | CODE\_SMELL | MINOR | 5 |
| Exception classes should be immutable | Exceptions are meant to represent the application's state at the point at which an error occurred. Making all fields in an Exception class final ensures that this state: Will be fully defined at the same time the Exception is instantiated. Won't be updated or corrupted by a questionable error handler. This will enable developers to quickly understand what went wrong. Noncompliant Code Example public class MyException extends Exception { private int status; // Noncompliant public MyException(String message) { super(message); } public int getStatus() { return status; } public void setStatus(int status) { this.status = status; } } Compliant Solution public class MyException extends Exception { private final int status; public MyException(String message, int status) { super(message); this.status = status; } public int getStatus() { return status; } } | CODE\_SMELL | MINOR | 4 |
| Overriding methods should do more than simply call the same method in the super class | Overriding a method just to call the same method from the super class without performing any other actions is useless and misleading. The only time this is justified is in final overriding methods, where the effect is to lock in the parent class behavior. This rule ignores such overrides of equals, hashCode and toString. Noncompliant Code Example public void doSomething() { super.doSomething(); } @Override public boolean isLegal(Action action) { return super.isLegal(action); } Compliant Solution @Override public boolean isLegal(Action action) { // Compliant - not simply forwarding the call return super.isLegal(new Action(/\* ... \*/)); } @Id @Override public int getId() { // Compliant - there is annotation different from @Override return super.getId(); } | CODE\_SMELL | MINOR | 10 |
| Array designators "[]" should be on the type, not the variable | Array designators should always be located on the type for better code readability. Otherwise, developers must look both at the type and the variable name to know whether or not a variable is an array. Noncompliant Code Example int matrix[][]; // Noncompliant int[] matrix[]; // Noncompliant Compliant Solution int[][] matrix; // Compliant | CODE\_SMELL | MINOR | 20 |
| Nested code blocks should not be used | Nested code blocks can be used to create a new scope and restrict the visibility of the variables defined inside it. Using this feature in a method typically indicates that the method has too many responsibilities, and should be refactored into smaller methods. Noncompliant Code Example public void evaluate(int operator) { switch (operator) { /\* ... \*/ case ADD: { // Noncompliant - nested code block '{' ... '}' int a = stack.pop(); int b = stack.pop(); int result = a + b; stack.push(result); break; } /\* ... \*/ } } Compliant Solution public void evaluate(int operator) { switch (operator) { /\* ... \*/ case ADD: // Compliant evaluateAdd(); break; /\* ... \*/ } } private void evaluateAdd() { int a = stack.pop(); int b = stack.pop(); int result = a + b; stack.push(result); } | CODE\_SMELL | MINOR | 121 |
| "equals(Object obj)" should be overridden along with the "compareTo(T obj)" method | According to the Java Comparable.compareTo(T o) documentation: It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals." If this rule is violated, weird and unpredictable failures can occur. For example, in Java 5 the PriorityQueue.remove() method relied on compareTo(), but since Java 6 it has relied on equals(). Noncompliant Code Example public class Foo implements Comparable&lt;Foo&gt; { @Override public int compareTo(Foo foo) { /\* ... \*/ } // Noncompliant as the equals(Object obj) method is not overridden } Compliant Solution public class Foo implements Comparable&lt;Foo&gt; { @Override public int compareTo(Foo foo) { /\* ... \*/ } // Compliant @Override public boolean equals(Object obj) { /\* ... \*/ } } | CODE\_SMELL | MINOR | 1 |
| A "while" loop should be used instead of a "for" loop | When only the condition expression is defined in a for loop, and the initialization and increment expressions are missing, a while loop should be used instead to increase readability. Noncompliant Code Example for (;condition;) { /\*...\*/ } Compliant Solution while (condition) { /\*...\*/ } | CODE\_SMELL | MINOR | 1 |
| "switch" statements should have at least 3 "case" clauses | switch statements are useful when there are many different cases depending on the value of the same expression. For just one or two cases however, the code will be more readable with if statements. Noncompliant Code Example switch (variable) { case 0: doSomething(); break; default: doSomethingElse(); break; } Compliant Solution if (variable == 0) { doSomething(); } else { doSomethingElse(); } | CODE\_SMELL | MINOR | 58 |
| Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList" |  | CODE\_SMELL | MINOR | 21 |
| Loops should not contain more than a single "break" or "continue" statement | Restricting the number of break and continue statements in a loop is done in the interest of good structured programming. One break and continue statement is acceptable in a loop, since it facilitates optimal coding. If there is more than one, the code should be refactored to increase readability. Noncompliant Code Example for (int i = 1; i &lt;= 10; i++) { // Noncompliant - 2 continue - one might be tempted to add some logic in between if (i % 2 == 0) { continue; } if (i % 3 == 0) { continue; } System.out.println("i = " + i); } | CODE\_SMELL | MINOR | 35 |
| Private fields only used as local variables in methods should become local variables | When the value of a private field is always assigned to in a class' methods before being read, then it is not being used to store class information. Therefore, it should become a local variable in the relevant methods to prevent any misunderstanding. Noncompliant Code Example public class Foo { private int a; private int b; public void doSomething(int y) { a = y + 5; ... if(a == 0) { ... } ... } public void doSomethingElse(int y) { b = y + 3; ... } } Compliant Solution public class Foo { public void doSomething(int y) { int a = y + 5; ... if(a == 0) { ... } } public void doSomethingElse(int y) { int b = y + 3; ... } } Exceptions This rule doesn't raise any issue on annotated field. | CODE\_SMELL | MINOR | 10 |
| Unused local variables should be removed | If a local variable is declared but not used, it is dead code and should be removed. Doing so will improve maintainability because developers will not wonder what the variable is used for. Noncompliant Code Example public int numberOfMinutes(int hours) { int seconds = 0; // seconds is never used return hours \* 60; } Compliant Solution public int numberOfMinutes(int hours) { return hours \* 60; } | CODE\_SMELL | MINOR | 27 |
| Local variables should not be declared and then immediately returned or thrown | Declaring a variable only to immediately return or throw it is a bad practice. Some developers argue that the practice improves code readability, because it enables them to explicitly name what is being returned. However, this variable is an internal implementation detail that is not exposed to the callers of the method. The method name should be sufficient for callers to know exactly what will be returned. Noncompliant Code Example public long computeDurationInMilliseconds() { long duration = (((hours \* 60) + minutes) \* 60 + seconds ) \* 1000 ; return duration; } public void doSomething() { RuntimeException myException = new RuntimeException(); throw myException; } Compliant Solution public long computeDurationInMilliseconds() { return (((hours \* 60) + minutes) \* 60 + seconds ) \* 1000 ; } public void doSomething() { throw new RuntimeException(); } | CODE\_SMELL | MINOR | 34 |
