Id	Name	Severity	Description
1	Field names should start with a lower case letter	MAJOR	Names of fields that are not final should be in mixed case with a lowercase first letter and the first letters of subsequent words capitalized. This rule is deprecated, use {rule:squid:S00116} instead.
2	Negating the result of compareTo()/compare()	MINOR	This code negatives the return value of a compareTo or compare method. This is a questionable or bad programming practice, since if the return value is Integer.MIN_VALUE, negating the return value won't negate the sign of the result. You can achieve the same intended result by reversing the order of the operands rather than by negating the results.

3	Bad practice - Equals checks for noncompatible operand	MAJOR	This equals method is checking to see if the argument is some incompatible type
			(i.e., a class that is neither a supertype nor subtype of the class that defines
			the equals method). For example, the Foo class might have an equals method
			that looks like:
			`public boolean equals(Object o) {`
			`  if (o instanceof Foo)`
			`    return name.equals(((Foo)o).name);`
			`  else if (o instanceof String)`
			`    return name.equals(o);`
			`  else return false;`
			This is considered bad practice, as it makes it very hard to implement an equals method that
			is symmetric and transitive. Without those properties, very unexpected behavoirs are possible.
			This rule is deprecated, use {rule:squid:S2162} instead.

4	Correctness - Value that might carry a type qualifier is always used in a way prohibits it from having that type qualifier	CRITICAL	A value that is annotated as possibility being an instance of the values denoted by the type qualifier, and the value is guaranteed to be used in a way that prohibits values denoted by that type qualifier.
5	Correctness - Format string placeholder incompatible with passed argument	CRITICAL	The format string placeholder is incompatible with the corresponding argument. For example, `System.out.println("%d\n", "hello");` The %d placeholder requires a numeric argument, but a string value is passed instead. A runtime exception will occur when this statement is executed.

			This rule is deprecated, use {rule:squid:S2275} instead.
6	Performance - Private method is never called	CRITICAL	This private method is never called. Although it is possible that the method will be invoked through reflection, it is more likely that the method is never used, and should be removed. This rule is deprecated, use {rule:squid:UnusedPrivateMethod} instead.
7	Dodgy - Thread passed where Runnable expected	MAJOR	A Thread object is passed as a parameter to a method where a Runnable is expected. This is rather unusual, and may indicate a logic error or cause unexpected behavior. This rule is deprecated, use {rule:squid:S2438} instead.
8	Unchecked/unconfirmed cast of return value from method	CRITICAL	This code performs an unchecked cast of the return value of a method. The code might be calling the method in such a way that the cast is guaranteed to be safe, but FindBugs is unable to verify that the cast is safe. Check that your program logic ensures that this cast will not fail.
9	Multithreaded correctness - A volatile reference to an array doesn't treat the array elements as volatile	MAJOR	This declares a volatile reference to an array, which might not be what you want. With a volatile reference to an array, reads and writes of the reference to the array are treated as volatile, but the array elements are non-volatile. To get volatile array elements, you will need to use one of the atomic array classes in java.util.concurrent (provided in Java 5.0).
10	Performance - Method invokes inefficient Number constructor; use static valueOf instead	CRITICAL	Using `new Integer(int)` is guaranteed to always result in a new object whereas `Integer.valueOf(int)` allows caching of values to be done by the compiler, class library, or JVM. Using of cached values avoids object allocation and the code will be faster. Values between -128 and 127 are guaranteed to have corresponding cached instances and using `valueOf` is approximately 3.5 times faster than using constructor. For values outside the constant range the performance of both styles is the same. Unless the class must be compatible with JVMs predating Java 1.5, use either autoboxing or the `valueOf()` method when creating instances of `Long` , `Integer` , `Short` , `Character` , and `Byte` .
11	Dodgy - Redundant nullcheck of value known to be null	CRITICAL	This method contains a redundant check of a known null value against the constant null.

<table>
<tr><th></th><th></th><th></th><th></th></tr>
<tr><td>12</td><td>Correctness - TestCase defines tearDown that doesn't call super.tearDown()</td><td>CRITICAL</td><td>Class is a JUnit TestCase and implements the tearDown method. The tearDown method should call super.tearDown(), but doesn't.</td></tr>
<tr><td>13</td><td>Performance - Method allocates a boxed primitive just to call toString</td><td>MAJOR</td><td>A boxed primitive is allocated just to call toString(). It is more effective to just use the static form of toString which takes the primitive value. So,
<table>
<tr><th>Replace...</th><th>With this...</th></tr>
<tr><td>new Integer(1).toString()</td><td>Integer.toString(1)</td></tr>
<tr><td>new Long(1).toString()</td><td>Long.toString(1)</td></tr>
<tr><td>new Float(1.0).toString()</td><td>Float.toString(1.0)</td></tr>
<tr><td>new Double(1.0).toString()</td><td>Double.toString(1.0)</td></tr>
<tr><td>new Byte(1).toString()</td><td>Byte.toString(1)</td></tr>
<tr><td>new Short(1).toString()</td><td>Short.toString(1)</td></tr>
<tr><td>new Boolean(true).toString()</td><td>Boolean.toString(true)</td></tr>
</table>
This rule is deprecated, use {rule:squid:S2131} instead.</td></tr>
<tr><td>14</td><td>Class defines equal(Object); should it be equals(Object)?</td><td>CRITICAL</td><td>This class defines a method `equal(Object)`.
This method does not override the `equals(Object)` method in `java.lang.Object`, which is probably what was intended.
This rule is deprecated, use {rule:squid:S1221} instead.</td></tr>
<tr><td>15</td><td>Correctness - TestCase implements a non-static suite method</td><td>CRITICAL</td><td>Class is a JUnit TestCase and implements the suite() method.
The suite method should be declared as being static, but isn't.</td></tr>
<tr><td>16</td><td>Correctness - Integer multiply of result of integer remainder</td><td>CRITICAL</td><td>The code multiplies the result of an integer remaining by an integer constant.
Be sure you don't have your operator precedence confused. For example
i % 60 * 1000 is (i % 60) * 1000, not i % (60 * 1000).</td></tr>
<tr><td>17</td><td>Bad practice - Method ignores results of InputStream.skip()</td><td>MAJOR</td><td>This method ignores the return value of `java.io.InputStream.skip()` which can skip multiple bytes.
If the return value is not checked, the caller will not be able to correctly handle the case where fewer bytes were skipped than the caller requested.
This is a particularly insidious kind of bug, because in many programs, skips from input streams usually do skip the full amount of data requested, causing the program to fail only sporadically. With Buffered streams, however,</td></tr>
</table>

			skip() will only skip data in the buffer, and will routinely fail to skip the requested number of bytes.
18	Dodgy - Method uses the same code for two switch clauses	CRITICAL	This method uses the same code to implement two clauses of a switch statement. This could be a case of duplicate code, but it might also indicate a coding mistake. This rule is deprecated, use {rule:squid:S1871} instead.
19	Correctness - Non-virtual method call passes null for nonnull parameter	CRITICAL	A possibly-null value is passed to a nonnull method parameter. Either the parameter is annotated as a parameter that should always be nonnull, or analysis has shown that it will always be dereferenced.

20	Correctness - Method ignores return value	MINOR	The return value of this method should be checked. One common cause of this warning is to invoke a method on an immutable object, thinking that it updates the object. For example, in the following code fragment,
			`String dateString = getHeaderField(name);`
			`dateString.trim();`
			the programmer seems to be thinking that the trim() method will update the String referenced by dateString. But since Strings are immutable, the trim() function returns a new String value, which is being ignored here. The code should be corrected to:
			`String dateString = getHeaderField(name);`
			`dateString = dateString.trim();`
			This rule is deprecated, use {rule:squid:S2201} instead.

			This code creates an exception (or error) object, but doesn't do anything with it. For example, something like

21	Correctness - Exception created and dropped rather than thrown	CRITICAL	`if (x < 0)`
			`  new IllegalArgumentException("x must be nonnegative");`
			It was probably the intent of the programmer to throw the created exception:
			`if (x < 0)`
			`  throw new IllegalArgumentException("x must be nonnegative");`
			This rule is deprecated, use {rule:squid:S1848} instead.

22	Bad practice - Classloaders should only be created inside doPrivileged block	MAJOR	This code creates a classloader, which requires a security manager. If this code will be granted security permissions, but might be invoked by code that does not have security permissions, then the classloader creation needs to occur inside a doPrivileged block.
23	Field isn't final but should be refactored to be so	MAJOR	This static field public but not final, and could be changed by malicious code or by accident from another package. The field could be made final to avoid this vulnerability. However, the static initializer contains more than one write to the field, so doing so will require some refactoring. This rule is deprecated, use {rule:squid:S1444} instead.

24	Internationalization - Consider using Locale parameterized version of invoked method	INFO	A String is being converted to upper or lowercase, using the platform's default encoding. This may result in improper conversions when used with international characters. Use the
			String.toUpperCase(Locale l)
			String.toLowerCase(Locale l)
			versions instead.

25	Dodgy - Vacuous comparison of integer value	CRITICAL	There is an integer comparison that always returns the same value (e.g., x <= Integer.MAX_VALUE).
26	Malicious code vulnerability - Field should be both final and	MAJOR	A mutable static field could be changed by malicious code or by accident from another package. The field could be made package protected and/or made final to avoid

	package protected		this vulnerability.
27	Don't reuse entry objects in iterators	MAJOR	The entrySet() method is allowed to return a view of the underlying Map in which an `Iterator` and `Map.Entry`. This clever idea was used in several Map implementations, but introduces the possibility of nasty coding mistakes. If a map m returns such an iterator for an entrySet, then `c.addAll(m.entrySet())` will go badly wrong. All of the Map implementations in OpenJDK 1.7 have been rewritten to avoid this, you should to.
28	Correctness - Value that might not carry a type qualifier is always used in a way requires that type qualifier	CRITICAL	A value that is annotated as possibility not being an instance of the values denoted by the type qualifier, and the value is guaranteed to be used in a way that requires values denoted by that type qualifier.
29	Bad practice - Random object created and used only once	CRITICAL	This code creates a java.util.Random object, uses it to generate one random number, and then discards the Random object. This produces mediocre quality random numbers and is inefficient. If possible, rewrite the code so that the Random object is created once and saved, and each time a new random number is required invoke a method on the existing Random object to obtain it. If it is important that the generated Random numbers not be guessable, you *must* not create a new Random for each random number; the values are too easily guessable. You should strongly consider using a java.security.SecureRandom instead (and avoid allocating a new SecureRandom for each random number needed).
30	Performance - Primitive value is boxed then unboxed to perform primitive coercion	MAJOR	A primitive boxed value constructed and then immediately converted into a different primitive type (e.g., `new Double(d).intValue()`). Just perform direct primitive coercion (e.g., `(int) d`). This rule is deprecated, use {rule:squid:S2153} instead.
31	Correctness - Check for sign of bitwise operation	CRITICAL	This method compares an expression such as `((event.detail & SWT.SELECTED) > 0)` . Using bit arithmetic and then comparing with the greater than operator can lead to unexpected results (of course depending on the value of SWT.SELECTED). If SWT.SELECTED is a negative number, this is a candidate for a bug. Even when SWT.SELECTED is not negative, it seems good practice to use '!= 0' instead of '> 0'. *Boris Bokowski*
32	Dodgy - Method checks to see if result of String.indexOf is positive	MINOR	The method invokes String.indexOf and checks to see if the result is positive or non-positive. It is much more typical to check to see if the result is negative or non-negative. It is positive only if the substring checked for occurs at some place other than at the beginning of the String.
33	Dodgy - Invocation of substring(0), which returns the original value	CRITICAL	This code invokes substring(0) on a String, which returns the original value.

34	Multithreaded correctness - Monitor wait() called on Condition	MAJOR	This method calls `wait()` on a `java.util.concurrent.locks.Condition` object. Waiting for a `Condition` should be done using one of the `await()` methods defined by the `Condition` interface. This rule is deprecated, use {rule:squid:S1844} instead.
35	Dodgy - Load of known null value	CRITICAL	The variable referenced at this point is known to be null due to an earlier check against null. Although this is valid, it might be a mistake (perhaps you intended to refer to a different variable, or perhaps the earlier check to see if the variable is null should have been a check to see if it was nonnull).
36	Correctness - equals method overrides equals in superclass and may not be symmetric	MAJOR	This class defines an equals method that overrides an equals method in a superclass. Both equals methods methods use `instanceof` in the determination of whether two objects are equal. This is fraught with peril, since it is important that the equals method is symmetrical (in other words, `a.equals(b) == b.equals(a)`). If B is a subtype of A, and A's equals method checks that the argument is an instanceof A, and B's equals method checks that the argument is an instanceof B, it is quite likely that the equivalence relation defined by these methods is not symmetric. This rule is deprecated, use {rule:squid:S2162} instead.
37	Correctness - Call to equals() comparing different interface types	CRITICAL	This method calls equals(Object) on two references of unrelated interface types, where neither is a subtype of the other, and there are no known non-abstract classes which implement both interfaces. Therefore, the objects being compared are unlikely to be members of the same class at runtime (unless some application classes were not analyzed, or dynamic class loading can occur at runtime). According to the contract of equals(), objects of different classes should always compare as unequal; therefore, according to the contract defined by java.lang.Object.equals(Object), the result of this comparison will always be false at runtime.
38	Performance - Method invokes inefficient floating-point Number constructor; use static valueOf instead	MAJOR	Using `new Double(double)` is guaranteed to always result in a new object whereas `Double.valueOf(double)` allows caching of values to be done by the compiler, class library, or JVM. Using of cached values avoids object allocation and the code will be faster. Unless the class must be compatible with JVMs predating Java 1.5, use either autoboxing or the `valueOf()` method when creating instances of `Double` and `Float`.
			This class implements the `Comparator` interface. You should consider whether or not it should also implement the `Serializable` interface. If a comparator is used to construct an ordered collection such as a `TreeMap`, then the `TreeMap` will be serializable only if the comparator is also serializable. As most comparators have little or no state, making them serializable is generally easy and good defensive programming.

39	Bad practice - Comparator doesn't implement Serializable	MAJOR	This rule is deprecated, use {rule:squid:S2063} instead.
40	Bad practice - Method might ignore exception	MAJOR	This method might ignore an exception. In general, exceptions should be handled or reported in some way, or they should be thrown out of the method. This rule is deprecated, use {rule:squid:S00108} instead.
41	Correctness - Bad constant value for month	CRITICAL	This code passes a constant month value outside the expected range of 0..11 to a method. This rule is deprecated, use {rule:squid:S2110} instead.
42	Bad practice - Finalizer does not call superclass finalizer	MAJOR	This `finalize()` method does not make a call to its superclass's `finalize()` method. So, any finalizer actions defined for the superclass will not be performed. Add a call to `super.finalize()`. This rule is deprecated, use {rule:squid:ObjectFinalizeOverridenCallsSuperFinalizeCheck} instead.
43	Correctness - TestCase defines setUp that doesn't call super.setUp()	CRITICAL	Class is a JUnit TestCase and implements the setUp method. The setUp method should call super.setUp(), but doesn't.
44	Correctness - Method attempts to access a prepared statement parameter with index 0	CRITICAL	A call to a setXXX method of a prepared statement was made where the parameter index is 0. As parameter indexes start at index 1, this is always a mistake.
45	Dodgy - Unusual equals method	MINOR	This class doesn't do any of the patterns we recognize for checking that the type of the argument is compatible with the type of the `this` object. There might not be anything wrong with this code, but it is worth reviewing.
46	Dodgy - Transient field of class that isn't Serializable.	MAJOR	The field is marked as transient, but the class isn't Serializable, so marking it as transient has absolutely no effect. This may be leftover marking from a previous version of the code in which the class was transient, or it may indicate a misunderstanding of how serialization works. This rule is deprecated, use {rule:squid:S2065} instead.

47	Correctness - Use of class without a hashCode() method in a hashed data structure	CRITICAL	A class defines an equals(Object) method but not a hashCode() method, and thus doesn't fulfill the requirement that equal objects have equal hashCodes. An instance of this class is used in a hash data structure, making the need to fix this problem of highest importance.
48	Correctness - Value required to have type qualifier, but marked as unknown	CRITICAL	A value is used in a way that requires it to be always be a value denoted by a type qualifier, but there is an explicit annotation stating that it is not known where the value is required to have that type qualifier. Either the usage or the annotation is incorrect.
49	Bad comparison of int value with long constant	MAJOR	This code compares an int value with a long constant that is outside the range of values that can be represented as an int value. This comparison is vacuous and possibily to be incorrect.
50	Bad practice - Fields of immutable classes should be final	MINOR	The class is annotated with net.jcip.annotations.Immutable, and the rules for that annotation require that all fields are final. .
51	Dodgy - Redundant nullcheck of value known to be non-null	CRITICAL	This method contains a redundant check of a known non-null value against the constant null.
52	Correctness - instanceof will always return false	CRITICAL	This instanceof test will always return false. Although this is safe, make sure it isn't an indication of some misunderstanding or some other logic error. This rule is deprecated, use {rule:squid:S1850} instead.
53	Bad practice - serialVersionUID isn't final	CRITICAL	This class defines a `serialVersionUID` field that is not final. The field should be made final if it is intended to specify the version UID for purposes of serialization. This rule is deprecated, use {rule:squid:S2057} instead.
54	Malicious code vulnerability - May expose internal representation by incorporating reference to mutable object	MAJOR	This code stores a reference to an externally mutable object into the internal representation of the object. If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different. Storing a copy of the object is better approach in many situations.
55	Dodgy - Remainder of 32-bit signed random integer	CRITICAL	This code generates a random signed integer and then computes the remainder of that value modulo another value. Since the random number can be negative, the result of the remainder operation can also be negative. Be sure this is intended, and strongly consider using the Random.nextInt(int) method instead.
			The code calls `putNextEntry()`, immediately followed by a call to `closeEntry()`. This results in an empty JarFile entry. The contents of the entry

56	Bad practice - Creates an empty jar file entry	MAJOR	should be written to the JarFile between the calls to `putNextEntry()` and `closeEntry()` .
57	Correctness - Method call passes null for nonnull parameter	CRITICAL	This method call passes a null value for a nonnull method parameter. Either the parameter is annotated as a parameter that should always be nonnull, or analysis has shown that it will always be dereferenced.
58	Method ignores return value, is this OK?	MINOR	This code calls a method and ignores the return value. The return value is the same type as the type the method is invoked on, and from our analysis it looks like the return value might be important (e.g., like ignoring the return value of `String.toLowerCase()`). We are guessing that ignoring the return value might be a bad idea just from a simple analysis of the body of the method. You can use a `@CheckReturnValue` annotation to instruct FindBugs as to whether ignoring the return value of this method is important or acceptable. Please investigate this closely to decide whether it is OK to ignore the return value. This rule is deprecated, use {rule:squid:S2201} instead.
59	Performance - Method invokes inefficient new String(String) constructor	MAJOR	Using the `java.lang.String(String)` constructor wastes memory because the object so constructed will be functionally indistinguishable from the `String` passed as a parameter. Just use the argument `String` directly.
60	Bad practice - Class is Serializable, but doesn't define serialVersionUID	MAJOR	This class implements the `Serializable` interface, but does not define a `serialVersionUID` field. A change as simple as adding a reference to a .class object will add synthetic fields to the class, which will unfortunately change the implicit serialVersionUID (e.g., adding a reference to `String.class` will generate a static field `class$java$lang$String`). Also, different source code to bytecode compilers may use different naming conventions for synthetic variables generated for references to class objects or inner classes. To ensure interoperability of Serializable across versions, consider adding an explicit serialVersionUID. This rule is deprecated, use {rule:squid:S2057} instead.
61	Bad practice - Non-serializable value stored into instance field of a serializable class	CRITICAL	A non-serializable value is stored into a non-transient field of a serializable class.
62	Bad practice - Class is Externalizable but doesn't define a void constructor	MAJOR	This class implements the `Externalizable` interface, but does not define a void constructor. When Externalizable objects are deserialized, they first need to be constructed by invoking the void constructor. Since this class does not have one, serialization and deserialization will fail at runtime.

63	Correctness - Using pointer equality to compare different types	CRITICAL	This method uses using pointer equality to compare two references that seem to be of different types. The result of this comparison will always be false at runtime. This rule is deprecated, use {rule:squid:S1698} instead.
64	Unused public or protected field	INFO	This field is never used. The field is public or protected, so perhaps it is intended to be used with classes not seen as part of the analysis. If not, consider removing it from the class.
65	Correctness - Method attempts to access a result set field with index 0	CRITICAL	A call to getXXX or updateXXX methods of a result set was made where the field index is 0. As ResultSet fields start at index 1, this is always a mistake.
66	Performance - Primitive value is boxed and then immediately unboxed	MAJOR	A primitive is boxed, and then immediately unboxed. This probably is due to a manual boxing in a place where an unboxed value is required, thus forcing the compiler to immediately undo the work of the boxing. This rule is deprecated, use {rule:squid:S2153} instead.
67	Correctness - JUnit assertion in run method will not be noticed by JUnit	CRITICAL	A JUnit assertion is performed in a run method. Failed JUnit assertions just result in exceptions being thrown. Thus, if this exception occurs in a thread other than the thread that invokes the test method, the exception will terminate the thread but not result in the test failing.
68	Correctness - Array formatted in useless way using format string	MAJOR	One of the arguments being formatted with a format string is an array. This will be formatted using a fairly useless format, such as [I@304282, which doesn't actually show the contents of the array. Consider wrapping the array using `Arrays.asList(...)` before handling it off to a formatted. This rule is deprecated, use {rule:squid:S2275} instead.
69	Bad practice - Usage of GetResource may be unsafe if class is extended	MAJOR	Calling `this.getClass().getResource(...)` could give results other than expected if this class is extended by a class in another package.
70	Bad practice - Static initializer creates instance before all static final fields assigned	CRITICAL	The class's static initializer creates an instance of the class before all of the static final fields are assigned.
71	Correctness - Repeated conditional tests	MAJOR	The code contains a conditional test is performed twice, one right after the other (e.g., `x == 0`
72	Multithreaded correctness - Synchronize and null check on	MAJOR	Since the field is synchronized on, it seems not likely to be null. If it is null and then synchronized on a NullPointerException will be thrown and the check would be pointless. Better to synchronize on another field.

	the same field.		
73	Correctness - Useless assignment in return statement	CRITICAL	This statement assigns to a local variable in a return statement. This assignment has effect. Please verify that this statement does the right thing. This rule is deprecated, use {rule:squid:AssignmentInSubExpressionCheck} instead.
74	Bad practice - Class inherits equals() and uses Object.hashCode()	CRITICAL	This class inherits `equals(Object)` from an abstract superclass, and `hashCode()` from `java.lang.Object` (which returns the identity hash code, an arbitrary value assigned to the object by the VM). Therefore, the class is very likely to violate the invariant that equal objects must have equal hashcodes. If you don't want to define a hashCode method, and/or don't believe the object will ever be put into a HashMap/Hashtable, define the `hashCode()` method to throw `UnsupportedOperationException`.
75	Correctness - File.separator used for regular expression	CRITICAL	The code here uses `File.separator` where a regular expression is required. This will fail on Windows platforms, where the `File.separator` is a backslash, which is interpreted in a regular expression as an escape character. Amoung other options, you can just use `File.separatorChar=='\' & "\\" : File.separator` instead of `File.separator`
76	Multithreaded correctness - Wait not in loop	CRITICAL	This method contains a call to `java.lang.Object.wait()` which is not in a loop. If the monitor is used for multiple conditions, the condition the caller intended to wait for might not be the one that actually occurred. This rule is deprecated, use {rule:squid:S2274} instead.
77	Bad practice - Method may fail to close stream on exception	CRITICAL	The method creates an IO stream object, does not assign it to any fields, pass it to other methods, or return it, and does not appear to close it on all possible exception paths out of the method. This may result in a file descriptor leak. It is generally a good idea to use a `finally` block to ensure that streams are closed.
78	Correctness - Can't use reflection to check for presence of annotation without runtime retention	MAJOR	Unless an annotation has itself been annotated with @Retention(RetentionPolicy.RUNTIME), the annotation can't be observe using reflection (e.g., by using the isAnnotationPresent method). .

			This rule is deprecated, use {rule:squid:S2109} instead.
79	Correctness - Null pointer dereference	CRITICAL	A null pointer is dereferenced here. This will lead to a `NullPointerException` when the code is executed.
80	Multithreaded correctness - Static Calendar	CRITICAL	Even though the JavaDoc does not contain a hint about it, Calendars are inherently unsafe for multihtreaded use. Sharing a single instance across thread boundaries without proper synchronization will result in erratic behavior of the application. Under 1.4 problems seem to surface less often than under Java 5 where you will probably see random ArrayIndexOutOfBoundsExceptions or IndexOutOfBoundsExceptions in sun.util.calendar.BaseCalendar.getCalendar You may also experience serialization problems. Using an instance field is recommended. For more information on this see Sun Bug #6231579 and Sun Bug #6178997.
81	Performance - Unused field	MAJOR	This field is never used. Consider removing it from the class. This rule is deprecated, use {rule:squid:S1068} instead.
82	Correctness - Value is null and guaranteed to be dereferenced on exception path	CRITICAL	There is a statement or branch on an exception path that if executed guarantees that a value is null at this point, and that value that is guaranteed to be dereferenced (except on forward paths involving runtime exceptions).
83	Multithreaded correctness - Field not guarded against concurrent access	CRITICAL	This field is annotated with net.jcip.annotations.GuardedBy, but can be accessed in a way that seems to violate the annotation.
84	Bad practice - Clone method may return null	CRITICAL	This clone method seems to return null in some circumstances, but clone is never allowed to return a null value. If you are convinced this path is unreachable, throw an AssertionError instead.
85	Correctness - Bad comparison of signed byte	CRITICAL	Signed bytes can only have a value in the range -128 to 127. Comparing a signed byte with a value outside that range is vacuous and likely to be incorrect. To convert a signed byte `b` to an unsigned value in the range 0..255, use `0xff & b`
			The method in the subclass doesn't override a similar method in a superclass because the type of a parameter doesn't exactly match the type of the corresponding parameter in the superclass. For example, if you have:

86	Correctness - Method doesn't override method in superclass due to wrong package for parameter	MAJOR	`import alpha.Foo;`
			`public class A {`
			`  public int f(Foo x) { return 17; }`
			`}`
			`----`
			`import beta.Foo;`
			`public class B extends A {`
			`  public int f(Foo x) { return 42; }`
			`}`
			
			The `f(Foo)` method defined in class `B` doesn't override the `f(Foo)` method defined in class `A`, because the argument types are `Foo`'s from different packages.

87	Dodgy - Useless control flow	CRITICAL	This method contains a useless control flow statement, where control flow continues onto the same place regardless of whether or not the branch is taken. For example, this is caused by having an empty statement block for an `if` statement:
			
			`  if (argv.length == 0) {`
			`// TODO: handle this case`
			`}`

88	Correctness - Don't use removeAll to clear a collection	CRITICAL	If you want to remove all elements from a collection `c`, use `c.clear`, not `c.removeAll(c)`. This rule is deprecated, use {rule:squid:S2114} instead.
89	Bad practice - equals() method does not check for null argument	CRITICAL	This implementation of equals(Object) violates the contract defined by java.lang.Object.equals() because it does not check for null being passed as the argument. All equals() methods should return false if passed a null value.
90	Security - A prepared statement is generated from a nonconstant String	CRITICAL	The code creates an SQL prepared statement from a nonconstant String. If unchecked, tainted data from a user is used in building this String, SQL injection could be used to make the prepared statement do something unexpected and undesirable. This rule is deprecated, use {rule:squid:S2077} instead.
			The code performs an integer shift by a constant amount outside the range 0..31.

91	Correctness - Integer shift by an amount not in the range 0..31	CRITICAL	The effect of this is to use the lower 5 bits of the integer value to decide how much to shift by. This probably isn't want was expected, and it at least confusing. This rule is deprecated, use {rule:squid:S2183} instead.

92	Dodgy - int division result cast to double or float	CRITICAL	This code casts the result of an integer division operation to double or float. Doing division on integers truncates the result to the integer value closest to zero. The fact that the result was cast to double suggests that this precision should have been retained. What was probably meant was to cast one or both of the operands to double *before* performing the division. Here is an example:
			`int x = 2;`
			`int y = 5;`
			`// Wrong: yields result 0.0`
			`double value1 =  x / y;`
			
			`// Right: yields result 0.4`
			`double value2 =  x / (double) y;`
			This rule is deprecated, use {rule:squid:S2184} instead.

93	Bad practice - The readResolve method must be declared with a return type of Object.	MAJOR	In order for the readResolve method to be recognized by the serialization mechanism, it must be declared to have a return type of Object. This rule is deprecated, use {rule:squid:S2061} instead.
94	Correctness - Class overrides a method implemented in super class Adapter wrongly	CRITICAL	This method overrides a method found in a parent class, where that class is an Adapter that implements a listener defined in the java.awt.event or javax.swing.event package. As a result, this method will not get called when the event occurs.
95	Correctness - Unnecessary type check done using instanceof operator	CRITICAL	Type check performed using the instanceof operator where it can be statically determined whether the object is of the type requested.
96	Performance - Inefficient use of keySet iterator instead of entrySet iterator	CRITICAL	This method accesses the value of a Map entry, using a key that was retrieved from a keySet iterator. It is more efficient to use an iterator on the entrySet of the map, to avoid the Map.get(key) lookup.
			This method contains a useless control flow statement in which control flow follows to the same or following line regardless of whether or not

97	Correctness - Useless control flow to next line	CRITICAL	the branch is taken.
			Often, this is caused by inadvertently using an empty statement as the body of an `if` statement, e.g.:
			`if (argv.length == 1);`
			`    System.out.println("Hello, " + argv[0]);`

98	Malicious code vulnerability - May expose internal static state by storing a mutable object into a static field	MAJOR	This code stores a reference to an externally mutable object into a static field. If unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different. Storing a copy of the object is better approach in many situations.
99	Correctness - Null value is guaranteed to be dereferenced	BLOCKER	There is a statement or branch that if executed guarantees that a value is null at this point, and that value that is guaranteed to be dereferenced (except on forward paths involving runtime exceptions).
100	Multithreaded correctness - Condition.await() not in loop	CRITICAL	This method contains a call to `java.util.concurrent.await()` (or variants) which is not in a loop. If the object is used for multiple conditions, the condition the caller intended to wait for might not be the one that actually occurred. This rule is deprecated, use {rule:squid:S2274} instead.
101	Bad practice - Use of identifier that is a keyword in later versions of Java	MAJOR	This identifier is used as a keyword in later versions of Java. This code, and any code that references this API, will need to be changed in order to compile it in later versions of Java. This rule is deprecated, use {rule:squid:S1190} instead.
102	Multithreaded correctness - Constructor invokes Thread.start()	CRITICAL	The constructor starts a thread. This is likely to be wrong if the class is ever extended/subclassed, since the thread will be started before the subclass constructor is started.
103	Correctness - Invocation of hashCode on an array	CRITICAL	The code invokes hashCode on an array. Calling hashCode on an array returns the same value as System.identityHashCode, and ingores the contents and length of the array. If you need a hashCode that depends on the contents of an array `a`, use `java.util.Arrays.hashCode(a)`. This rule is deprecated, use {rule:squid:S2116} instead.