| Abstract classes without fields should be converted to interfaces | With Java 8's "default method" feature, any abstract class without direct or inherited field should be converted into an interface. However, this change may not be appropriate in libraries or other applications where the class is intended to be used as an API. Note that this rule is automatically disabled when the project's sonar.java.source is lower than 8. Noncompliant Code Example public abstract class Car { public abstract void start(Environment c); public void stop(Environment c) { c.freeze(this); } } Compliant Solution public interface Car { public void start(Environment c); public default void stop(Environment c) { c.freeze(this); } } | CODE\_SMELL | MINOR | 2 |
| Strings should not be concatenated using '+' in a loop | Strings are immutable objects, so concatenation doesn't simply add the new String to the end of the existing string. Instead, in each loop iteration, the first String is converted to an intermediate object type, the second string is appended, and then the intermediate object is converted back to a String. Further, performance of these intermediate operations degrades as the String gets longer. Therefore, the use of StringBuilder is preferred. Noncompliant Code Example String str = ""; for (int i = 0; i &lt; arrayOfStrings.length ; ++i) { str = str + arrayOfStrings[i]; } Compliant Solution StringBuilder bld = new StringBuilder(); for (int i = 0; i &lt; arrayOfStrings.length; ++i) { bld.append(arrayOfStrings[i]); } String str = bld.toString(); | CODE\_SMELL | MINOR | 13 |
| Multiple variables should not be declared on the same line | Declaring multiple variables on one line is difficult to read. Noncompliant Code Example class MyClass { private int a, b; public void method(){ int c; int d; } } Compliant Solution class MyClass { private int a; private int b; public void method(){ int c; int d; } } See CERT, DCL52-J. - Do not declare more than one variable per declaration CERT, DCL04-C. - Do not declare more than one variable per declaration | CODE\_SMELL | MINOR | 141 |
| "toString()" should never be called on a String object | Invoking a method designed to return a string representation of an object which is already a string is a waste of keystrokes. This redundant construction may be optimized by the compiler, but will be confusing in the meantime. Noncompliant Code Example String message = "hello world"; System.out.println(message.toString()); // Noncompliant; Compliant Solution String message = "hello world"; System.out.println(message); | CODE\_SMELL | MINOR | 1 |
| Redundant casts should not be used |  | CODE\_SMELL | MINOR | 27 |
| Boolean checks should not be inverted | It is needlessly complex to invert the result of a boolean comparison. The opposite comparison should be made instead. Noncompliant Code Example if ( !(a == 2)) { ...} // Noncompliant boolean b = !(i &lt; 10); // Noncompliant Compliant Solution if (a != 2) { ...} boolean b = (i &gt;= 10); | CODE\_SMELL | MINOR | 3 |
| Fields in non-serializable classes should not be "transient" | transient is used to mark fields in a Serializable class which will not be written out to file (or stream). In a class that does not implement Serializable, this modifier is simply wasted keystrokes, and should be removed. Noncompliant Code Example class Vegetable { // does not implement Serializable private transient Season ripe; // Noncompliant // ... } Compliant Solution class Vegetable { private Season ripe; // ... } | CODE\_SMELL | MINOR | 6 |
| Classes should not be empty | There is no good excuse for an empty class. If it's being used simply as a common extension point, it should be replaced with an interface. If it was stubbed in as a placeholder for future development it should be fleshed-out. In any other case, it should be eliminated. Noncompliant Code Example public class Nothing { // Noncompliant } Compliant Solution public interface Nothing { } Exceptions Empty classes can be used as marker types (for Spring for instance), therefore empty classes that are annotated will be ignored. @Configuration @EnableWebMvc public final class ApplicationConfiguration { } | CODE\_SMELL | MINOR | 1 |
| Parsing should be used to convert "Strings" to primitives | Rather than creating a boxed primitive from a String to extract the primitive value, use the relevant parse method instead. It will be clearer and more efficient. Noncompliant Code Example String myNum = "12.2"; float f = (new Float(myNum)).floatValue(); // Noncompliant; creates &amp; discards a Float Compliant Solution String myNum = "12.2"; float f = Float.parseFloat(myNum); | CODE\_SMELL | MINOR | 4 |
| Catches should be combined | Since Java 7 it has been possible to catch multiple exceptions at once. Therefore, when multiple catch blocks have the same code, they should be combined for better readability. Note that this rule is automatically disabled when the project's sonar.java.source is lower than 7. Noncompliant Code Example catch (IOException e) { doCleanup(); logger.log(e); } catch (SQLException e) { // Noncompliant doCleanup(); logger.log(e); } catch (TimeoutException e) { // Compliant; block contents are different doCleanup(); throw e; } Compliant Solution catch (IOException|SQLException e) { doCleanup(); logger.log(e); } catch (TimeoutException e) { doCleanup(); throw e; } | CODE\_SMELL | MINOR | 14 |
| Subclasses that add fields should override "equals" | Extend a class that overrides equals and add fields without overriding equals in the subclass, and you run the risk of non-equivalent instances of your subclass being seen as equal, because only the superclass fields will be considered in the equality test. This rule looks for classes that do all of the following: extend classes that override equals. do not themselves override equals. add fields. Noncompliant Code Example public class Fruit { private Season ripe; public boolean equals(Object obj) { if (obj == this) { return true; } if (this.class != obj.class) { return false; } Fruit fobj = (Fruit) obj; if (ripe.equals(fobj.getRipe()) { return true; } return false; } } public class Raspberry extends Fruit { // Noncompliant; instances will use Fruit's equals method private Color ripeColor; } Compliant Solution public class Fruit { private Season ripe; public boolean equals(Object obj) { if (obj == this) { return true; } if (this.class != obj.class) { return false; } Fruit fobj = (Fruit) obj; if (ripe.equals(fobj.getRipe()) { return true; } return false; } } public class Raspberry extends Fruit { private Color ripeColor; public boolean equals(Object obj) { if (! super.equals(obj)) { return false; } Raspberry fobj = (Raspberry) obj; if (ripeColor.equals(fobj.getRipeColor()) { // added fields are tested return true; } return false; } } | CODE\_SMELL | MINOR | 1 |
| The diamond operator ("<>") should be used |  | CODE\_SMELL | MINOR | 382 |
| Nested "enum"s should not be declared static | According to the docs: Nested enum types are implicitly static. So there's no need to declare them static explicitly. Noncompliant Code Example public class Flower { static enum Color { // Noncompliant; static is redundant here RED, YELLOW, BLUE, ORANGE } // ... } Compliant Solution public class Flower { enum Color { // Compliant RED, YELLOW, BLUE, ORANGE } // ... } | CODE\_SMELL | MINOR | 3 |
| Static non-final field names should comply with a naming convention | Shared naming conventions allow teams to collaborate efficiently. This rule checks that static non-final field names match a provided regular expression. Noncompliant Code Example With the default regular expression ^[a-z][a-zA-Z0-9]\*$: public final class MyClass { private static String foo\_bar; } Compliant Solution class MyClass { private static String fooBar; } | CODE\_SMELL | MINOR | 46 |
| "private" methods called only by inner classes should be moved to those classes | When a private method is only invoked by an inner class, there's no reason not to move it into that class. It will still have the same access to the outer class' members, but the outer class will be clearer and less cluttered. Noncompliant Code Example public class Outie { private int i=0; private void increment() { // Noncompliant i++; } public class Innie { public void doTheThing() { Outie.this.increment(); } } } Compliant Solution public class Outie { private int i=0; public class Innie { public void doTheThing() { increment(); } private void increment() { Outie.this.i++; } } } | CODE\_SMELL | MINOR | 18 |
| Methods should not return constants | There's no point in forcing the overhead of a method call for a method that always returns the same constant value. Even worse, the fact that a method call must be made will likely mislead developers who call the method thinking that something more is done. Declare a constant instead. This rule raises an issue if on methods that contain only one statement: the return of a constant value. Noncompliant Code Example int getBestNumber() { return 12; // Noncompliant } Compliant Solution static int bestNumber = 12; Exceptions Methods with annotations, such as @Override and Spring's @RequestMapping, are ignored. | CODE\_SMELL | MINOR | 6 |
| Jump statements should not be redundant | Jump statements such as return and continue let you change the default flow of program execution, but jump statements that direct the control flow to the original direction are just a waste of keystrokes. Noncompliant Code Example public void foo() { while (condition1) { if (condition2) { continue; // Noncompliant } else { doTheThing(); } } return; // Noncompliant; this is a void method } Compliant Solution public void foo() { while (condition1) { if (!condition2) { doTheThing(); } } } | CODE\_SMELL | MINOR | 10 |
| Arrays should not be created for varargs parameters | There's no point in creating an array solely for the purpose of passing it as a varargs (...) argument; varargs is an array. Simply pass the elements directly. They will be consolidated into an array automatically. Incidentally passing an array where Object ... is expected makes the intent ambiguous: Is the array supposed to be one object or a collection of objects? Noncompliant Code Example public void callTheThing() { //... doTheThing(new String[] { "s1", "s2"}); // Noncompliant: unnecessary doTheThing(new String[12]); // Compliant doTheOtherThing(new String[8]); // Noncompliant: ambiguous // ... } public void doTheThing (String ... args) { // ... } public void doTheOtherThing(Object ... args) { // ... } Compliant Solution public void callTheThing() { //... doTheThing("s1", "s2"); doTheThing(new String[12]); doTheOtherThing((Object[]) new String[8]); // ... } public void doTheThing (String ... args) { // ... } public void doTheOtherThing(Object ... args) { // ... } | CODE\_SMELL | MINOR | 5 |
| Null checks should not be used with "instanceof" | There's no need to null test in conjunction with an instanceof test. null is not an instanceof anything, so a null check is redundant. Noncompliant Code Example if (x != null &amp;&amp; x instanceof MyClass) { ... } // Noncompliant if (x == null || ! x instanceof MyClass) { ... } // Noncompliant Compliant Solution if (x instanceof MyClass) { ... } if (! x instanceof MyClass) { ... } | CODE\_SMELL | MINOR | 5 |
| "StandardCharsets" constants should be preferred | JDK7 introduced the class java.nio.charset.StandardCharsets. It provides constants for all charsets that are guaranteed to be available on every implementation of the Java platform. ISO\_8859\_1 US\_ASCII UTF\_16 UTF\_16BE UTF\_16LE UTF\_8 These constants should be preferred to: - the use of a String such as "UTF-8" which has the drawback of requiring the catch/throw of an UnsupportedEncodingException that will never actually happen - the use of Guava’s Charsets class, which has been obsolete since JDK7 Noncompliant Code Example try { byte[] bytes = string.getBytes("UTF-8"); // Noncompliant; use a String instead of StandardCharsets.UTF\_8 } catch (UnsupportedEncodingException e) { throw new AssertionError(e); } // ... byte[] bytes = string.getBytes(Charsets.UTF\_8); // Noncompliant; Guava way obsolete since JDK7 Compliant Solution byte[] bytes = string.getBytes(StandardCharsets.UTF\_8) | CODE\_SMELL | MINOR | 6 |
| An iteration on a Collection should be performed on the type handled by the Collection | This rule raises an issue when an iteration over the items of a Collection is performed on a super-type of the type handled by the Collection. Relying on Object or any classes between Object and the real class handled by the Collection is not recommended. While it's accepted by the language, this practice reduces readability of the code and forces to down-cast the item of the Collection to be able to call a method on it while simply using the correct type in the iteration makes things more clear and simple. Noncompliant Code Example public Collection&lt;Person&gt; getPersons() { ... } for (Object item : getPersons()) { // Noncompliant Person person = (Person) item; // Noncompliant; it's required to down-cast to the to correct type to use "item" person.getAdress(); } Compliant Solution for (Person person : getPersons()) { // Compliant person.getAddress() ; } | CODE\_SMELL | MINOR | 1 |
| Boxed "Boolean" should be avoided in boolean expressions | When boxed type java.lang.Boolean is used as an expression it will throw NullPointerException if the value is null as defined in Java Language Specification §5.1.8 Unboxing Conversion. It is safer to avoid such conversion altogether and handle the null value explicitly. Noncompliant Code Example Boolean b = getBoolean(); if (b) { // Noncompliant, it will throw NPE when b == null foo(); } else { bar(); } Compliant Solution Boolean b = getBoolean(); if (Boolean.TRUE.equals(b)) { foo(); } else { bar(); // will be invoked for both b == false and b == null } See \* Java Language Specification §5.1.8 Unboxing Conversion | CODE\_SMELL | MINOR | 9 |