<table>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td>104</td><td>Correctness - Uninitialized read of field method called from constructor of superclass</td><td>MAJOR</td><td>This method is invoked in the constructor of of the superclass. At this point, the fields of the class have not yet initialized.
To make this more concrete, consider the following classes:
<pre>
abstract class A {
  int hashCode;
  abstract Object getValue();
  A() {
    hashCode = getValue().hashCode();
  }
}
class B extends A {
  Object value;
  B(Object v) {
    this.value = v;
  }
  Object getValue() {
    return value;
  }
}
</pre>
When a B is constructed, the constructor for the A class is invoked before the constructor for B sets value. Thus, when the constructor for A invokes getValue, an uninitialized value is read for value.</td></tr>
<tr><td>105</td><td>Correctness - Nonsensical self computation involving a variable (e.g., x & x)</td><td>CRITICAL</td><td>This method performs a nonsensical computation of a local variable with another
reference to the same variable (e.g., x&x or x-x). Because of the nature
of the computation, this operation doesn't seem to make sense,
and may indicate a typo or
a logic error. Double check the computation.

This rule is deprecated, use {rule:squid:S1764} instead.</td></tr>
<tr><td>106</td><td>Correctness - Illegal format string</td><td>CRITICAL</td><td>The format string is syntactically invalid,
and a runtime exception will occur when
this statement is executed.

This rule is deprecated, use {rule:squid:S2275} instead.</td></tr>
<tr><td>107</td><td>Correctness - Method call passes null for nonnull parameter (ALL<i>TARGETS</i>DANGEROUS)</td><td>CRITICAL</td><td>A possibly-null value is passed at a call site where all known
target methods require the parameter to be nonnull.
Either the parameter is annotated as a parameter that should
always be nonnull, or analysis has shown that it will always be
dereferenced.</td></tr>
<tr><td>108</td><td>Correctness - The readResolve method must not</td><td>MAJOR</td><td>In order for the readResolve method to be recognized by the serialization
mechanism, it must not be declared as a static method.</td></tr>
</table>

	be declared as a static method.		This rule is deprecated, use {rule:squid:S2061} instead.
109	Dodgy - Class implements same interface as superclass	MAJOR	This class declares that it implements an interface that is also implemented by a superclass. This is redundant because once a superclass implements an interface, all subclasses by default also implement this interface. It may point out that the inheritance hierarchy has changed since this class was created, and consideration should be given to the ownership of the interface's implementation.
110	Dodgy - Class extends Servlet class and uses instance variables	CRITICAL	This class extends from a Servlet class, and uses an instance member variable. Since only one instance of a Servlet class is created by the J2EE framework, and used in a multithreaded way, this paradigm is highly discouraged and most likely problematic. Consider only using method local variables. This rule is deprecated, use {rule:squid:S2226} instead.
111	Bad practice - Finalizer only nulls fields	MAJOR	This finalizer does nothing except null out fields. This is completely pointless, and requires that the object be garbage collected, finalized, and then garbage collected again. You should just remove the finalize method. This rule is deprecated, use {rule:squid:S2165} instead.
112	Correctness - No previous argument for format string	CRITICAL	The format string specifies a relative index to request that the argument for the previous format specifier be reused. However, there is no previous argument. For example, `formatter.format("%<s %s", "a", "b")` would throw a MissingFormatArgumentException when executed. This rule is deprecated, use {rule:squid:S2275} instead.
113	Dodgy - Call to unsupported method	MAJOR	All targets of this method invocation throw an UnsupportedOperationException.
114	Correctness - Random value from 0 to 1 is coerced to the integer 0	MAJOR	A random value from 0 to 1 is being coerced to the integer value 0. You probably want to multiple the random value by something else before coercing it to an integer, or use the `Random.nextInt(n)` method.
			An inner class is invoking a method that could be resolved to either a inherited method or a method defined in an outer class. By the Java semantics, it will be resolved to invoke the inherited method, but this may not be want you intend. If you really intend to invoke the inherited method, invoke it by invoking the method on super (e.g., invoke super.foo(17)), and

115	Dodgy - Ambiguous invocation of either an inherited or outer method	MAJOR	thus it will be clear to other readers of your code and to FindBugs that you want to invoke the inherited method, not the method in the outer class. This rule is deprecated, use {rule:squid:S2388} instead.
116	Correctness - Check to see if ((...) & 0) == 0	CRITICAL	This method compares an expression of the form (e & 0) to 0, which will always compare equal. This may indicate a logic error or typo.
117	Performance - Method invokes inefficient new String() constructor	MAJOR	Creating a new `java.lang.String` object using the no-argument constructor wastes memory because the object so created will be functionally indistinguishable from the empty string constant `""`. Java guarantees that identical string constants will be represented by the same `String` object. Therefore, you should just use the empty string constant directly.

118	Multithreaded correctness - Method does not release lock on all paths	CRITICAL	This method acquires a JSR-166 (`java.util.concurrent`) lock,
			but does not release it on all paths out of the method. In general, the correct idiom
			for using a JSR-166 lock is:
			
			`Lock l = ...;`
			`l.lock();`
			`try {`
			`    // do something`
			`} finally {`
			`    l.unlock();`
			`}`

119	Security - HTTP Response splitting vulnerability	MAJOR	This code directly writes an HTTP parameter to an HTTP header, which allows for a HTTP response splitting vulnerability. See http://en.wikipedia.org/wiki/HTTP*response*splitting for more information. FindBugs looks only for the most blatant, obvious cases of HTTP response splitting. If FindBugs found *any*, you *almost certainly* have more vulnerabilities that FindBugs doesn't report. If you are concerned about HTTP response splitting, you should seriously consider using a commercial static analysis or pen-testing tool.
120	Bad practice - Class defines clone() but doesn't implement Cloneable	MAJOR	This class defines a clone() method but the class doesn't implement Cloneable. There are some situations in which this is OK (e.g., you want to control how subclasses can clone themselves), but just make sure that this is what you intended. This rule is deprecated, use {rule:squid:S1182} instead.
			The code synchronizes on interned String.

121	Multithreaded correctness - Synchronization on interned String could lead to deadlock	CRITICAL	`private static String LOCK = "LOCK";`
			`...`
			`  synchronized(LOCK) { ...}`
			`...`
			Constant Strings are interned and shared across all other classes loaded by the JVM. Thus, this could is locking on something that other code might also be locking. This could result in very strange and hard to diagnose blocking and deadlock behavior. See http://www.javalobby.org/java/forums/t96352.html and http://jira.codehaus.org/browse/JETTY-352.
			This rule is deprecated, use {rule:squid:S1860} instead.

122	Correctness - Method defines a variable that obscures a field	MAJOR	This method defines a local variable with the same name as a field in this class or a superclass. This may cause the method to read an uninitialized value from the field, leave the field uninitialized, or both.
123	Unwritten public or protected field	MINOR	No writes were seen to this public/protected field. All reads of it will return the default value. Check for errors (should it have been initialized?), or remove it if it is useless.
124	Correctness - Self comparison of field with itself	CRITICAL	This method compares a field with itself, and may indicate a typo or a logic error. Make sure that you are comparing the right things. This rule is deprecated, use {rule:squid:S1764} instead.
125	Bad practice - Serializable inner class	MAJOR	This Serializable class is an inner class. Any attempt to serialize it will also serialize the associated outer instance. The outer instance is serializable, so this won't fail, but it might serialize a lot more data than intended. If possible, making the inner class a static inner class (also known as a nested class) should solve the problem. This rule is deprecated, use {rule:squid:S2059} instead.
126	Dodgy - Class is final but declares protected field	MINOR	This class is declared to be final, but declares fields to be protected. Since the class is final, it can not be derived from, and the use of protected is confusing. The access modifier for the field should be changed to private or public to represent the true use for the field. This rule is deprecated, use {rule:squid:S2156} instead.
			This field is never written. All reads of it will return the default value. Check for errors (should it have been initialized?), or remove it if it is useless.

127	Correctness - Unwritten field	MINOR	This rule is deprecated, use {rule:squid:S1068} instead.
128	Bad practice - Finalizer does nothing but call superclass finalizer	MINOR	The only thing this `finalize()` method does is call the superclass's `finalize()` method, making it redundant. Delete it. This rule is deprecated, use {rule:squid:S1185} instead.
129	Correctness - Apparent method/constructor confusion	MAJOR	This regular method has the same name as the class it is defined in. It is likely that this was intended to be a constructor. If it was intended to be a constructor, remove the declaration of a void return value. If you had accidently defined this method, realized the mistake, defined a proper constructor but can't get rid of this method due to backwards compatibility, deprecate the method. This rule is deprecated, use {rule:squid:S1223} instead.
130	Multithreaded correctness - Method synchronizes on an updated field	MAJOR	This method synchronizes on an object referenced from a mutable field. This is unlikely to have useful semantics, since different threads may be synchronizing on different objects. This rule is deprecated, use {rule:squid:S2445} instead.
131	Correctness - Method performs math using floating point precision	CRITICAL	The method performs math operations using floating point precision. Floating point precision is very imprecise. For example, 16777216.0f + 1.0f = 16777216.0f. Consider using double math instead. This rule is deprecated, use {rule:squid:S2164} instead.
132	Correctness - MessageFormat supplied where printf style format expected	MAJOR	A method is called that expects a Java printf format string and a list of arguments. However, the format string doesn't contain any format specifiers (e.g., %s) but does contain message format elements (e.g., {0}). It is likely that the code is supplying a format string is required. At runtime, all of the arguments will be ignored and the format string will be returned exactly as prov This rule is deprecated, use {rule:squid:S2275} instead.
	Dodgy - Non serializable object		This code seems to be passing a non-serializable object to the ObjectOutput.writeObject method. If the object is, indeed, non-serializable, an error will result.

133	written to ObjectOutput	CRITICAL	This rule is deprecated, use {rule:squid:S2118} instead.
134	Dodgy - Class extends Struts Action class and uses instance variables	CRITICAL	This class extends from a Struts Action class, and uses an instance member variable. Since only one instance of a struts Action class is created by the Struts framework, and used in a multithreaded way, this paradigm is highly discouraged and most likely problematic. Consider only using method local variables. Only instance fields that are written outside of a monitor are reported. This rule is deprecated, use {rule:squid:S2226} instead.
135	Correctness - equals() method defined that doesn't override Object.equals(Object)	MAJOR	This class defines an `equals()` method, that doesn't override the normal `equals(Object)` method defined in the base `java.lang.Object` class. The class should probably define a `boolean equals(Object)` method. This rule is deprecated, use {rule:squid:S1201} instead.
136	Dodgy - Redundant comparison of non-null value to null	CRITICAL	This method contains a reference known to be non-null with another reference known to be null.
137	Format string should use %n rather than \n	MAJOR	This format string include a newline character (\n). In format strings, it is generally preferable better to use %n, which will produce the platform-specific line separator. This rule is deprecated, use {rule:squid:S2275} instead.
138	Malicious code vulnerability - Field isn't final and can't be protected from malicious code	MAJOR	A mutable static field could be changed by malicious code or by accident from another package. Unfortunately, the way the field is used doesn't allow any easy fix to this problem. This rule is deprecated, use {rule:squid:S1444} instead.
139	Correctness - Store of null value into field annotated NonNull	CRITICAL	A value that could be null is stored into a field that has been annotated as NonNull.
	Correctness - Invocation of		The code invokes toString on an (anonymous) array. Calling toString on an array generates a fairly useless result such as [C@16f0472. Consider using Arrays.toString to convert the array into a readable String that gives the contents of the array. See Programming Puzzlers, chapter 3, puzzle 12.

140	toString on an anonymous array	CRITICAL	This rule is deprecated, use {rule:squid:S2116} instead.
141	Dodgy - Check for oddness that won't work for negative numbers	CRITICAL	The code uses x % 2 == 1 to check to see if a value is odd, but this won't work for negative numbers (e.g., (-5) % 2 == -1). If this code is intending to check for oddness, consider using x & 1 == 1, or x % 2 != 0. This rule is deprecated, use {rule:squid:S2197} instead.
142	Multithreaded correctness - Method spins on field	MAJOR	This method spins in a loop which reads a field. The compiler may legally hoist the read out of the loop, turning the code into an infinite loop. The class should be changed so it uses proper synchronization (including wait and notify calls).
143	Multithreaded correctness - Inconsistent synchronization	CRITICAL	The fields of this class appear to be accessed inconsistently with respect to synchronization. This bug report indicates that the bug pattern detector judged that 1. The class contains a mix of locked and unlocked accesses, 2. At least one locked access was performed by one of the class's own methods, and 3. The number of unsynchronized field accesses (reads and writes) was no more than one third of all accesses, with writes being weighed twice as high as reads A typical bug matching this bug pattern is forgetting to synchronize one of the methods in a class that is intended to be thread-safe. You can select the nodes labeled "Unsynchronized access" to show the code locations where the detector believed that a field was accessed without synchronization. Note that there are various sources of inaccuracy in this detector; for example, the detector cannot statically detect all situations in which a lock is held. Also, even when the detector is accurate in distinguishing locked vs. unlocked accesses, the code in question may still be correct.
144	Correctness - int value cast to double and then passed to	CRITICAL	This code converts an int value to a double precision floating point number and then passing the result to the Math.ceil() function, which rounds a double to the next higher integer value. This operation should always be a no-op, since the converting an integer to a double should give a number with no fractional part.

<table>
<tr><th></th><th></th><th></th><th></th></tr>
<tr><td></td><td>Math.ceil</td><td></td><td>It is likely that the operation that generated the value to be passed
to Math.ceil was intended to be performed using double precision
floating point arithmetic.</td></tr>
<tr><td>145</td><td>Bad practice - Method may fail to close database resource on exception</td><td>CRITICAL</td><td>The method creates a database resource (such as a database connection
or row set), does not assign it to any
fields, pass it to other methods, or return it, and does not appear to close
the object on all exception paths out of the method. Failure to
close database resources on all paths out of a method may
result in poor performance, and could cause the application to
have problems communicating with the database.</td></tr>
<tr><td>146</td><td>Bad practice - Class implements Cloneable but does not define or use clone method</td><td>MAJOR</td><td>Class implements Cloneable but does not define or
use the clone method.

This rule is deprecated, use {rule:squid:S2157} instead.</td></tr>
<tr><td>147</td><td>Correctness - Creation of ScheduledThreadPoolExecutor with zero core threads</td><td>MINOR</td><td>(Javadoc)
A ScheduledThreadPoolExecutor with zero core threads will never execute anything; changes to the max pool size are ignor

This rule is deprecated, use {rule:squid:S2122} instead.</td></tr>
<tr><td>148</td><td>Bad practice - Unchecked type in generic call</td><td>CRITICAL</td><td>This call to a generic collection method passes an argument
while compile type Object where a specific type from
the generic type parameters is expected.
Thus, neither the standard Java type system nor static analysis
can provide useful information on whether the
object being passed as a parameter is of an appropriate type.

This rule is deprecated, use {rule:squid:S2175} instead.</td></tr>
<tr><td>149</td><td>Multithreaded correctness - Synchronization on Boolean could lead to deadlock</td><td>CRITICAL</td><td>The code synchronizes on a boxed primitive constant, such as an Boolean.
<pre>
private static Boolean inited = Boolean.FALSE;
...
  synchronized(inited) {
    if (!inited) {
       init();
       inited = Boolean.TRUE;
       }
     }
...
</pre>
Since there normally exist only two Boolean objects, this code could be synchronizing on the same object as other, unrelated
code, leading to unresponsiveness
and possible deadlock</td></tr>
</table>

<table>
<tr><td></td><td></td><td></td><td>This rule is deprecated, use {rule:squid:S1860} instead.</td></tr>
<tr><td>150</td><td>Bad practice - Certain swing methods needs to be invoked in Swing thread</td><td>MAJOR</td><td>(From JDC Tech Tip): The Swing methods
show(), setVisible(), and pack() will create the associated peer for the frame.
With the creation of the peer, the system creates the event dispatch thread.
This makes things problematic because the event dispatch thread could be notifying listeners while pack and validate are still processing. This situation could result in two threads going through the Swing component-based GUI -- it's a serious flaw that could result in deadlocks or other related threading issues. A pack call causes components to be realized. As they are being realized (that is, not necessarily visible), they could trigger listener notification on the event dispatch thread.</td></tr>
<tr><td>151</td><td>Dead store due to switch statement fall through to throw</td><td>CRITICAL</td><td>A value stored in the previous switch case is ignored here due to a switch fall through to a place where an exception is thrown.
It is likely that you forgot to put a break or return at the end of the previous case.</td></tr>
<tr><td>152</td><td>Dodgy - Possible null pointer dereference due to return value of called method</td><td>CRITICAL</td><td>

The return value from a method is dereferenced without a null check,
and the return value of that method is one that should generally be checked for null (which requires to use Findbugs annotati
to express the developer's intend).
This may lead to a `NullPointerException` when the code is executed.