| Unnecessary imports should be removed | The imports part of a file should be handled by the Integrated Development Environment (IDE), not manually by the developer. Unused and useless imports should not occur if that is the case. Leaving them in reduces the code's readability, since their presence can be confusing. Noncompliant Code Example package my.company; import java.lang.String; // Noncompliant; java.lang classes are always implicitly imported import my.company.SomeClass; // Noncompliant; same-package files are always implicitly imported import java.io.File; // Noncompliant; File is not used import my.company2.SomeType; import my.company2.SomeType; // Noncompliant; 'SomeType' is already imported class ExampleClass { public String someString; public SomeType something; } Exceptions Imports for types mentioned in comments, such as Javadocs, are ignored. | CODE\_SMELL | MINOR | 15 |
| Using pseudorandom number generators (PRNGs) is security-sensitive | Using pseudorandom number generators (PRNGs) is security-sensitive. For example, it has led in the past to the following vulnerabilities: CVE-2013-6386 CVE-2006-3419 CVE-2008-4102 When software generates predictable values in a context requiring unpredictability, it may be possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information. As the java.util.Random class relies on a pseudorandom number generator, this class and relating java.lang.Math.random() method should not be used for security-critical applications or for protecting sensitive data. In such context, the java.security.SecureRandom class which relies on a cryptographically strong random number generator (RNG) should be used in place. Ask Yourself Whether the code using the generated value requires it to be unpredictable. It is the case for all encryption mechanisms or when a secret value, such as a password, is hashed. the function you use generates a value which can be predicted (pseudo-random). the generated value is used multiple times. an attacker can access the generated value. You are at risk if you answered yes to the first question and any of the following ones. Recommended Secure Coding Practices Use a cryptographically strong random number generator (RNG) like "java.security.SecureRandom" in place of this PRNG. Use the generated random values only once. You should not expose the generated random value. If you have to store it, make sure that the database or file is secure. Sensitive Code Example Random random = new Random(); // Questionable use of Random byte bytes[] = new byte[20]; random.nextBytes(bytes); // Check if bytes is used for hashing, encryption, etc... Compliant Solution SecureRandom random = new SecureRandom(); // Compliant for security-sensitive use cases byte bytes[] = new byte[20]; random.nextBytes(bytes); See OWASP Top 10 2017 Category A3 - Sensitive Data Exposure MITRE, CWE-338 - Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG) MITRE, CWE-330 - Use of Insufficiently Random Values MITRE, CWE-326 - Inadequate Encryption Strength CERT, MSC02-J. - Generate strong random numbers CERT, MSC30-C. - Do not use the rand() function for generating pseudorandom numbers CERT, MSC50-CPP. - Do not use std::rand() for generating pseudorandom numbers Derived from FindSecBugs rule Predictable Pseudo Random Number Generator | SECURITY\_HOTSPOT | CRITICAL | 2 |
| Changing or bypassing accessibility is security-sensitive | Changing or bypassing accessibility is security-sensitive. For example, it has led in the past to the following vulnerability: CVE-2012-4681 private methods were made private for a reason, and the same is true of every other visibility level. Altering or bypassing the accessibility of classes, methods, or fields violates the encapsulation principle and could introduce security holes. This rule raises an issue when reflection is used to change the visibility of a class, method or field, and when it is used to directly update a field value. Ask Yourself Whether there is a good reason to override the existing accessibility level of the method/field. This is very rarely the case. Accessing hidden fields and methods will make your code unstable as they are not part of the public API and may change in future versions. this method is called by untrusted code. \* it is possible to modify or bypass the accessibility of sensitive methods or fields using this code. \* untrusted code can access the java reflection API. \* \* You are at risk if you answered yes to those questions. Recommended Secure Coding Practices Don't change or bypass the accessibility of any method or field if possible. If untrusted code can execute this method, make sure that it cannot decide which method or field's accessibility can be modified or bypassed. Untrusted code should never have direct access to the java Reflection API. If this method can do it, make sure that it is an exception. Use ClassLoaders and SecurityManagers in order to sandbox any untrusted code and forbid access to the Reflection API. Sensitive Code Example public void makeItPublic(String methodName) throws NoSuchMethodException { this.getClass().getMethod(methodName).setAccessible(true); // Questionable } public void setItAnyway(String fieldName, int value) { this.getClass().getDeclaredField(fieldName).setInt(this, value); // Questionable; bypasses controls in setter } See OWASP Top 10 2017 Category A3 - Sensitive Data Exposure CERT, SEC05-J. - Do not use reflection to increase accessibility of classes, methods, or fields | SECURITY\_HOTSPOT | CRITICAL | 4 |
| Deserializing objects from an untrusted source is security-sensitive | Deserializing objects is security-sensitive. For example, it has led in the past to the following vulnerabilities: CVE-2018-10654: Hazelcast Library: Java deserialization vulnerability CVE-2018-1000058: Jenkins Pipeline: arbitrary code execution vulnerability Object deserialization from an untrusted source can lead to unexpected code execution. ObjectInputStream doesn't provide a way to apply rules on its InputStream argument. Knowing that all serializable classes in the scope of the classloader will be deserialized, there is a possibility that malicious code could be executed during the deserialization phase even if, in the end, a ClassCastException will be raised. Deserialization takes a stream of bits and turns it into an object. If the stream contains the type of object you expect, all is well. But if you're deserializing untrusted input, and an attacker has inserted some other type of object, you're in trouble. Why? There are a few different attack scenarios, but one widely-documented one goes like this: Deserialization first instantiates an Object, then uses the readObject method to populate it. If the attacker has overridden readObject then he is entirely in control of what code executes during that process. It is only after readObject has completed that your newly-minted Object can be cast to the type you expected. A ClassCastException or ClassNotFoundException will be thrown, but at that point it's too late. Ask Yourself Whether an attacker could have tampered with the source provided to the deserialization function. you are using an unsafe deserialization function. See the Recommended Secure Coding Practices for examples of safe libraries. You are at risk if you answered yes to any of those questions. Recommended Secure Coding Practices To prevent insecure deserialization, you should either use look-ahead deserialization (pre-Java 9) or a filter to make sure you're dealing with the correct type of object before you act on it. Several third-party libraries offer look-ahead deserialization, including: ikkisoft's SerialKiller Apache Commons Class IO's ValidatingObjectInputStream contrast-rO0's SafeObjectInputStream Note that it is possible to set a deserialization filter at the level of the JVM, but relying on that requires that your environment be configured perfectly. Every time. Additionally, such a filter may have unwanted impacts on other applications in the environment. On the other hand, setting a filter as close as possible to the deserialization that uses it allows you to specify a very narrow, focused filter. You should also limit access to the serialized source. For example: if it is a file, restrict the access to it. if it comes from the network, restrict who has access to the process, such as with a Firewall or by authenticating the sender first. See OWASP - Deserialization of untrusted data OWASP Top 10 2017 Category A8 - Insecure Deserialization MITRE, CWE-502 - Deserialization of Untrusted Data Derived from FindSecBugs rule OBJECT\_DESERIALIZATION | SECURITY\_HOTSPOT | CRITICAL | 2 |