Noncompliant Code Example

```
public long getTime() {
  return getDate().getTime();      // NullPointerException may occur
}

@CheckForNull                       // See javax.annotation.CheckForNull (JSR-305)
public Date getDate() { /* ... / }
```

Compliant Solution

```
public long getTime() {
  Date date = getDate();
  if (date == null) {
    throw new IllegalStateException("...");
  }
  return date.getTime();
}

@CheckForNull                       // See javax.annotation.CheckForNull (JSR-305)
public Date getDate() { / ... */ }
```

</td></tr>
</table>

153	Reliance on default encoding	MAJOR	Found a call to a method which will perform a byte to String (or String to byte) conversion, and will assume that the default platform encoding is suitable. This will cause the application behaviour to vary between platforms. Use an alternative API and specify a charset name or Charset object explicitly. This rule is deprecated, use {rule:squid:S1943} instead.
154	Multithreaded correctness - Possible double check of field	MAJOR	This method may contain an instance of double-checked locking. This idiom is not correct according to the semantics of the Java memory model. For more information, see the web page >http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.
155	Correctness - Invocation of equals() on an array, which is equivalent to ==	CRITICAL	This method invokes the .equals(Object o) method on an array. Since arrays do not override the equals method of Object, calling equals on an array is the same as comparing their addresses. To compare the contents of the arrays, use java.util.Arrays.equals(Object[], Object[]). This rule is deprecated, use {rule:squid:S1294} instead.
156	Dodgy - Exception is caught when Exception is not thrown	MAJOR	This method uses a try-catch block that catches Exception objects, but Exception is not thrown within the try block, and RuntimeException is not explicitly caught. It is a common bug pattern to say try { ... } catch (Exception e) { something } as a shorthand for catching a number of types of exception each of whose catch blocks is identical, but this construct also accidentally catches RuntimeException as well, masking potential bugs.
157	Correctness - Value annotated as never carrying a type qualifier used where value carrying that qualifier is required	CRITICAL	A value specified as not carrying a type qualifier annotation is guaranteed to be consumed in a location or locations requiring that the value does carry that annotation. More precisely, a value annotated with a type qualifier specifying when=NEVER is guaranteed to reach a use or uses where the same type qualifier specifies when=ALWAYS. TODO: example
158	Correctness - Uninitialized read of field in constructor	MAJOR	This constructor reads a field which has not yet been assigned a value. This is often caused when the programmer mistakenly uses the field instead of one of the constructor's parameters.
			This call doesn't make sense. For any collection `c`, calling `c.containsAll(c)` should always be true, and `c.retainAll(c)` should have no effect.

159	Correctness - Vacuous call to collections	CRITICAL	This rule is deprecated, use {rule:squid:S2114} instead.
160	Correctness - Futile attempt to change max pool size of ScheduledThreadPoolExecutor	MINOR	(Javadoc) While ScheduledThreadPoolExecutor inherits from ThreadPoolExecutor, a few of the inherited tuning methods are not useful for it. In particular, because it acts as a fixed-sized pool using corePoolSize threads and an unbounded queue, adjustments t effect.
161	Correctness - Incompatible bit masks (BIT_AND)	CRITICAL	This method compares an expression of the form (e & C) to D, which will always compare unequal due to the specific values of constants C and D. This may indicate a logic error or typo.
162	Correctness - Collections should not contain themselves	CRITICAL	This call to a generic collection's method would only make sense if a collection contained itself (e.g., if `s.contains(s)` were true). This is unlikely to be true and would cause problems if it were true (such as the computation of the hash code resulting in infinite recursion). It is likely that the wrong value is being passed as a parameter. This rule is deprecated, use {rule:squid:S2114} instead.
163	Method may fail to clean up stream or resource on checked exception	CRITICAL	This method may fail to clean up (close, dispose of) a stream, database object, or other resource requiring an explicit cleanup operation. In general, if a method opens a stream or other resource, the method should use a try/finally block to ensure that the stream or resource is cleaned up before the method returns. This bug pattern is essentially the same as the OS*OPEN*STREAM and ODR*OPEN*DATABASE_RESOURCE bug patterns, b on a different (and hopefully better) static analysis technique. See Weimer and Necula, Finding and Preventing Run-Time Error Handling M for a description of the analysis technique. .
164	Correctness - Signature declares use of unhashable class in hashed construct	CRITICAL	A method, field or class declares a generic signature where a non-hashable class is used in context where a hashable class is required. A class that declares an equals method but inherits a hashCode() method from Object is unhashable, since it doesn't fulfill the requirement that equal objects have equal hashCodes.
165	Correctness - Call to equals() with null argument	CRITICAL	This method calls equals(Object), passing a null value as the argument. According to the contract of the equals() method, this call should always return `false`. This rule is deprecated, use {rule:squid:S1318} instead.
166	Correctness - Method may return null, but is declared	CRITICAL	This method may return a null value, but the method (or a superclass method which it overrides) is declared to return @NonNull.

	@NonNull		
167	Correctness - Primitive array passed to function expecting a variable number of object arguments	CRITICAL	This code passes a primitive array to a function that takes a variable number of object arguments. This creates an array of length one to hold the primitive array and passes it to the function.
168	Dodgy - Dead store to local variable	CRITICAL	This instruction assigns a value to a local variable, but the value is not read or used in any subsequent instruction. Often, this indicates an error, because the value computed is never used. Note that Sun's javac compiler often generates dead stores for final local variables. Because FindBugs is a bytecode-based tool, there is no easy way to eliminate these false positives. This rule is deprecated, use {rule:squid:S1481} instead.
169	Correctness - Bad attempt to compute absolute value of signed 32-bit random integer	CRITICAL	This code generates a random signed integer and then computes the absolute value of that random integer. If the number returned by the random number generator is `Integer.MINVALUE` *, then the result will be negative as well (since* `Math.abs(Integer.MIN` `VALUE) == Integer.MIN_VALUE`).
170	Dodgy - Test for floating point equality	CRITICAL	This operation compares two floating point values for equality. Because floating point calculations may involve rounding, calculated float and double values may not be accurate. For values that must be precise, such as monetary values, consider using a fixed-precision type such as BigDecimal. For values that need not be precise, consider comparing for equality within some range, for example: `if ( Math.abs(x - y) < .0000001 )`. See the Java Language Specification, section 4.2.4. This rule is deprecated, use {rule:squid:S1244} instead.
171	Nonnull field is not initialized	CRITICAL	The field is marked as nonnull, but isn't written to by the constructor. The field might be initialized elsewhere during constructor, or might always be initialized before use.
172	Bad practice - Class names shouldn't shadow simple name of implemented interface	MAJOR	This class/interface has a simple name that is identical to that of an implemented/extended interface, except that the interface is in a different package (e.g., `alpha.Foo` extends `beta.Foo`). This can be exceptionally confusing, create lots of situations in which you have to look at import statements to resolve references and creates many opportunities to accidently define methods that do not override methods in their superclasses. This rule is deprecated, use {rule:squid:S2176} instead.

173	Multithreaded correctness - Sychronization on getClass rather than class literal	CRITICAL	This instance method synchronizes on `this.getClass()`. If this class is subclassed, subclasses will synchronize on the class object for the subclass, which isn't likely what was intended. For example, consider this code from java.awt.Label:
			`private static final String base = "label";`
			`private static int nameCounter = 0;`
			`String constructComponentName() {`
			`   synchronized (getClass()) {`
			`       return base + nameCounter++;`
			`   }`
			`}`
			Subclasses of `Label` won't synchronize on the same subclass, giving rise to a datarace. Instead, this code should be synchronizing on `Label.class`
			`private static final String base = "label";`
			`private static int nameCounter = 0;`
			`String constructComponentName() {`
			`   synchronized (Label.class) {`
			`       return base + nameCounter++;`
			`   }`
			`}`
			Bug pattern contributed by Jason Mehrens

174	Security - Nonconstant string passed to execute method on an SQL statement	CRITICAL	The method invokes the execute method on an SQL statement with a String that seems to be dynamically generated. Consider using a prepared statement instead. It is more efficient and less vulnerable to SQL injection attacks. This rule is deprecated, use {rule:squid:S2077} instead.
175	Correctness - Invalid syntax for regular expression	CRITICAL	The code here uses a regular expression that is invalid according to the syntax for regular expressions. This statement will throw a PatternSyntaxException when executed.
176	Dodgy - Parameter must be nonnull but is marked as nullable	CRITICAL	This parameter is always used in a way that requires it to be nonnull, but the parameter is explicitly annotated as being Nullable. Either the use of the parameter or the annotation is wrong.
177	Multithreaded correctness - Call to static Calendar	CRITICAL	Even though the JavaDoc does not contain a hint about it, Calendars are inherently unsafe for multihtreaded use. The detector has found a call to an instance of Calendar that has been obtained via a static field. This looks suspicous. For more information on this see Sun Bug #6231579 and Sun Bug #6178997.

178	BigDecimal constructed from double that isn't represented precisely	MAJOR	This code creates a BigDecimal from a double value that doesn't translate well to a decimal number. For example, one might assume that writing `new BigDecimal(0.1)` in Java creates a BigDecimal which is exactly equal to 0.1 (an unscaled value of 1, with a scale of 1), but it is actually equal to 0.10000000000000000555111512312578270211815834 You probably want to use the `BigDecimal.valueOf(double d)` method, which uses the String representation of the double to create the BigDecimal (e.g., `BigDecimal.valueOf(0.1)` gives 0.1). This rule is deprecated, use {rule:squid:S2111} instead.
179	Correctness - Bad comparison of nonnegative value with negative constant	CRITICAL	This code compares a value that is guaranteed to be non-negative with a negative constant.
180	Multithreaded correctness - Inconsistent synchronization	MAJOR	The fields of this class appear to be accessed inconsistently with respect to synchronization. This bug report indicates that the bug pattern detector judged that 1. The class contains a mix of locked and unlocked accesses, 2. At least one locked access was performed by one of the class's own methods, and 3. The number of unsynchronized field accesses (reads and writes) was no more than one third of all accesses, with writes being weighed twice as high as reads A typical bug matching this bug pattern is forgetting to synchronize one of the methods in a class that is intended to be thread-safe. Note that there are various sources of inaccuracy in this detector; for example, the detector cannot statically detect all situations in which a lock is held. Also, even when the detector is accurate in distinguishing locked vs. unlocked accesses, the code in question may still be correct.

			OpenJDK introduces a potential incompatibility. In particular, the java.util.logging.Logger behavior has changed. Instead of using strong references, it now uses weak references internally. That's a reasonable change, but unfortunately some code relies on the old behavior - when changing logger configuration, it simply drops the logger reference. That means that the garbage collector is free to reclaim that memory, which means that the logger configuration is lost. For example, consider:
			`public static void initLogging() throws Exception {`
			`      Logger logger = Logger.getLogger("edu.umd.cs");`
			`      logger.addHandler(new FileHandler()); // call to change logger configuration`
			`      logger.setUseParentHandlers(false); // another call to change logger configuration`
			`      }`

181	Experimental - Potential lost logger changes due to weak reference in OpenJDK	INFO	The logger reference is lost at the end of the method (it doesn't escape the method), so if you have a garbage collection cycle just after the call to initLogging, the logger configuration is lost (because Logger only keeps weak references).
			`public static void main(String[] args) throws Exception {`
			`      initLogging(); // adds a file handler to the logger`
			`      System.gc(); // logger configuration lost`
			`      Logger.getLogger("edu.umd.cs").info("Some message"); // this isn't logged to the file as exp`
			`      }`
			Ulf Ochsenfahrt and Eric Fellheimer

182	Dodgy - Remainder of hashCode could be negative	CRITICAL	This code computes a hashCode, and then computes the remainder of that value modulo another value. Since the hashCode can be negative, the result of the remainder operation can also be negative. Assuming you want to ensure that the result of your computation is nonnegative, you may need to change your code. If you know the divisor is a power of 2, you can use a bitwise and operator instead (i.e., instead of using `x.hashCode()%n` , use `x.hashCode()&(n-1)` . This is probably faster than computing the remainder as well. If you don't know that the divisor is a power of 2, take the absolute value of the result of the remainder operation (i.e., use `Math.abs(x.hashCode()%n)` This rule is deprecated, use {rule:squid:S2197} instead.
183	Adding elements of an entry set may fail due to reuse of Entry objects	MAJOR	The entrySet() method is allowed to return a view of the underlying Map in which a single Entry object is reused and returned during the iteration. As of Java 1.6, both IdentityHashMap and EnumMap did so. When iterating through such a Map, the Entry value is only valid until you advance to the next iteration. If, for example, you try to pass such an entrySet to an addAll method, things will go badly wrong.
184	Correctness - int value cast to float and then passed to Math.round	CRITICAL	This code converts an int value to a float precision floating point number and then passing the result to the Math.round() function, which returns the int/long closest to the argument. This operation should always be a no-op, since the converting an integer to a float should give a number with no fractional part. It is likely that the operation that generated the value to be passed to Math.round was intended to be performed using floating point arithmetic.
185	Correctness - Bad attempt to compute absolute value of signed 32-bit hashcode	CRITICAL	This code generates a hashcode and then computes the absolute value of that hashcode. If the hashcode is `Integer.MINVALUE` *, then the result will be negative as well (since* `Math.abs(Integer.MIN` `VALUE) == Integer.MIN_VALUE`).

186	Correctness - hasNext method invokes next	CRITICAL	The hasNext() method invokes the next() method. This is almost certainly wrong, since the hasNext() method is not supposed to change the state of the iterator, and the next method is supposed to change the state of the iterator. This rule is deprecated, use {rule:squid:S1849} instead.
187	Correctness - Possible null pointer dereference	CRITICAL	There is a branch of statement that, *if executed,* guarantees that a null value will be dereferenced, which would generate a `NullPointerException` when the code is executed. Of course, the problem might be that the branch or statement is infeasible and that the null pointer exception can't ever be executed; deciding that is beyond the ability of FindBugs.
188	Multithreaded correctness - Class's readObject() method is synchronized	CRITICAL	This serializable class defines a `readObject()` which is synchronized. By definition, an object created by deserialization is only reachable by one thread, and thus there is no need for `readObject()` to be synchronized. If the `readObject()` method itself is causing the object to become visible to another thread, that is an example of very dubious coding style.
189	Correctness - equals(...) used to compare incompatible arrays	BLOCKER	This method invokes the .equals(Object o) to compare two arrays, but the arrays of of incompatible types (e.g., String[] and StringBuffer[], or String[] and int[]). They will never be equal. In addition, when equals(...) is used to compare arrays it on array, and ignores the contents of the arrays. This rule is deprecated, use {rule:squid:S1294} instead.
190	Dodgy - Unchecked/unconfirmed cast	CRITICAL	This cast is unchecked, and not all instances of the type casted from can be cast to the type it is being cast to. Ensure that your program logic ensures that this cast will not fail.
191	Bad practice - equals method fails for subtypes	CRITICAL	This class has an equals method that will be broken if it is inherited by subclasses. It compares a class literal with the class of the argument (e.g., in class `Foo` it might check if `Foo.class == o.getClass()`). It is better to check if `this.getClass() == o.getClass()` .

192	Correctness - Double assignment of field	CRITICAL	This method contains a double assignment of a field; e.g.
			`int x,y;`
			`public void foo() {`
			`  x = x = 17;`
			`}`
			Assigning to a field twice is useless, and may indicate a logic error or typo.
			This rule is deprecated, use {rule:squid:S1656} instead.