| Using regular expressions is security-sensitive | Using regular expressions is security-sensitive. It has led in the past to the following vulnerabilities: CVE-2017-16021 CVE-2018-13863 Evaluating regular expressions against input strings is potentially an extremely CPU-intensive task. Specially crafted regular expressions such as (a+)+s will take several seconds to evaluate the input string aaaaaaaaaaaaaaaaaaaaaaaaaaaaabs. The problem is that with every additional a character added to the input, the time required to evaluate the regex doubles. However, the equivalent regular expression, a+s (without grouping) is efficiently evaluated in milliseconds and scales linearly with the input size. Evaluating such regular expressions opens the door to Regular expression Denial of Service (ReDoS) attacks. In the context of a web application, attackers can force the web server to spend all of its resources evaluating regular expressions thereby making the service inaccessible to genuine users. This rule flags any execution of a hardcoded regular expression which has at least 3 characters and at least two instances of any of the following characters: \*+{. Example: (a+)\* Ask Yourself Whether the executed regular expression is sensitive and a user can provide a string which will be analyzed by this regular expression. your regular expression engine performance decrease with specially crafted inputs and regular expressions. You may be at risk if you answered yes to any of those questions. Recommended Secure Coding Practices Check whether your regular expression engine (the algorithm executing your regular expression) has any known vulnerabilities. Search for vulnerability reports mentioning the one engine you're are using. Use if possible a library which is not vulnerable to Redos Attacks such as Google Re2. Remember also that a ReDos attack is possible if a user-provided regular expression is executed. This rule won't detect this kind of injection. Sensitive Code Example import java.util.regex.Pattern; class BasePattern { String regex = "(a+)+b"; // a regular expression String input; // a user input void foo(CharSequence htmlString) { input.matches(regex); // Sensitive Pattern.compile(regex); // Sensitive Pattern.compile(regex, Pattern.CASE\_INSENSITIVE); // Sensitive String replacement = "test"; input.replaceAll(regex, replacement); // Sensitive input.replaceFirst(regex, replacement); // Sensitive if (!Pattern.matches(".\*&lt;script&gt;(a+)+b", htmlString)) { // Sensitive } } } This also applies for bean validation, where regexp can be specified: import java.io.Serializable; import javax.validation.constraints.Pattern; import javax.validation.constraints.Email; import org.hibernate.validator.constraints.URL; class BeansRegex implements Serializable { @Pattern(regexp=".+@(a+)+b") // Sensitive private String email; @Email(regexp=".+@(a+)+b") // Sensitive private String email2; @URL(regexp="(a+)+b.com") // Sensitive private String url; // ... } Exceptions Calls to String.split(regex) and String.split(regex, limit) will not raise an exception despite their use of a regular expression. These methods are used most of the time to split on simple regular expressions which don't create any vulnerabilities. Some corner-case regular expressions will not raise an issue even though they might be vulnerable. For example: (a|aa)+, (a|a?)+. It is a good idea to test your regular expression if it has the same pattern on both side of a "|". See OWASP Top 10 2017 Category A1 - Injection MITRE, CWE-624 - Executable Regular Expression Error OWASP Regular expression Denial of Service - ReDoS | SECURITY\_HOTSPOT | CRITICAL | 2 |
| Hashing data is security-sensitive | Hashing data is security-sensitive. It has led in the past to the following vulnerabilities: CVE-2018-9233 CVE-2013-5097 CVE-2007-1051 Cryptographic hash functions are used to uniquely identify information without storing their original form. When not done properly, an attacker can steal the original information by guessing it (ex: with a rainbow table), or replace the original data with another one having the same hash. This rule flags code that initiates hashing. Ask Yourself Whether the hashed value is used in a security context. the hashing algorithm you are using is known to have vulnerabilities. salts are not automatically generated and applied by the hashing function. any generated salts are cryptographically weak or not credential-specific. You are at risk if you answered yes to the first question and any of the following ones. Recommended Secure Coding Practices for security related purposes, use only hashing algorithms which are currently known to be strong. Avoid using algorithms like MD5 and SHA1 completely in security contexts. do not define your own hashing- or salt algorithms as they will most probably have flaws. do not use algorithms that compute too quickly, like SHA256, as it must remain beyond modern hardware capabilities to perform brute force and dictionary based attacks. use a hashing algorithm that generate its own salts as part of the hashing. If you generate your own salts, make sure that a cryptographically strong salt algorithm is used, that generated salts are credential-specific, and finally, that the salt is applied correctly before the hashing. save both the salt and the hashed value in the relevant database record; during future validation operations, the salt and hash can then be retrieved from the database. The hash is recalculated with the stored salt and the value being validated, and the result compared to the stored hash. the strength of hashing algorithms often decreases over time as hardware capabilities increase. Check regularly that the algorithms you are using are still considered secure. If needed, rehash your data using a stronger algorithm. Sensitive Code Example // === MessageDigest === import java.security.MessageDigest; import java.security.Provider; class A { void foo(String algorithm, String providerStr, Provider provider) throws Exception { MessageDigest.getInstance(algorithm); // Sensitive MessageDigest.getInstance(algorithm, providerStr); // Sensitive MessageDigest.getInstance(algorithm, provider); // Sensitive } } Regarding SecretKeyFactory. Any call to SecretKeyFactory.getInstance("...") with an argument starting by "PBKDF2" will be highlighted. See OWASP guidelines, list of standard algorithms and algorithms on android. // === javax.crypto === import javax.crypto.spec.PBEKeySpec; import javax.crypto.SecretKeyFactory; class A { void foo(char[] password, byte[] salt, int