193	Correctness - Number of format-string arguments does not correspond to number of placeholders	CRITICAL	A format-string method with a variable number of arguments is called, but the number of arguments passed does not match with the number of % placeholders in the format string. This is probably not what the author intended. This rule is deprecated, use {rule:squid:S2275} instead.
194	Correctness - Primitive value is unboxed and coerced for ternary operator	MAJOR	A wrapped primitive value is unboxed and converted to another primitive type as part of the evaluation of a conditional ternary operator (the `b ? e1 : e2` operator). The semantics of Java mandate that if `e1` and `e2` are wrapped numeric values, the values are unboxed and converted/coerced to their common type (e.g, if `e1` is of type `Integer` and `e2` is of type `Float` , then `e1` is unboxed, converted to a floating point value, and boxed. See JLS Section 15.25. This rule is deprecated, use {rule:squid:S2154} instead.
195	Correctness - Double.longBitsToDouble invoked on an int	CRITICAL	The Double.longBitsToDouble method is invoked, but a 32 bit int value is passed as an argument. This almostly certainly is not intended and is unlikely to give the intended result. This rule is deprecated, use {rule:squid:S2127} instead.
196	Correctness - TestCase has no tests	CRITICAL	Class is a JUnit TestCase but has not implemented any test methods
197	Correctness - "." used for regular expression	CRITICAL	A String function is being invoked and "." is being passed to a parameter that takes a regular expression as an argument. Is this what you intended? For example s.replaceAll(".", "/") will return a String in which *every* character has been replaced by a / character.
198	Correctness - Self comparison of value with itself	CRITICAL	This method compares a local variable with itself, and may indicate a typo or a logic error. Make sure that you are comparing the right things. This rule is deprecated, use {rule:squid:S1764} instead.
			During the initialization of a class, the class makes an active use of a subclass. That subclass will not yet be initialized at the time of this use.

199	Bad practice - Superclass uses subclass during initialization	MAJOR	For example, in the following code, `foo` will be null.
			`public class CircularClassInitialization {`
			` static class InnerClassSingleton extends CircularClassInitialization {`
			`     static InnerClassSingleton singleton = new InnerClassSingleton();`
			`  }`
			
			`   static CircularClassInitialization foo = InnerClassSingleton.singleton;`
			`}`

200	Multithreaded correctness - Unsynchronized get method, synchronized set method	MAJOR	This class contains similarly-named get and set methods where the set method is synchronized and the get method is not. This may result in incorrect behavior at runtime, as callers of the get method will not necessarily see a consistent state for the object. The get method should be made synchronized.
201	Correctness - Impossible downcast	BLOCKER	This cast will always throw a ClassCastException. The analysis believes it knows the precise type of the value being cast, and the attempt to downcast it to a subtype will always fail by throwing a ClassCastException.
202	Correctness - Nonsensical self computation involving a field (e.g., x & x)	CRITICAL	This method performs a nonsensical computation of a field with another reference to the same field (e.g., x&x or x-x). Because of the nature of the computation, this operation doesn't seem to make sense, and may indicate a typo or a logic error. Double check the computation. This rule is deprecated, use {rule:squid:S1764} instead.
203	Multithreaded correctness - Using notify() rather than notifyAll()	CRITICAL	This method calls `notify()` rather than `notifyAll()`. Java monitors are often used for multiple conditions. Calling `notify()` only wakes up one thread, meaning that the thread woken up might not be the one waiting for the condition that the caller just satisfied. This rule is deprecated, use {rule:squid:S2446} instead.
204	Boxed value is unboxed and then immediately reboxed	MAJOR	A boxed value is unboxed and then immediately reboxed. This rule is deprecated, use {rule:squid:S2153} instead.
205	Dead store due to switch statement fall through	CRITICAL	A value stored in the previous switch case is overwritten here due to a switch fall through. It is likely that you forgot to put a break or return at the end of the previous case.
			This method contains an unsynchronized lazy initialization of a static field. After the field is set, the object stored into that location is further accessed. The setting of the field is visible to other threads as soon as it is set. If the futher accesses in the method that set the field serve to initialize the object, then

206	Multithreaded correctness - Incorrect lazy initialization and update of static field	CRITICAL	you have a *very serious* multithreading bug, unless something else prevents any other thread from accessing the stored object until it is fully initialized. This rule is deprecated, use {rule:squid:S2444} instead.
207	Bad practice - serialVersionUID isn't long	MAJOR	This class defines a `serialVersionUID` field that is not long. The field should be made long if it is intended to specify the version UID for purposes of serialization. This rule is deprecated, use {rule:squid:S2057} instead.
208	Bad practice - Method may fail to close database resource	CRITICAL	The method creates a database resource (such as a database connection or row set), does not assign it to any fields, pass it to other methods, or return it, and does not appear to close the object on all paths out of the method. Failure to close database resources on all paths out of a method may result in poor performance, and could cause the application to have problems communicating with the database.
209	Correctness - Suspicious reference comparison to constant	MAJOR	This method compares a reference value to a constant using the == or != operator, where the correct way to compare instanc of this type is generally with the equals() method. It is possible to create distinct instances that are equal but do not compare Examples of classes which should generally not be compared by reference are java.lang.Integer, java.lang.Float, etc. This rule is deprecated, use {rule:squid:S1698} instead.
210	Security - JSP reflected cross site scripting vulnerability	CRITICAL	This code directly writes an HTTP parameter to JSP output, which allows for a cross site scripting vulnerability. See http://en.wikipedia.org/wiki/Cross-site_scripting for more information. FindBugs looks only for the most blatant, obvious cases of cross site scripting. If FindBugs found *any*, you *almost certainly* have more cross site scripting vulnerabilities that FindBugs doesn't report. If you are concerned about cross site scripting, you should seriously consider using a commercial static analysis or pen-testing tool.
211	Performance - Method uses toArray() with zero-length array argument	CRITICAL	This method uses the toArray() method of a collection derived class, and passes in a zero-length prototype array argument. It is more efficient to use `myCollection.toArray(new Foo[myCollection.size()])` If the array passed in is big enough to store all of the elements of the collection, then it is populated and returned directly. This avoids the need to create a second array (by reflection) to return as the result.
212	Performance - Should be a static inner class	MAJOR	This class is an inner class, but does not use its embedded reference to the object which created it. This reference makes the instances of the class larger, and may keep the reference to the creator object alive longer than necessary. If possible, the class should be made static.

213	Bad practice - Comparison of String parameter using == or !=	MAJOR	This code compares a `java.lang.String` parameter for reference equality using the == or != operators. Requiring callers to pass only String constants or interned strings to a method is unnecessarily fragile, and rarely leads to measurable performance gains. Consider using the `equals(Object)` method instead. This rule is deprecated, use {rule:squid:StringEqualityComparisonCheck} instead.
214	Correctness - A collection is added to itself	CRITICAL	A collection is added to itself. As a result, computing the hashCode of this set will throw a StackOverflowException. This rule is deprecated, use {rule:squid:S2114} instead.
215	Correctness - The type of a supplied argument doesn't match format specifier	CRITICAL	One of the arguments is uncompatible with the corresponding format string specifier. As a result, this will generate a runtime exception when executed. For example, `String.format("%d", "1")` will generate an exception, since the String "1" is incompatible with the format specifier %d. This rule is deprecated, use {rule:squid:S2275} instead.
216	Correctness - A parameter is dead upon entry to a method but overwritten	CRITICAL	The initial value of this parameter is ignored, and the parameter is overwritten here. This often indicates a mistaken belief that the write to the parameter will be conveyed back to the caller.
217	Bad practice - Covariant compareTo() method defined	MAJOR	This class defines a covariant version of `compareTo()` . To correctly override the `compareTo()` method in the `Comparable` interface, the parameter of `compareTo()` must have type `java.lang.Object` .
218	Malicious code vulnerability - Field is a mutable array	MAJOR	A final static field references an array and can be accessed by malicious code or by accident from another package. This code can freely modify the contents of the array.
219	Multithreaded correctness - Unconditional wait	MAJOR	This method contains a call to `java.lang.Object.wait()` which is not guarded by conditional control flow. The code should verify that condition it intends to wait for is not already satisfied before calling wait; any previous notifications will be ignored. This rule is deprecated, use {rule:squid:S2274} instead.

220	An increment to a volatile field isn't atomic	CRITICAL	This code increments a volatile field. Increments of volatile fields aren't atomic. If more than one thread is incrementing the field at the same time, increments could be lost.
221	Dead store to local variable that shadows field	MAJOR	This instruction assigns a value to a local variable, but the value is not read or used in any subsequent instruction. Often, this indicates an error, because the value computed is never used. There is a field with the same name as the local variable. Did you mean to assign to that variable instead?
222	compareTo()/compare() returns Integer.MIN_VALUE	MAJOR	In some situation, this compareTo or compare method returns the constant Integer.MIN*VALUE, which is an exceptionally bad practice. The only thing that matters about the return value of compareTo is the sign of the result. But people will sometimes negate the return value of compareTo, expecting that this will negate the sign of the result. And it will, except in the case where the value returned is Integer.MIN*VALUE. So just return -1 rather than Integer.MIN_VALUE. This rule is deprecated, use {rule:squid:S2167} instead.

223	Multithreaded correctness - Synchronization on boxed primitive values	CRITICAL	The code synchronizes on an apparently unshared boxed primitive, such as an Integer.
			`private static final Integer fileLock = new Integer(1);`
			`...`
			`  synchronized(fileLock) {`
			`     .. do something ..`
			`     }`
			`...`
			It would be much better, in this code, to redeclare fileLock as
			`private static final Object fileLock = new Object();`
			The existing code might be OK, but it is confusing and a future refactoring, such as the "Remove Boxing" refactoring in IntelliJ, might replace this with the use of an interned Integer object shared throughout the JVM, leading to very confusing behavior and potential deadlock.
			This rule is deprecated, use {rule:squid:S1860} instead.

224	Bad practice - Method with Boolean return type returns explicit null	MAJOR	A method that returns either Boolean.TRUE, Boolean.FALSE or null is an accident waiting to happen. This method can be invoked as though it returned a value of type boolean, and the compiler will insert automatic unboxing of the Boolean value. If a null value is returned, this will result in a NullPointerException. This rule is deprecated, use {rule:squid:S2447} instead.

225	Multithreaded correctness - Synchronization on boxed primitive could lead to deadlock	CRITICAL	The code synchronizes on a boxed primitive constant, such as an Integer.
			`private static Integer count = 0;`
			`...`
			`  synchronized(count) {`
			`     count++;`
			`     }`
			`...`
			Since Integer objects can be cached and shared,
			this code could be synchronizing on the same object as other, unrelated code, leading to unresponsiveness
			and possible deadlock
			This rule is deprecated, use {rule:squid:S1860} instead.

226	Bad practice - Method might drop exception	MAJOR	This method might drop an exception. In general, exceptions should be handled or reported in some way, or they should be thrown out of the method. This rule is deprecated, use {rule:squid:S00108} instead.
227	Dodgy - Method discards result of readLine after checking if it is nonnull	MAJOR	The value returned by readLine is discarded after checking to see if the return value is non-null. In almost all situations, if the result is non-null, you will want to use that non-null value. Calling readLine again will give you a different line.
228	Performance - Unread field	MAJOR	This field is never read. Consider removing it from the class. This rule is deprecated, use {rule:squid:S1068} instead.
229	Correctness - A known null value is checked to see if it is an instance of a type	BLOCKER	This instanceof test will always return false, since the value being checked is guaranteed to be null. Although this is safe, make sure it isn't an indication of some misunderstanding or some other logic error. This rule is deprecated, use {rule:squid:S1850} instead.
230	Malicious code vulnerability - Field should be package protected	MAJOR	A mutable static field could be changed by malicious code or by accident. The field could be made package protected to avoid this vulnerability.

231	Correctness - Value required to not have type qualifier, but marked as unknown	CRITICAL	A value is used in a way that requires it to be never be a value denoted by a type qualifier, but there is an explicit annotation stating that it is not known where the value is prohibited from having that type qualifier. Either the usage or the annotation is incorrect.
232	Code checks for specific values returned by compareTo	MAJOR	This code invoked a compareTo or compare method, and checks to see if the return value is a specific value, such as 1 or -1. When invoking these methods, you should only check the sign of the result, not for any specific non-zero value. While many or most compareTo and compare methods only return -1, 0 or 1, some of them will return other values. This rule is deprecated, use {rule:squid:S2200} instead.

233	int value converted to long and used as absolute time	MAJOR	This code converts a 32-bit int value to a 64-bit long value, and then passes that value for a method parameter that requires an absolute time value. An absolute time value is the number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT. For example, the following method, intended to convert seconds since the epoc into a Date, is badly broken:
			`Date getDate(int seconds) { return new Date(seconds * 1000); }`
			The multiplication is done using 32-bit arithmetic, and then converted to a 64-bit value. When a 32-bit value is converted to 64-bits and used to express an absolute time value, only dates in December 1969 and January 1970 can be represented.
			Correct implementations for the above method are:
			`// Fails for dates after 2037`
			`Date getDate(int seconds) { return new Date(seconds * 1000L); }`
			`// better, works for all dates`
			`Date getDate(long seconds) { return new Date(seconds * 1000); }`
			This rule is deprecated, use {rule:squid:S2184} instead.

234	Multithreaded correctness - Method calls Thread.sleep() with a lock held	CRITICAL	This method calls Thread.sleep() with a lock held. This may result in very poor performance and scalability, or a deadlock, since other threads may be waiting to acquire the lock. It is a much better idea to call wait() on the lock, which releases the lock and allows other threads to run. This rule is deprecated, use {rule:squid:S2276} instead.
			A class's `finalize()` method should have protected access,

235	Malicious code vulnerability - Finalizer should be protected, not public	MAJOR	not public. This rule is deprecated, use {rule:squid:S1174} instead.
236	Correctness - Unneeded use of currentThread() call, to call interrupted()	CRITICAL	This method invokes the Thread.currentThread() call, just to call the interrupted() method. As interrupted() is a static method, is more simple and clear to use Thread.interrupted().
237	Bad practice - Check for sign of bitwise operation	CRITICAL	This method compares an expression such as `((event.detail & SWT.SELECTED) > 0)` . Using bit arithmetic and then comparing with the greater than operator can lead to unexpected results (of course depending on the value of SWT.SELECTED). If SWT.SELECTED is a negative number, this is a candidate for a bug. Even when SWT.SELECTED is not negative, it seems good practice to use '!= 0' instead of '> 0'. *Boris Bokowski*
238	Correctness - Field not initialized in constructor	MINOR	This field is never initialized within any constructor, and is therefore could be null after the object is constructed. This could be a either an error or a questionable design, since it means a null pointer exception will be generated if that field
239	Correctness - Method call passes null to a nonnull parameter	CRITICAL	This method passes a null value as the parameter of a method which must be nonnull. Either this parameter has been explicitly marked as @Nonnull, or analysis has determined that this parameter is always dereferenced.
240	Bad practice - Empty finalizer should be deleted	MAJOR	Empty `finalize()` methods are useless, so they should be deleted. This rule is deprecated, use {rule:squid:S1186} instead.
241	Correctness - close() invoked on a value that is always null	BLOCKER	close() is being invoked on a value that is always null. If this statement is executed, a null pointer exception will occur. But the big risk here you never close something that should be closed.
242	Correctness - Field only ever set to null	CRITICAL	All writes to this field are of the constant value null, and thus all reads of the field will return null. Check for errors, or remove it if it is useless. This rule is deprecated, use {rule:squid:S1068} instead.
			A call to `notify()` or `notifyAll()`

243	Multithreaded correctness - Naked notify	CRITICAL	was made without any (apparent) accompanying modification to mutable object state. In general, calling a notify method on a monitor is done because some condition another thread is waiting for has become true. However, for the condition to be meaningful, it must involve a heap object that is visible to both threads. This bug does not necessarily indicate an error, since the change to mutable object state may have taken place in a method which then called the method containing the notification.
244	Bad practice - Method may fail to close stream	CRITICAL	The method creates an IO stream object, does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. It is generally a good idea to use a `finally` block to ensure that streams are closed.
245	Bad practice - Class names shouldn't shadow simple name of superclass	MAJOR	This class has a simple name that is identical to that of its superclass, except that its superclass is in a different package (e.g., `alpha.Foo` extends `beta.Foo`). This can be exceptionally confusing, create lots of situations in which you have to look at import statements to resolve references and creates many opportunities to accidently define methods that do not override methods in their superclasses. This rule is deprecated, use {rule:squid:S2176} instead.
246	Bad practice - Store of non serializable object into HttpSession	CRITICAL	This code seems to be storing a non-serializable object into an HttpSession. If this session is passivated or migrated, an error will result. This rule is deprecated, use {rule:squid:S2441} instead.
247	Dodgy - Redundant comparison of two null values	CRITICAL	This method contains a redundant comparison of two references known to both be definitely null.
248	Performance - Could be refactored into a named static inner class	MAJOR	This class is an inner class, but does not use its embedded reference to the object which created it. This reference makes the instances of the class larger, and may keep the reference to the creator object alive longer than necessary. If possible, the class should be made into a *static* inner class. Since anonymous inner classes cannot be marked as static, doing this will require refactoring the inner class so that it is a named inner class.
249	Bad practice - Class is not derived from an Exception, even though it is named as such	MAJOR	This class is not derived from another exception, but ends with 'Exception'. This will be confusing to users of this class.

			This rule is deprecated, use {rule:squid:S2166} instead.
250	Correctness - Method must be private in order for serialization to work	MAJOR	This class implements the `Serializable` interface, and defines a method for custom serialization/deserialization. But since that method isn't declared private, it will be silently ignored by the serialization/deserialization API. This rule is deprecated, use {rule:squid:S2061} instead.
251	Correctness - Method assigns boolean literal in boolean expression	CRITICAL	This method assigns a literal boolean value (true or false) to a boolean variable inside an if or while expression. Most probably this was supposed to be a boolean comparison using ==, not an assignment using =. This rule is deprecated, use {rule:squid:AssignmentInSubExpressionCheck} instead.
252	Performance - Unread field: should this field be static?	MAJOR	This class contains an instance final field that is initialized to a compile-time static value. Consider making the field static. This rule is deprecated, use {rule:squid:S1170} instead.
253	Correctness - equals() used to compare array and nonarray	CRITICAL	This method invokes the .equals(Object o) to compare an array and a reference that doesn't seem to be an array. If things being compared are of different types, they are guaranteed to be unequal and the comparison is almost certainly an error. Even if they are both arrays, the equals method on arrays only determines of the two arrays are the same object. To compare the contents of the arrays, use java.util.Arrays.equals(Object[], Object[]).
254	Correctness - Incompatible bit masks (BIT_IOR)	CRITICAL	This method compares an expression of the form (e
255	Malicious code vulnerability - Field isn't final but should be	MAJOR	A mutable static field could be changed by malicious code or by accident from another package. The field could be made final to avoid this vulnerability. This rule is deprecated, use {rule:squid:S1444} instead.
256	Bad practice - Transient field that isn't set by deserialization.	MAJOR	This class contains a field that is updated at multiple places in the class, thus it seems to be part of the state of the class. However, since the field is marked as transient and not set in readObject or readResolve, it will contain the default deserialized instance of the class.
			A format-string method with a variable number of arguments is called,

257	Correctness - More arguments are passed that are actually used in the format string	MAJOR	but more arguments are passed than are actually used by the format string. This won't cause a runtime exception, but the code may be silently omitting information that was intended to be included in the formatted string. This rule is deprecated, use {rule:squid:S2275} instead.
258	Correctness - equals method always returns false	BLOCKER	This class defines an equals method that always returns false. This means that an object is not equal to itself, and it is impossible to create useful Maps or Sets of this class. More fundamentally, it means that equals is not reflexive, one of the requirements of the equals method. The likely intended semantics are object identity: that an object is equal to itself. This is the behavior inherited from class `Object`. If you need to override an equals inherited from a different superclass, you can use use: `public boolean equals(Object o) { return this == o; }`
259	Unread public/protected field	INFO	This field is never read. The field is public or protected, so perhaps it is intended to be used with classes not seen as part of the analysis. If not, consider removing it from the class.
260	Dodgy - Vacuous bit mask operation on integer value	CRITICAL	This is an integer bit operation (and, or, or exclusive or) that doesn't do any useful work (e.g., `v & 0xffffffff`). This rule is deprecated, use {rule:squid:S2437} instead.
261	Correctness - An apparent infinite loop	CRITICAL	This loop doesn't seem to have a way to terminate (other than by perhaps throwing an exception).
262	Correctness - equals method compares class names rather than class objects	MAJOR	This method checks to see if two objects are the same class by checking to see if the names of their classes are equal. You can have different classes with the same name if they are loaded by different class loaders. Just check to see if the class objects are the same. This rule is deprecated, use {rule:squid:S1872} instead.
263	Correctness - Null pointer dereference in method on exception path	CRITICAL	A pointer which is null on an exception path is dereferenced here. This will lead to a `NullPointerException` when the code is executed. Note that because FindBugs currently does not prune infeasible exception paths, this may be a false warning. Also note that FindBugs considers the default case of a switch statement to be an exception path, since the default case is often infeasible.

264	Dodgy - Method uses the same code for two branches	CRITICAL	This method uses the same code to implement two branches of a conditional branch. Check to ensure that this isn't a coding mistake.
265	Bad practice - Dubious catching of IllegalMonitorStateException	MAJOR	IllegalMonitorStateException is generally only thrown in case of a design flaw in your code (calling wait or notify on an object you do not hold a lock on). This rule is deprecated, use {rule:squid:S2235} instead.
266	Security - Servlet reflected cross site scripting vulnerability	CRITICAL	This code directly writes an HTTP parameter to Servlet output, which allows for a reflected cross site scripting vulnerability. See http://en.wikipedia.org/wiki/Cross-site_scripting for more information. FindBugs looks only for the most blatant, obvious cases of cross site scripting. If FindBugs found *any*, you *almost certainly* have more cross site scripting vulnerabilities that FindBugs doesn't report. If you are concerned about cross site scripting, you should seriously consider using a commercial static analysis or pen-testing tool.
267	Dodgy - Code contains a hard coded reference to an absolute pathname	CRITICAL	This code constructs a File object using a hard coded to an absolute pathname (e.g., `new File("/home/dannyc/workspace/j2ee/src/share/com/sun/enterprise/deployment");`
268	Correctness - Call to equals() comparing unrelated class and interface	CRITICAL	This method calls equals(Object) on two references, one of which is a class and the other an interface, where neither the class nor any of its non-abstract subclasses implement the interface. Therefore, the objects being compared are unlikely to be members of the same class at runtime (unless some application classes were not analyzed, or dynamic class loading can occur at runtime). According to the contract of equals(), objects of different classes should always compare as unequal; therefore, according to the contract defined by java.lang.Object.equals(Object), the result of this comparison will always be false at runtime.
269	Dodgy - Dereference of the result of readLine() without nullcheck	CRITICAL	The result of invoking readLine() is dereferenced without checking to see if the result is null. If there are no more lines of text to read, readLine() will return null and dereferencing that will generate a null pointer exception.
270	Correctness - Impossible cast	BLOCKER	This cast will always throw a ClassCastException.
271	Correctness - Possible null pointer dereference in method on exception path	CRITICAL	A reference value which is null on some exception control path is dereferenced here. This may lead to a `NullPointerException` when the code is executed. Note that because FindBugs currently does not prune infeasible exception paths, this may be a false warning.

			Also note that FindBugs considers the default case of a switch statement to be an exception path, since the default case is often infeasible.
272	Dodgy - Unsigned right shift cast to short/byte	CRITICAL	The code performs an unsigned right shift, whose result is then cast to a short or byte, which discards the upper bits of the result. Since the upper bits are discarded, there may be no difference between a signed and unsigned right shift (depending upon the size of the shift).
273	Dodgy - Initialization circularity	CRITICAL	A circularity was detected in the static initializers of the two classes referenced by the bug instance. Many kinds of unexpected behavior may arise from such circularity.
			This method uses a static method from java.lang.Math on a constant value. This method's result in this case, can be determined statically, and is faster and sometimes more accurate to just use the constant. Methods detected are:

<table>
<tr>
<td>274</td>
<td>Performance - Method calls static Math class method on a constant value</td>
<td>CRITICAL</td>
<td>
<table>
<tr><th>Method</th><th>Parameter</th></tr>
<tr><td>abs</td><td>-any-</td></tr>
<tr><td>acos</td><td>0.0 or 1.0</td></tr>
<tr><td>asin</td><td>0.0 or 1.0</td></tr>
<tr><td>atan</td><td>0.0 or 1.0</td></tr>
<tr><td>atan2</td><td>0.0</td></tr>
<tr><td>cbrt</td><td>0.0 or 1.0</td></tr>
<tr><td>ceil</td><td>-any-</td></tr>
<tr><td>cos</td><td>0.0</td></tr>
<tr><td>cosh</td><td>0.0</td></tr>
<tr><td>exp</td><td>0.0 or 1.0</td></tr>
<tr><td>expm1</td><td>0.0</td></tr>
<tr><td>floor</td><td>-any-</td></tr>
<tr><td>log</td><td>0.0 or 1.0</td></tr>
<tr><td>log10</td><td>0.0 or 1.0</td></tr>
<tr><td>rint</td><td>-any-</td></tr>
<tr><td>round</td><td>-any-</td></tr>
<tr><td>sin</td><td>0.0</td></tr>
<tr><td>sinh</td><td>0.0</td></tr>
<tr><td>sqrt</td><td>0.0 or 1.0</td></tr>
<tr><td>tan</td><td>0.0</td></tr>
<tr><td>tanh</td><td>0.0</td></tr>
<tr><td>toDegrees</td><td>0.0 or 1.0</td></tr>
<tr><td>toRadians</td><td>0.0</td></tr>
</table>
</td>
</tr>
</table>

275	Correctness - Nullcheck of value previously dereferenced	CRITICAL	A value is checked here to see whether it is null, but this value can't be null because it was previously dereferenced and if it were null a null pointer exception would have occurred at the earlier dereference. Essentially, this code and the previous dereference disagree as to whether this value is allowed to be null. Either the check is redundant or the previous dereference is erroneous.
276	Dodgy - Class doesn't override equals in superclass	MAJOR	This class extends a class that defines an equals method and adds fields, but doesn't define an equals method itself. Thus, equality on instances of this class will ignore the identity of the subclass and the added fields. Be sure this is what is intended, and that you don't need to override the equals method. Even if you don't need to override the equals method, consider overriding it anyway to document the fact that the equals method for the subclass just return the result of invoking super.equals(o). This rule is deprecated, use {rule:squid:S2160} instead.
277	Class defines tostring(); should it be toString()?	MAJOR	This class defines a method called `tostring()` . This method does not override the `toString()` method in `java.lang.Object` , which is probably what was intended.
278	Multithreaded correctness - A thread was created using the default empty run method	MAJOR	This method creates a thread without specifying a run method either by deriving from the Thread class, or by passing a Runnable object. This thread, then, does nothing but waste time. This rule is deprecated, use {rule:squid:S2134} instead.
			This method calls `wait()` , `notify()` or `notifyAll()` on an object that also provides an `await()` , `signal()` , `signalAll()` method (such as util.concurrent Condition objects). This probably isn't what you want, and even if you do want it, you should consider changing your

279	Using monitor style wait methods on util.concurrent abstraction	MAJOR	design, as other developers will find it exceptionally confusing. This rule is deprecated, use {rule:squid:S1844} instead.
280	Bad practice - Finalizer nullifies superclass finalizer	CRITICAL	This empty `finalize()` method explicitly negates the effect of any finalizer defined by its superclass. Any finalizer actions defined for the superclass will not be performed. Unless this is intended, delete this method. This rule is deprecated, use {rule:squid:ObjectFinalizeOverridenCallsSuperFinalizeCheck}, {rule:squid:S1186} instead.
281	Malicious code vulnerability - Field is a mutable Hashtable	MAJOR	A final static field references a Hashtable and can be accessed by malicious code or by accident from another package. This code can freely modify the contents of the Hashtable.
282	Bad practice - Confusing method names	MAJOR	The referenced methods have names that differ only by capitalization.
283	Performance - Method allocates an object, only to get the class object	MAJOR	This method allocates an object just to call getClass() on it, in order to retrieve the Class object for it. It is simpler to just access the .class property of the class. This rule is deprecated, use {rule:squid:S2133} instead.
284	Dodgy - Computation of average could overflow	CRITICAL	The code computes the average of two integers using either division or signed right shift, and then uses the result as the index of an array. If the values being averaged are very large, this can overflow (resulting in the computation of a negative average). Assuming that the result is intended to be nonnegative, you can use an unsigned right shift instead. In other words, rather that using `(low+high)/2`, use `(low+high) >>> 1` This bug exists in many earlier implementations of binary search and merge sort. Martin Buchholz found and fixed it in the JDK libraries, and Joshua Bloch widely publicized the bug pattern.
285	Bad practice - Non-serializable class has a serializable inner class	MINOR	This Serializable class is an inner class of a non-serializable class. Thus, attempts to serialize it will also attempt to associate instance of the outer class with which it is associated, leading to a runtime error. If possible, making the inner class a static inner class should solve the problem. Making the outer class serializable might also work, but that would mean serializing an instance of the inner class would always also serialize the instance of the outer class, which it often not what you really want.

			This rule is deprecated, use {rule:squid:S2066} instead.
286	Correctness - Read of unwritten field	MAJOR	The program is dereferencing a field that does not seem to ever have a non-null value written to it. Dereferencing this value will generate a null pointer exception.
287	Bad practice - Method ignores exceptional return value	MAJOR	This method returns a value that is not checked. The return value should be checked since it can indicate an unusual or unexpected function execution. For example, the `File.delete()` method returns false if the file could not be successfully deleted (rather than throwing an Exception). If you don't check the result, you won't notice if the method invocation signals unexpected behavior by returning an atypical return value.

288	Comparing values with incompatible type qualifiers	MAJOR	A value specified as carrying a type qualifier annotation is compared with a value that doesn't ever carry that qualifier.
			More precisely, a value annotated with a type qualifier specifying when=ALWAYS is compared with a value that where the same type qualifier specifies when=NEVER.
			For example, say that @NonNegative is a nickname for the type qualifier annotation @Negative(when=When.NEVER). The following code will generate this warning because the return statement requires a @NonNegative value, but receives one that is marked as @Negative.
			`public boolean example(@Negative Integer value1, @NonNegative Integer value2) {`
			`  return value1.equals(value2);`
			`}`

289	Bad practice - Suspicious reference comparison	CRITICAL	This method compares two reference values using the == or != operator, where the correct way to compare instances of this type is generally with the equals() method. Examples of classes which should generally not be compared by reference are java.lang.Integer, java.lang.Float, etc. This rule is deprecated, use {rule:squid:S1698} instead.
290	Correctness - Class defines field that masks a superclass field	MAJOR	This class defines a field with the same name as a visible instance field in a superclass. This is confusing, and may indicate an error if methods update or access one of the fields when they wanted the other.
	Dodgy - Class exposes		This class uses synchronization along with wait(), notify() or notifyAll() on itself (the this reference). Client classes that use this class, may, in addition, use an instance of this class

291	synchronization and semaphores in its public interface	CRITICAL	as a synchronizing object. Because two classes are using the same object for synchronization, Multithread correctness is suspect. You should not synchronize nor call semaphore methods on a public reference. Consider using a internal private member variable to control synchronization.
292	Dodgy - Class too big for analysis	MINOR	This class is bigger than can be effectively handled, and was not fully analyzed for errors.
293	Performance - Huge string constants is duplicated across multiple class files	CRITICAL	A large String constant is duplicated across multiple class files. This is likely because a final field is initialized to a String constant, and the Java language mandates that all references to a final field from other classes be inlined into that classfile. See JDK bug 6447475 for a description of an occurrence of this bug in the JDK and how resolving it reduced the size of the JDK by 1 megabyte.
294	Bad practice - Class is Serializable but its superclass doesn't define a void constructor	MAJOR	This class implements the `Serializable` interface and its superclass does not. When such an object is deserialized, the fields of the superclass need to be initialized by invoking the void constructor of the superclass. Since the superclass does not have one, serialization and deserialization will fail at runtime. This rule is deprecated, use {rule:squid:S2055} instead.
295	Multithreaded correctness - Mismatched notify()	CRITICAL	This method calls Object.notify() or Object.notifyAll() without obviously holding a lock on the object. Calling notify() or notifyAll() without a lock held will result in an `IllegalMonitorStateException` being thrown. This rule is deprecated, use {rule:squid:S2273} instead.
296	Multithreaded correctness - Call to static DateFormat	CRITICAL	As the JavaDoc states, DateFormats are inherently unsafe for multithreaded use. The detector has found a call to an instance of DateFormat that has been obtained via a static field. This looks suspicous. For more information on this see Sun Bug #6231579 and Sun Bug #6178997.