iterationCount, int keyLength) throws Exception { // Sensitive. Review this, even if it is the way recommended by OWASP SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA512"); PBEKeySpec spec = new PBEKeySpec(password, salt, iterationCount, keyLength); factory.generateSecret(spec).getEncoded(); } } Regarding Guava, only the hashing functions which are usually misused for sensitive data will raise an issue, i.e. md5 and sha\*. // === Guava === import com.google.common.hash.Hashing; class A { void foo() { Hashing.md5(); // Sensitive Hashing.sha1(); // Sensitive Hashing.sha256(); // Sensitive Hashing.sha384(); // Sensitive Hashing.sha512(); // Sensitive } } // === org.apache.commons === import org.apache.commons.codec.digest.DigestUtils; class A { void foo(String strName, byte[] data, String str, java.io.InputStream stream) throws Exception { new DigestUtils(strName); // Sensitive new DigestUtils(); // Sensitive DigestUtils.getMd2Digest(); // Sensitive DigestUtils.getMd5Digest(); // Sensitive DigestUtils.getShaDigest(); // Sensitive DigestUtils.getSha1Digest(); // Sensitive DigestUtils.getSha256Digest(); // Sensitive DigestUtils.getSha384Digest(); // Sensitive DigestUtils.getSha512Digest(); // Sensitive DigestUtils.md2(data); // Sensitive DigestUtils.md2(stream); // Sensitive DigestUtils.md2(str); // Sensitive DigestUtils.md2Hex(data); // Sensitive DigestUtils.md2Hex(stream); // Sensitive DigestUtils.md2Hex(str); // Sensitive DigestUtils.md5(data); // Sensitive DigestUtils.md5(stream); // Sensitive DigestUtils.md5(str); // Sensitive DigestUtils.md5Hex(data); // Sensitive DigestUtils.md5Hex(stream); // Sensitive DigestUtils.md5Hex(str); // Sensitive DigestUtils.sha(data); // Sensitive DigestUtils.sha(stream); // Sensitive DigestUtils.sha(str); // Sensitive DigestUtils.shaHex(data); // Sensitive DigestUtils.shaHex(stream); // Sensitive DigestUtils.shaHex(str); // Sensitive DigestUtils.sha1(data); // Sensitive DigestUtils.sha1(stream); // Sensitive DigestUtils.sha1(str); // Sensitive DigestUtils.sha1Hex(data); // Sensitive DigestUtils.sha1Hex(stream); // Sensitive DigestUtils.sha1Hex(str); // Sensitive DigestUtils.sha256(data); // Sensitive DigestUtils.sha256(stream); // Sensitive DigestUtils.sha256(str); // Sensitive DigestUtils.sha256Hex(data); // Sensitive DigestUtils.sha256Hex(stream); // Sensitive DigestUtils.sha256Hex(str); // Sensitive DigestUtils.sha384(data); // Sensitive DigestUtils.sha384(stream); // Sensitive DigestUtils.sha384(str); // Sensitive DigestUtils.sha384Hex(data); // Sensitive DigestUtils.sha384Hex(stream); // Sensitive DigestUtils.sha384Hex(str); // Sensitive DigestUtils.sha512(data); // Sensitive DigestUtils.sha512(stream); // Sensitive DigestUtils.sha512(str); // Sensitive DigestUtils.sha512Hex(data); // Sensitive DigestUtils.sha512Hex(stream); // Sensitive DigestUtils.sha512Hex(str); // Sensitive } } See OWASP Top 10 2017 Category A3 - Sensitive Data Exposure OWASP Top 10 2017 Category A6 - Security Misconfiguration MITRE, CWE-916 - Use of Password Hash With Insufficient Computational Effort MITRE, CWE-759 - Use of a One-Way Hash without a Salt MITRE, CWE-760 - Use of a One-Way Hash with a Predictable Salt SANS Top 25 - Porous Defenses | SECURITY\_HOTSPOT | CRITICAL | 1 |
| Using Sockets is security-sensitive | Using sockets is security-sensitive. It has led in the past to the following vulnerabilities: CVE-2011-178 CVE-2017-5645 CVE-2018-6597 Sockets are vulnerable in multiple ways: They enable a software to interact with the outside world. As this world is full of attackers it is necessary to check that they cannot receive sensitive information or inject dangerous input. The number of sockets is limited and can be exhausted. Which makes the application unresponsive to users who need additional sockets. This rules flags code that creates sockets. It matches only the direct use of sockets, not use through frameworks or high-level APIs such as the use of http connections. Ask Yourself Whether sockets are created without any limit every time a user performs an action. input received from sockets is used without being sanitized. sensitive data is sent via sockets without being encrypted. You are at risk if you answered yes to any of these questions. Recommended Secure Coding Practices In many cases there is no need to open a socket yourself. Use instead libraries and existing protocols. Encrypt all data sent if it is sensitive. Usually it is better to encrypt it even if the data is not sensitive as it might change later. Sanitize any input read from the socket. Limit the number of sockets a given user can create. Close the sockets as soon as possible. Sensitive Code Example // === java.net === import java.net.Socket; import java.net.InetAddress; import java.net.Proxy; import java.net.ServerSocket; import javax.net.SocketFactory; class A { void foo(SocketFactory factory, String address, int port, InetAddress localAddr, int localPort, boolean stream, String host, Proxy proxy, int backlog, InetAddress bindAddr) throws Exception { new Socket(); // Questionable. new Socket(address, port); // Questionable. new Socket(address, port, localAddr, localPort); // Questionable. new Socket(host, port, stream); // Questionable. new Socket(proxy); // Questionable. new Socket(host, port); // Questionable. new Socket(host, port, stream); // Questionable. new Socket(host, port, localAddr, localPort); // Questionable. new ServerSocket(); // Questionable. new ServerSocket(port); // Questionable. new ServerSocket(port, backlog); // Questionable. new ServerSocket(port, backlog, bindAddr); // Questionable. factory.createSocket(); // Questionable } } abstract class mySocketFactory extends SocketFactory { // Questionable. Review how the sockets are created. // ... } // === java.nio.channels === import java.net.SocketAddress; import java.nio.channels.AsynchronousChannelGroup; import java.nio.channels.AsynchronousServerSocketChannel; import java.nio.channels.AsynchronousSocketChannel; import java.nio.channels.SocketChannel; import java.nio.channels.ServerSocketChannel; class A { void foo(AsynchronousChannelGroup group, SocketAddress remote) throws Exception { AsynchronousServerSocketChannel.open(); // Questionable. AsynchronousServerSocketChannel.open(group); // Questionable. AsynchronousSocketChannel.open(); // Questionable. AsynchronousSocketChannel.open(group); // Questionable. SocketChannel.open(); // Questionable. SocketChannel.open(remote); // Questionable. ServerSocketChannel.open(); // Questionable. } } // === Netty === import io.netty.channel.ChannelInitializer; import io.netty.channel.socket.ServerSocketChannel; import io.netty.channel.socket.SocketChannel; class CustomChannelInitializer extends ChannelInitializer&lt;ServerSocketChannel&gt; { // Questionable. Review how the SocketChannel is used. @Override protected void initChannel(ServerSocketChannel ch) throws Exception { } } class A { void foo() { new ChannelInitializer&lt;SocketChannel&gt;() { // Questionable @Override public void initChannel(SocketChannel ch) throws Exception { // ... } }; } } See OWASP Top 10 2017 Category A3 - Sensitive Data Exposure MITRE, CWE-20 - Improper Input Validation MITRE, CWE-400 - Uncontrolled Resource Consumption ('Resource Exhaustion') MITRE, CWE-200 - Information Exposure SANS Top 25 - Risky Resource Management SANS Top 25 - Porous Defenses | SECURITY\_HOTSPOT | CRITICAL | 3 |