297	Dodgy - Double assignment of local variable	CRITICAL	This method contains a double assignment of a local variable; e.g.
			`public void foo() {`
			`  int x,y;`
			`  x = x = 17;`
			`}`
			Assigning the same value to a variable twice is useless, and may indicate a logic error or typo.

			This rule is deprecated, use {rule:squid:S1656} instead.
298	Bad practice - Method invoked that should be only be invoked inside a doPrivileged block	MAJOR	This code invokes a method that requires a security permission check. If this code will be granted security permissions, but might be invoked by code that does not have security permissions, then the invocation needs to occur inside a doPrivileged block.

299	Correctness - Method ignores return value	MAJOR	The return value of this method should be checked. One common cause of this warning is to invoke a method on an immutable object, thinking that it updates the object. For example, in the following code fragment,
			`String dateString = getHeaderField(name);`
			`dateString.trim();`
			the programmer seems to be thinking that the trim() method will update the String referenced by dateString. But since Strings are immutable, the trim() function returns a new String value, which is being ignored here. The code should be corrected to:
			`String dateString = getHeaderField(name);`
			`dateString = dateString.trim();`
			This rule is deprecated, use {rule:squid:S2201} instead.

300	Correctness - Very confusing method names	MAJOR	The referenced methods have names that differ only by capitalization. This is very confusing because if the capitalization were identical then one of the methods would override the other.
301	Correctness - Doomed attempt to append to an object output stream	CRITICAL	This code opens a file in append mode and then wraps the result in an object output stream. This won't allow you to append to an existing object output stream stored in a file. If you want to be able to append to an object output stream, you need to keep the object output stream open. The only situation in which opening a file in append mode and the writing an object output stream could work is if on reading the file you plan to open it in random access mode and seek to the byte offset where the append started. TODO: example.

302	Bad practice - Method invokes dangerous method runFinalizersOnExit	MAJOR	*Never call System.runFinalizersOnExit or Runtime.runFinalizersOnExit for any reason: they are among the most dangerous methods in the Java libraries.* -- Joshua Bloch This rule is deprecated, use {rule:squid:S2151} instead.
303	Multithreaded correctness - Static DateFormat	CRITICAL	As the JavaDoc states, DateFormats are inherently unsafe for multithreaded use. Sharing a single instance across thread boundaries without proper synchronization will result in erratic behavior of the application. You may also experience serialization problems. Using an instance field is recommended. For more information on this see Sun Bug #6231579 and Sun Bug #6178997.
304	Bad practice - Finalizer nulls fields	MAJOR	This finalizer nulls out fields. This is usually an error, as it does not aid garbage collection, and the object is going to be garbage collected anyway. This rule is deprecated, use {rule:squid:S2165} instead.
305	Correctness - Doomed test for equality to NaN	CRITICAL	This code checks to see if a floating point value is equal to the special Not A Number value (e.g., `if (x == Double.NaN)`). However, because of the special semantics of `NaN`, no value is equal to `Nan`, including `NaN`. Thus, `x == Double.NaN` always evaluates to false. To check to see if a value contained in `x` is the special Not A Number value, use `Double.isNaN(x)` (or `Float.isNaN(x)` if `x` is floating point precision). This rule is deprecated, use {rule:squid:S1244} instead.
306	Multithreaded correctness - Wait with two locks held	MAJOR	Waiting on a monitor while two locks are held may cause deadlock. Performing a wait only releases the lock on the object being waited on, not any other locks. This not necessarily a bug, but is worth examining closely.
			This code creates a database connect using a hardcoded, constant password. Anyone with access to either the source code

307	Security - Hardcoded constant database password	BLOCKER	or the compiled code can easily learn the password. This rule is deprecated, use {rule:squid:S2068} instead.
308	Reversed method arguments	MINOR	The arguments to this method call seem to be in the wrong order. For example, a call `Preconditions.checkNotNull("message", message)` has reserved arguments: the value to be checked is the first argument.
309	Performance - Maps and sets of URLs can be performance hogs	BLOCKER	This method or field is or uses a Map or Set of URLs. Since both the equals and hashCode method of URL perform domain name resolution, this can result in a big performance hit. See http://michaelscharf.blogspot.com/2006/11/javaneturlequals-and-hashcode-make.html for more information. Consider using `java.net.URI` instead.
310	Correctness - Deadly embrace of non-static inner class and thread local	MAJOR	This class is an inner class, but should probably be a static inner class. As it is, there is a serious danger of a deadly embrace between the inner class and the thread local in the outer class. Because the inner class isn't static, it retains a refere contains a reference to an instance of the inner class, the inner and outer instance will both be reachable and not eligible for

311	Bad practice - Method doesn't override method in superclass due to wrong package for parameter	MAJOR	The method in the subclass doesn't override a similar method in a superclass because the type of a parameter doesn't exactly match the type of the corresponding parameter in the superclass. For example, if you have:
			`import alpha.Foo;`
			`public class A {`
			`  public int f(Foo x) { return 17; }`
			`}`
			`----`
			`import beta.Foo;`
			`public class B extends A {`
			`  public int f(Foo x) { return 42; }`
			`  public int f(alpha.Foo x) { return 27; }`
			`}`
			The `f(Foo)` method defined in class `B` doesn't override the `f(Foo)` method defined in class `A` , because the argument types are `Foo` 's from different packages.
			In this case, the subclass does define a method with a signature identical to the method in the superclass, so this is presumably understood. However, such methods are exceptionally confusing. You should strongly consider removing or deprecating the method with the similar but not identical signature.

312	Correctness - Integer remainder modulo 1	CRITICAL	Any expression (exp % 1) is guaranteed to always return zero. Did you mean (exp & 1) or (exp % 2) instead?

313	Bad practice - Comparison of String objects using == or !=	MAJOR	This code compares `java.lang.String` objects for reference equality using the == or != operators. Unless both strings are either constants in a source file, or have been interned using the `String.intern()` method, the same string value may be represented by two different String objects. Consider using the `equals(Object)` method instead. This rule is deprecated, use {rule:squid:StringEqualityComparisonCheck} instead.
314	Dodgy - Complicated, subtle or wrong increment in for-loop	CRITICAL	Are you sure this for loop is incrementing the correct variable? It appears that another variable is being initialized and checked by the for loop. This rule is deprecated, use {rule:squid:S1994} instead.
315	Dodgy - Non-Boolean argument formatted using %b format specifier	MAJOR	An argument not of type Boolean is being formatted with a %b format specifier. This won't throw an exception; instead, it will print true for any nonnull value, and false for null. This feature of format strings is strange, and may not be what you intended. This rule is deprecated, use {rule:squid:S2275} instead.
316	Dodgy - Method directly allocates a specific implementation of xml interfaces	CRITICAL	This method allocates a specific implementation of an xml interface. It is preferable to use the supplied factory classes to create these objects so that the implementation can be changed at runtime. See • javax.xml.parsers.DocumentBuilderFactory • javax.xml.parsers.SAXParserFactory • javax.xml.transform.TransformerFactory • org.w3c.dom.Document.create*XXXX* for details.
317	Multithreaded correctness - Class's writeObject() method is synchronized but nothing else is	CRITICAL	This class has a `writeObject()` method which is synchronized; however, no other method of the class is synchronized.
			This instruction assigns a class literal to a variable and then never uses it. The behavior of this differs in Java 1.4 and in Java 5. In Java 1.4 and earlier, a reference to `Foo.class` would force the static initializer for `Foo` to be executed, if it has not been executed already.

318	Correctness - Dead store of class literal	CRITICAL	In Java 5 and later, it does not. See Sun's article on Java SE compatibility for more details and examples, and suggestions on how to force class initialization in Java 5.
319	Bad practice - Equals method should not assume anything about the type of its argument	CRITICAL	The `equals(Object o)` method shouldn't make any assumptions about the type of `o`. It should simply return false if `o` is not the same type as `this`.

320	Multithreaded correctness - Synchronization on field in futile attempt to guard that field	MAJOR	This method synchronizes on a field in what appears to be an attempt to guard against simultaneous updates to that field. But guarding a field gets a lock on the referenced object, not on the field. This may not provide the mutual exclusion you need, and other threads might be obtaining locks on the referenced objects (for other purposes). An example of this pattern would be:
			`private Long myNtfSeqNbrCounter = new Long(0);`
			`private Long getNotificationSequenceNumber() {`
			`     Long result = null;`
			`     synchronized(myNtfSeqNbrCounter) {`
			`         result = new Long(myNtfSeqNbrCounter.longValue() + 1);`
			`         myNtfSeqNbrCounter = new Long(result.longValue());`
			`     }`
			`     return result;`
			` }`
			This rule is deprecated, use {rule:squid:S2445} instead.

321	Multithreaded correctness - Mismatched wait()	CRITICAL	This method calls Object.wait() without obviously holding a lock on the object. Calling wait() without a lock held will result in an `IllegalMonitorStateException` being thrown. This rule is deprecated, use {rule:squid:S2273} instead.
322	Performance - Use the nextInt method of Random rather than nextDouble to generate a random integer	MAJOR	If `r` is a `java.util.Random`, you can generate a random number from `0` to `n-1` using `r.nextInt(n)`, rather than using `(int)(r.nextDouble() * n)`.
323	Sequence of calls to concurrent abstraction may not be atomic	MAJOR	This code contains a sequence of calls to a concurrent abstraction (such as a concurrent hash map). These calls will not be executed atomically.
324	Dodgy - Dead store of null to local variable	CRITICAL	The code stores null into a local variable, and the stored value is not read. This store may have been introduced to assist the garbage collector, but as of Java SE 6.0, this is no longer needed or useful.

325	Performance - The equals and hashCode methods of URL are blocking	BLOCKER	The equals and hashCode method of URL perform domain name resolution, this can result in a big performance hit. See http://michaelscharf.blogspot.com/2006/11/javaneturlequals-and-hashcode-make.html for more information. Consider using `java.net.URI` instead.
326	Dodgy - Possible null pointer dereference on path that might be infeasible	CRITICAL	There is a branch of statement that, *if executed,* guarantees that a null value will be dereferenced, which would generate a `NullPointerException` when the code is executed. Of course, the problem might be that the branch or statement is infeasible and that the null pointer exception can't ever be executed; deciding that is beyond the ability of FindBugs. Due to the fact that this value had been previously tested for nullness, this is a definite possibility.
327	Absolute path traversal in servlet	MAJOR	The software uses an HTTP request parameter to construct a pathname that should be within a restricted directory, but it does not properly neutralize absolute path sequences such as "/abs/path" that can resolve to a location that is outside of that directory. See http://cwe.mitre.org/data/definitions/36.html for more information. FindBugs looks only for the most blatant, obvious cases of absolute path traversal. If FindBugs found *any*, you *almost certainly* have more vulnerabilities that FindBugs doesn't report. If you are concerned about absolute path traversal, you should seriously consider using a commercial static analysis or pen-testing tool.
328	Bad practice - Very confusing method names (but perhaps intentional)	MAJOR	The referenced methods have names that differ only by capitalization. This is very confusing because if the capitalization were identical then one of the methods would override the other. From the existence of other methods, it seems that the existence of both of these methods is intentional, but is sure is confusing. You should try hard to eliminate one of them, unless you are forced to have both due to frozen APIs.
329	Security - Servlet reflected cross site scripting vulnerability	CRITICAL	This code directly writes an HTTP parameter to a Server error page (using HttpServletResponse.sendError). Echoing this untrusted input allows for a reflected cross site scripting vulnerability. See http://en.wikipedia.org/wiki/Cross-site_scripting for more information. FindBugs looks only for the most blatant, obvious cases of cross site scripting. If FindBugs found *any*, you *almost certainly* have more cross site scripting vulnerabilities that FindBugs doesn't report. If you are concerned about cross site scripting, you should seriously consider using a commercial static analysis or pen-testing tool.
330	Dodgy - private readResolve method not inherited by subclasses	MAJOR	This class defines a private readResolve method. Since it is private, it won't be inherited by subclasses. This might be intentional and OK, but should be reviewed to ensure it is what is intended.
331	Correctness - Invocation of toString on an array	CRITICAL	The code invokes toString on an array, which will generate a fairly useless result such as [C@16f0472. Consider using Arrays.toString to convert the array into a readable String that gives the contents of the array. See Programming Puzzlers, chapter 3, puzzle 12.

			This rule is deprecated, use {rule:squid:S2116} instead.
332	Correctness - Method does not check for null argument	MAJOR	A parameter to this method has been identified as a value that should always be checked to see whether or not it is null, but it is being dereferenced without a preceding null check.
333	Bad practice - Abstract class defines covariant compareTo() method	MAJOR	This class defines a covariant version of `compareTo()` . To correctly override the `compareTo()` method in the `Comparable` interface, the parameter of `compareTo()` must have type `java.lang.Object` .
334	Correctness - Return value of putIfAbsent ignored, value passed to putIfAbsent reused	MAJOR	The putIfAbsent method is typically used to ensure that a single value is associated with a given key (the first value for which put if absent succeeds). If you ignore the return value and retain a reference to the value passed in, you run the risk of is associated with the key in the map. If it matters which one you use and you use the one that isn't stored in the map, your p

335	Multithreaded correctness - Method does not release lock on all exception paths	CRITICAL	This method acquires a JSR-166 (`java.util.concurrent`) lock, but does not release it on all exception paths out of the method. In general, the correct idiom for using a JSR-166 lock is:
			`Lock l = ...;`
			`l.lock();`
			`try {`
			`    // do something`
			`} finally {`
			`    l.unlock();`
			`}`

336	Non-transient non-serializable instance field in serializable class	MAJOR	This Serializable class defines a non-primitive instance field which is neither transient, Serializable, or `java.lang.Object` , and does not appear to implement the `Externalizable` interface or the `readObject()` and `writeObject()` methods. Objects of this class will not be deserialized correctly if a non-Serializable object is stored in this field. This rule is deprecated, use {rule:squid:S1948} instead.

337	Dodgy - Self assignment of local variable	CRITICAL	This method contains a self assignment of a local variable; e.g.
			`public void foo() {`
			`  int x = 3;`
			`  x = x;`
			`}`
			Such assignments are useless, and may indicate a logic error or typo.
			This rule is deprecated, use {rule:squid:S1656} instead.