| Using command line arguments is security-sensitive |  | SECURITY\_HOTSPOT | CRITICAL | 5 |
| Reading the Standard Input is security-sensitive | Reading Standard Input is security-sensitive. It has led in the past to the following vulnerabilities: CVE-2005-2337 CVE-2017-11449 It is common for attackers to craft inputs enabling them to exploit software vulnerabilities. Thus any data read from the standard input (stdin) can be dangerous and should be validated. This rule flags code that reads from the standard input. Ask Yourself Whether data read from the standard input is not sanitized before being used. You are at risk if you answered yes to this question. Recommended Secure Coding Practices Sanitize all data read from the standard input before using it. Sensitive Code Example class A { void foo(String fmt, Object args) throws Exception { // Questionable. Check how the standard input is used. System.in.read(); // Questionable. Check how safe this new InputStream is. System.setIn(new java.io.FileInputStream("test.txt")); java.io.Console console = System.console(); // Questionable. All the following calls should be reviewed as they use the standard input. console.reader(); console.readLine(); console.readLine(fmt, args); console.readPassword(); console.readPassword(fmt, args); } } Exceptions All references to System.in will create issues except direct calls to System.in.close(). Command line parsing libraries such as JCommander often read standard input when asked for passwords. However this rule doesn't raise any issue in this case as another hotspot rule covers command line arguments. See: MITRE, CWE-20 - Improper Input Validation | SECURITY\_HOTSPOT | CRITICAL | 3 |
| Expanding archive files is security-sensitive | Expanding archive files is security-sensitive. For example, expanding archive files has led in the past to the following vulnerabilities: CVE-2018-1263 CVE-2018-16131 Applications that expand archive files (zip, tar, jar, war, 7z, ...) should verify the path where the archive's files are expanded and not trust blindly the content of the archive. Archive's files should not be expanded outside of the root directory where the archive is supposed to be expanded. Also, applications should control the size of the expanded data to not be a victim of Zip Bomb attack. Failure to do so could allow an attacker to use a specially crafted archive that holds directory traversal paths (e.g. ../../attacker.sh) or the attacker could overload the file system, processors or memory of the operating system where the archive is expanded making the target OS completely unusable. This rule raises an issue when code handle archives. The goal is to guide security code reviews. Ask Yourself Whether there is no validation of the name of the archive entry there is no validation of the effective path where the archive entry is going to be expanded there is no validation of the size of the expanded archive entry there is no validation of the ratio between the compressed and uncompressed archive entry You are at risk if you answered yes to any of those questions. Recommended Secure Coding Practices Validate the full path of the extracted file against the full path of the directory where files are uncompressed. the canonical path of the uncompressed file must start with the canonical path of the directory where files are extracted. the name of the archive entry must not contain "..", i.e. reference to a parent directory. String canonicalDirPath = outputDir.getCanonicalPath(); String canonicalDestPath = targetFile.getCanonicalPath(); if (!canonicalDestPath.startsWith(canonicalDirPath + File.separator)) { // Sanitizer throw new ArchiverException("Entry is trying to leave the target dir: " + zipEntry.getName()); } Stop extracting the archive if any of its entries has been tainted with a directory traversal path. Define and control the ratio between compressed and uncompress bytes. Define and control the maximum allowed uncompressed file size. Count the number of file entries extracted from the archive and abort the extraction if their number is greater than a predefined threshold. Sensitive Code Example java.util.zip.ZipFile zipFile = new ZipFile(zipFileName); Enumeration&lt;? extends ZipEntry&gt; entries = zipFile.entries(); while (entries.hasMoreElements()) { ZipEntry e = entries.nextElement(); // Questionable File f = new File(outputDir, e.getName()); InputStream input = zipFile.getInputStream(e); extractFile(new ZipInputStream(input), outputDir, e.getName()); } Exceptions This rule doesn't raise an issue when a ZipEntry or a ArchiveEntry: is declared as a class field is a parameter of an abstract method of an interface or abstract class See OWASP Top 10 2017 Category A1 - Injection MITRE, CWE-409 - Improper Handling of Highly Compressed Data (Data Amplification) CERT, IDS04-J. - Safely extract files from ZipInputStream Snyk Research Team: Zip Slip Vulnerability | SECURITY\_HOTSPOT | CRITICAL | 7 |
| Using environment variables is security-sensitive | Using environment variables is security-sensitive. For example, their use has led in the past to the following vulnerabilities: CVE-2014-6278 CVE-2019-3464 CVE-2018-1000402 CVE-2016-10530 Environment variables are sensitive to injection attacks, just like any other input. Note also that environment variables can be exposed in multiple ways, storing sensitive information in them should be done carefully: on Unix systems environment variables of one process can be read by another process running with the same UID. environment variables might be forwarded to child processes. application running in debug mode often exposes their environment variable. This rule raises an issue when environment variables are read. Ask Yourself Whether Environment variables are used without being sanitized. You store sensitive information in environment variables and other processes might be able to access them. You are at risk if you answered yes to any of those questions. Recommended Secure Coding Practices Sanitize every environment variable before using its value. If you store sensitive information in an environment variable, make sure that no other process can access them, i.e. the process runs with a separate user account and child processes don't have access to their parent's environment. Don't run your application in debug mode if it has access to sensitive information, including environment variables. Sensitive Code Example public class Main { public static void main (String[] args) { System.getenv(); // Sensitive System.getenv("myvar"); // Sensitive ProcessBuilder processBuilder = new ProcessBuilder(); Map&lt;String, String&gt; environment = processBuilder.environment(); // Sensitive environment.put("VAR", "value"); Runtime.getRuntime().exec("ping", new String[]{"env=val"}); // Sensitive } } See MITRE, CWE-526 - Information Exposure Through Environmental Variables MITRE, CWE-74 - Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection') | SECURITY\_HOTSPOT | CRITICAL | 5 |
| Using hardcoded IP addresses is security-sensitive | Hardcoding IP addresses is security-sensitive. It has led in the past to the following vulnerabilities: CVE-2006-5901 CVE-2005-3725 Today's services have an ever-changing architecture due to their scaling and redundancy needs. It is a mistake to think that a service will always have the same IP address. When it does change, the hardcoded IP will have to be modified too. This will have an impact on the product development, delivery and deployment: The developers will have to do a rapid fix every time this happens, instead of having an operation team change a configuration file. It forces the same address to be used in every environment (dev, sys, qa, prod). Last but not least it has an effect on application security. Attackers might be able to decompile the code and thereby discover a potentially sensitive address. They can perform a Denial of Service attack on the service at this address or spoof the IP address. Such an attack is always possible, but in the case of a hardcoded IP address the fix will be much slower, which will increase an attack's impact. Recommended Secure Coding Practices make the IP address configurable. Noncompliant Code Example String ip = "192.168.12.42"; // Noncompliant Socket socket = new Socket(ip, 6667); Exceptions No issue is reported for the following cases because they are not considered sensitive: Loopback addresses 127.0.0.0/8 in CIDR notation (from 127.0.0.0 to 127.255.255.255) Broadcast address 255.255.255.255 Non routable address 0.0.0.0 Strings of the form 2.5.&lt;number&gt;.&lt;number&gt; as they often match Object Identifiers (OID). See OWASP Top 10 2017 Category A3 - Sensitive Data Exposure CERT, MSC03-J. - Never hard code sensitive information | SECURITY\_HOTSPOT | MINOR | 1 |
| Untrusted XML should be parsed without resolving external data |  | VULNERABILITY | BLOCKER | 3 |
| Class variable fields should not have public accessibility | Public class variable fields do not respect the encapsulation principle and has three main disadvantages: Additional behavior such as validation cannot be added. The internal representation is exposed, and cannot be changed afterwards. Member values are subject to change from anywhere in the code and may not meet the programmer's assumptions. By using private attributes and accessor methods (set and get), unauthorized modifications are prevented. Noncompliant Code Example public class MyClass { public static final int SOME\_CONSTANT = 0; // Compliant - constants are not checked public String firstName; // Noncompliant } Compliant Solution public class MyClass { public static final int SOME\_CONSTANT = 0; // Compliant - constants are not checked private String firstName; // Compliant public String getFirstName() { return firstName; } public void setFirstName(String firstName) { this.firstName = firstName; } } Exceptions Because they are not modifiable, this rule ignores public final fields. See MITRE, CWE-493 - Critical Public Variable Without Final Modifier | VULNERABILITY | MINOR | 161 |
| Throwable.printStackTrace(...) should not be called | Throwable.printStackTrace(...) prints a Throwable and its stack trace to some stream. By default that stream System.Err, which could inadvertently expose sensitive information. Loggers should be used instead to print Throwables, as they have many advantages: Users are able to easily retrieve the logs. The format of log messages is uniform and allow users to browse the logs easily. This rule raises an issue when printStackTrace is used without arguments, i.e. when the stack trace is printed to the default stream. Noncompliant Code Example try { /\* ... \*/ } catch(Exception e) { e.printStackTrace(); // Noncompliant } Compliant Solution try { /\* ... \*/ } catch(Exception e) { LOGGER.log("context", e); } See OWASP Top 10 2017 Category A3 - Sensitive Data Exposure MITRE, CWE-489 - Leftover Debug Code | VULNERABILITY | MINOR | 12 |
| "public static" fields should be constant | There is no good reason to declare a field "public" and "static" without also declaring it "final". Most of the time this is a kludge to share a state among several objects. But with this approach, any object can do whatever it wants with the shared state, such as setting it to null. Noncompliant Code Example public class Greeter { public static Foo foo = new Foo(); ... } Compliant Solution public class Greeter { public static final Foo FOO = new Foo(); ... } See MITRE, CWE-500 - Public Static Field Not Marked Final CERT OBJ10-J. - Do not use public static nonfinal fields | VULNERABILITY | MINOR | 47 |
| Mutable fields should not be "public static" | There is no good reason to have a mutable object as the public (by default), static member of an interface. Such variables should be moved into classes and their visibility lowered. Similarly, mutable static members of classes and enumerations which are accessed directly, rather than through getters and setters, should be protected to the degree possible. That can be done by reducing visibility or making the field final if appropriate. Note that making a mutable field, such as an array, final will keep the variable from being reassigned, but doing so has no effect on the mutability of the internal state of the array (i.e. it doesn't accomplish the goal). This rule raises issues for public static array, Collection, Date, and awt.Point members. Noncompliant Code Example public interface MyInterface { public static String [] strings; // Noncompliant } public class A { public static String [] strings1 = {"first","second"}; // Noncompliant public static String [] strings2 = {"first","second"}; // Noncompliant public static List&lt;String&gt; strings3 = new ArrayList&lt;&gt;(); // Noncompliant // ... } See MITRE, CWE-582 - Array Declared Public, Final, and Static MITRE, CWE-607 - Public Static Final Field References Mutable Object CERT, OBJ01-J. - Limit accessibility of fields CERT, OBJ13-J. - Ensure that references to mutable objects are not exposed | VULNERABILITY | MINOR | 7 |
| Return values should not be ignored when they contain the operation status code | When the return value of a function call contain the operation status code, this value should be tested to make sure the operation completed successfully. This rule raises an issue when the return values of the following are ignored: java.io.File operations that return a status code (except mkdirs) Iterator.hasNext() Enumeration.hasMoreElements() Lock.tryLock() non-void Condition.await\* methods CountDownLatch.await(long, TimeUnit) Semaphore.tryAcquire BlockingQueue: offer, remove Noncompliant Code Example public void doSomething(File file, Lock lock) { file.delete(); // Noncompliant // ... lock.tryLock(); // Noncompliant } Compliant Solution public void doSomething(File file, Lock lock) { if (!lock.tryLock()) { // lock failed; take appropriate action } if (!file.delete()) { // file delete failed; take appropriate action } } See MISRA C:2004, 16.10 - If a function returns error information, then that error information shall be tested MISRA C++:2008, 0-1-7 - The value returned by a function having a non-void return type that is not an overloaded operator shall always be used. MISRA C:2012, Dir. 4.7 - If a function returns error information, then that error information shall be tested MISRA C:2012, 17.7 - The value returned by a function having non-void return type shall be used CERT, ERR33-C. - Detect and handle standard library errors CERT, POS54-C. - Detect and handle POSIX library errors CERT, EXP00-J. - Do not ignore values returned by methods CERT, EXP12-C. - Do not ignore values returned by functions CERT, FIO02-J. - Detect and handle file-related errors MITRE, CWE-754 - Improper Check for Unusual Exceptional Conditions | VULNERABILITY | MINOR | 28 |