338	Read of unwritten public or protected field	MAJOR	The program is dereferencing a public or protected field that does not seem to ever have a non-null value written to it. Unless the field is initialized via some mechanism not seen by the analysis, dereferencing this value will generate a null pointer exception.
339	Bad practice - Method ignores results of InputStream.read()	MAJOR	This method ignores the return value of one of the variants of `java.io.InputStream.read()` which can return multiple bytes. If the return value is not checked, the caller will not be able to correctly handle the case where fewer bytes were read than the caller requested. This is a particularly insidious kind of bug, because in many programs, reads from input streams usually do read the full amount of data requested, causing the program to fail only sporadically.
340	Dodgy - Write to static field from instance method	CRITICAL	This instance method writes to a static field. This is tricky to get correct if multiple instances are being manipulated, and generally bad practice.
341	Bad practice - toString method may return null	CRITICAL	This toString method seems to return null in some circumstances. A liberal reading of the spec could be interpreted as allowing this, but it is probably a bad idea and could cause other code to break. Return the empty string or some other appropriate string rather than null. This rule is deprecated, use {rule:squid:S2225} instead.
342	Correctness - No relationship between generic parameter and method argument	CRITICAL	This call to a generic collection method contains an argument with an incompatible class from that of the collection's parameter (i.e., the type of the argument is neither a supertype nor a subtype of the corresponding generic type argument). Therefore, it is unlikely that the collection contains any objects that are equal to the method argument used here. Most likely, the wrong value is being passed to the method. In general, instances of two unrelated classes are not equal. For example, if the `Foo` and `Bar` classes are not related by subtyping, then an instance of `Foo` should not be equal to an instance of `Bar`. Among other issues, doing so will likely result in an equals method that is not symmetrical. For example, if you define the `Foo` class so that a `Foo` can be equal to a `String`, your equals method isn't symmetrical since a `String` can only be equal to a `String`. In rare cases, people do define nonsymmetrical equals methods and still manage to make their code work. Although none of the APIs document or guarantee it, it is typically the case that if you check if a `Collection<String>` contains a `Foo`, the equals method of argument (e.g., the equals method of the `Foo` class) used to perform the equality checks. This rule is deprecated, use {rule:squid:S2175} instead.

343	Correctness - An apparent infinite recursive loop	CRITICAL	This method unconditionally invokes itself. This would seem to indicate an infinite recursive loop that will result in a stack overflow.
344	Performance - Method invokes toString() method on a String	INFO	Calling `String.toString()` is just a redundant operation. Just use the String.
345	Bad practice - Creates an empty zip file entry	MAJOR	The code calls `putNextEntry()`, immediately followed by a call to `closeEntry()`. This results in an empty ZipFile entry. The contents of the entry should be written to the ZipFile between the calls to `putNextEntry()` and `closeEntry()`.
346	Dodgy - Questionable use of non-short-circuit logic	MAJOR	This code seems to be using non-short-circuit logic (e.g., & or
347	Performance - Method invokes inefficient Boolean constructor; use Boolean.valueOf(...) instead	MAJOR	Creating new instances of `java.lang.Boolean` wastes memory, since `Boolean` objects are immutable and there are only two useful values of this type. Use the `Boolean.valueOf()` method (or Java 1.5 autoboxing) to create `Boolean` objects instead.
348	Dodgy - Questionable cast to concrete collection	CRITICAL	This code casts an abstract collection (such as a Collection, List, or Set) to a specific concrete implementation (such as an ArrayList or HashSet). This might not be correct, and it may make your code fragile, since it makes it harder to switch to other concrete implementations at a future point. Unless you have a particular reason to do so, just use the abstract collection class.
349	Performance - Could be refactored into a static inner class	MAJOR	This class is an inner class, but does not use its embedded reference to the object which created it except during construction of the inner object. This reference makes the instances of the class larger, and may keep the reference to the creator object alive longer than necessary. If possible, the class should be made into a *static* inner class. Since the reference to the outer object is required during construction of the inner instance, the inner class will need to be refactored so as to pass a reference to the outer instance to the constructor for the inner class.
350	Multithreaded correctness - Incorrect lazy initialization of static field	CRITICAL	This method contains an unsynchronized lazy initialization of a non-volatile static field. Because the compiler or processor may reorder instructions, threads are not guaranteed to see a completely initialized object, *if the method can be called by multiple threads*. You can make the field volatile to correct the problem. For more information, see the Java Memory Model web site. This rule is deprecated, use {rule:squid:S2444} instead.
	Bad practice - Use of identifier		The identifier is a word that is reserved as a keyword in later versions of Java, and your code will need to be changed in order to compile it in later versions of Java.

351	that is a keyword in later versions of Java	MAJOR	This rule is deprecated, use {rule:squid:S1190} instead.

352	Dodgy - Result of integer multiplication cast to long	CRITICAL	This code performs integer multiply and then converts the result to a long, as in:
			`long convertDaysToMilliseconds(int days) { return 1000360024days; }`
			If the multiplication is done using long arithmetic, you can avoid the possibility that the result will overflow. For example, you could fix the above code to:
			`long convertDaysToMilliseconds(int days) { return 1000L360024days; }`
			or
			`static final long MILLISECONDSPERDAY = 24L36001000;`
			`long convertDaysToMilliseconds(int days) { return days * MILLISECONDSPERDAY; }`
			This rule is deprecated, use {rule:squid:S2184} instead.

353	Correctness - Static Thread.interrupted() method invoked on thread instance	CRITICAL	This method invokes the Thread.interrupted() method on a Thread object that appears to be a Thread object that is not the current thread. As the interrupted() method is static, the interrupted method will be called on a different object than the one the author intended.
354	D'oh! A nonsensical method invocation	MAJOR	This partical method invocation doesn't make sense, for reasons that should be apparent from inspection.

355	Correctness - Self assignment of field	CRITICAL	This method contains a self assignment of a field; e.g.
			`int x;`
			`public void foo() {`
			`  x = x;`
			`}`
			Such assignments are useless, and may indicate a logic error or typo.
			This rule is deprecated, use {rule:squid:S1656} instead.

356	Security - Empty database password	CRITICAL	This code creates a database connect using a blank or empty password. This indicates that the database is not protected by a password.

357	Correctness - TestCase declares a bad suite method	CRITICAL	Class is a JUnit TestCase and defines a suite() method.
			However, the suite method needs to be declared as either
			`public static junit.framework.Test suite()`
			or
			`public static junit.framework.TestSuite suite()`

358	Correctness - Format string references missing argument	CRITICAL	Not enough arguments are passed to satisfy a placeholder in the format string. A runtime exception will occur when this statement is executed. This rule is deprecated, use {rule:squid:S2275} instead.

359	Correctness - Bitwise add of signed byte value	CRITICAL	Adds a byte value and a value which is known to the 8 lower bits clear.
			Values loaded from a byte array are sign extended to 32 bits
			before any any bitwise operations are performed on the value.
			Thus, if `b[0]` contains the value `0xff`, and
			`x` is initially 0, then the code
			`((x << 8) + b[0])` will sign extend `0xff`
			to get `0xffffffff`, and thus give the value
			`0xffffffff` as the result.
			In particular, the following code for packing a byte array into an int is badly wrong:
			`int result = 0;`
			`for(int i = 0; i < 4; i++)`
			`  result = ((result << 8) + b[i]);`
			The following idiom will work instead:
			`int result = 0;`
			`for(int i = 0; i < 4; i++)`
			`  result = ((result << 8) + (b[i] & 0xff));`

			The method seems to be building a String using concatenation in a loop. In each iteration, the String is converted to a StringBuffer/StringBuilder, appended to, and converted back to a String. This can lead to a cost quadratic in the number of iterations, as the growing string is recopied in each iteration.

360	Performance - Method concatenates strings using + in a loop	CRITICAL	Better performance can be obtained by using
			a StringBuffer (or StringBuilder in Java 1.5) explicitly.
			
			For example:
			
			`// This is bad`
			`String s = "";`
			`for (int i = 0; i < field.length; ++i) {`
			`  s = s + field[i];`
			`}`
			
			`// This is better`
			`StringBuffer buf = new StringBuffer();`
			`for (int i = 0; i < field.length; ++i) {`
			`  buf.append(field[i]);`
			`}`
			`String s = buf.toString();`
			
			This rule is deprecated, use {rule:squid:S1643} instead.

361	Correctness - Value annotated as carrying a type qualifier used where a value that must not carry that qualifier is required	CRITICAL	A value specified as carrying a type qualifier annotation is
			consumed in a location or locations requiring that the value not
			carry that annotation.
			
			More precisely, a value annotated with a type qualifier specifying when=ALWAYS
			is guaranteed to reach a use or uses where the same type qualifier specifies when=NEVER.
			
			For example, say that @NonNegative is a nickname for
			the type qualifier annotation @Negative(when=When.NEVER).
			The following code will generate this warning because
			the return statement requires a @NonNegative value,
			but receives one that is marked as @Negative.
			
			`public @NonNegative Integer example(@Negative Integer value) {`
			`    return value;`
			`}`

362	Correctness - equals method always returns true	BLOCKER	This class defines an equals method that always returns true. This is imaginative, but not very smart. Plus, it means that the equals method is not symmetric.

363	Bad practice - Needless instantiation of class that only supplies static methods	MAJOR	This class allocates an object that is based on a class that only supplies static methods. This object does not need to be created, just access the static methods directly using the class name as a qualifier. This rule is deprecated, use {rule:squid:S2440} instead.
364	Dodgy - Questionable cast to abstract collection	MAJOR	This code casts a Collection to an abstract collection (such as `List` , `Set` , or `Map`). Ensure that you are guaranteed that the object is of the type you are casting to. If all you need is to be able to iterate through a collection, you don't need to cast it to a Set or List.
365	Bad practice - Iterator next() method can't throw NoSuchElementException	MINOR	This class implements the `java.util.Iterator` interface. However, its `next()` method is not capable of throwing `java.util.NoSuchElementException` . The `next()` method should be changed so it throws `NoSuchElementException` if is called when there are no more elements to return. This rule is deprecated, use {rule:squid:S2272} instead.
366	Malicious code vulnerability - May expose internal representation by returning reference to mutable object	MAJOR	Returning a reference to a mutable object value stored in one of the object's fields exposes the internal representation of the object. If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different. Returning a new copy of the object is better approach in many situations.
367	Multithreaded correctness - Mutable servlet field	MAJOR	A web server generally only creates one instance of servlet or jsp class (i.e., treats the class as a Singleton), and will have multiple threads invoke methods on that instance to service multiple simultaneous requests. Thus, having a mutable instance field generally creates race conditions. This rule is deprecated, use {rule:squid:S2226} instead.
368	Correctness - Bitwise OR of signed byte value	CRITICAL	Loads a value from a byte array and performs a bitwise OR with that value. Values loaded from a byte array are sign extended to 32 bits before any any bitwise operations are performed on the value. Thus, if `b[0]` contains the value `0xff` , and `x` is initially 0, then the code `((x << 8)`
369	Multithreaded correctness - Synchronization performed on java.util.concurrent Lock	CRITICAL	This method performs synchronization on an implementation of `java.util.concurrent.locks.Lock` . You should use the `lock()` and `unlock()` methods instead. This rule is deprecated, use {rule:squid:S2442} instead.

370	Bad practice - serialVersionUID isn't static	MAJOR	This class defines a `serialVersionUID` field that is not static. The field should be made static if it is intended to specify the version UID for purposes of serialization. This rule is deprecated, use {rule:squid:S2057} instead.
371	Dodgy - instanceof will always return true	CRITICAL	This instanceof test will always return true (unless the value being tested is null). Although this is safe, make sure it isn't an indication of some misunderstanding or some other logic error. If you really want to test the value for being null, perhaps it would be clearer to do better to do a null test rather than an instanceof test. This rule is deprecated, use {rule:squid:S1850} instead.
372	Correctness - Suspicious reference comparison of Boolean values	MAJOR	This method compares two Boolean values using the == or != operator. Normally, there are only two Boolean values (Boolea and Boolean.FALSE), but it is possible to create other Boolean objects using the new Boolean(b) constructor. It is best to avo checking Boolean objects for equality using == or != will give results than are different than you would get using .equals(...) This rule is deprecated, use {rule:squid:S1698} instead.
373	Dodgy - Potentially dangerous use of non-short-circuit logic	CRITICAL	This code seems to be using non-short-circuit logic (e.g., & or

374	Correctness - Impossible downcast of toArray() result	BLOCKER	This code is casting the result of calling toArray() on a collection to a type more specific than Object[], as in:
			`String[] getAsArray(Collection c) {`
			`  return (String[]) c.toArray();`
			`}`
			This will usually fail by throwing a ClassCastException. The `toArray()` of almost all collections return an `Object[]`. They can't really do anything else, since the Collection object has no reference to the declared generic type of
			The correct way to do get an array of a specific type from a collection is to use `c.toArray(new String[]);` or `c.toArray(new String[c.size()]);` (the latter is slightly more efficient).
			There is one common/known exception exception to this. The toArray() method of lists returned by Arrays.asList(...) will return a covariantly typed array. For example, `Arrays.asArray(new String[] { "a" }).toArray()` will return a S suppress such cases, but may miss some.

			A final static field that is defined in an interface references a mutable object such as an array or hashtable.

375	Malicious code vulnerability - Field should be moved out of an interface and made package protected	MAJOR	This mutable object could be changed by malicious code or by accident from another package. To solve this, the field needs to be moved to a class and made package protected to avoid this vulnerability. This rule is deprecated, use {rule:squid:S2386} instead.
376	Correctness - Call to equals() comparing different types	CRITICAL	This method calls equals(Object) on two references of different class types with no common subclasses. Therefore, the objects being compared are unlikely to be members of the same class at runtime (unless some application classes were not analyzed, or dynamic class loading can occur at runtime). According to the contract of equals(), objects of different classes should always compare as unequal; therefore, according to the contract defined by java.lang.Object.equals(Object), the result of this comparison will always be false at runtime.
377	Malicious code vulnerability - Public static method may expose internal representation by returning array	CRITICAL	A public static method returns a reference to an array that is part of the static state of the class. Any code that calls this method can freely modify the underlying array. One fix is to return a copy of the array.
378	Dodgy - Immediate dereference of the result of readLine()	CRITICAL	The result of invoking readLine() is immediately dereferenced. If there are no more lines of text to read, readLine() will return null and dereferencing that will generate a null pointer exception.
379	Multithreaded correctness - Empty synchronized block	MAJOR	The code contains an empty synchronized block: `synchronized() {}` Empty synchronized blocks are far more subtle and hard to use correctly than most people recognize, and empty synchronized blocks are almost never a better solution than less contrived solutions. This rule is deprecated, use {rule:squid:S00108} instead.
380	Security - HTTP cookie formed from untrusted input	MAJOR	This code constructs an HTTP Cookie using an untrusted HTTP parameter. If this cookie is added to an HTTP response, it will allow a HTTP response splitting vulnerability. See http://en.wikipedia.org/wiki/HTTP*response*splitting for more information. FindBugs looks only for the most blatant, obvious cases of HTTP response splitting.

			If FindBugs found *any*, you *almost certainly* have more vulnerabilities that FindBugs doesn't report. If you are concerned about HTTP response splitting, you should seriously consider using a commercial static analysis or pen-testing tool.
381	Correctness - Uncallable method defined in anonymous class	CRITICAL	This anonymous class defined a method that is not directly invoked and does not override a method in a superclass. Since methods in other classes cannot directly invoke methods declared in an anonymous class, it seems that this method is uncallable. The method might simply be dead code, but it is also possible that the method is intended to override a method declared in a superclass, and due to an typo or other error the method does not, in fact, override the method it is intended to.
382	Correctness - Overwritten increment	CRITICAL	The code performs an increment operation (e.g., `i++`) and then immediately overwrites it. For example, `i = i++` immediately overwrites the incremented value with the original value. This rule is deprecated, use {rule:squid:S2123} instead.
383	Relative path traversal in servlet	MAJOR	The software uses an HTTP request parameter to construct a pathname that should be within a restricted directory, but it does not properly neutralize sequences such as ".." that can resolve to a location that is outside of that directory. See http://cwe.mitre.org/data/definitions/23.html for more information. FindBugs looks only for the most blatant, obvious cases of relative path traversal. If FindBugs found *any*, you *almost certainly* have more vulnerabilities that FindBugs doesn't report. If you are concerned about relative path traversal, you should seriously consider using a commercial static analysis or pen-testing tool.
384	Singular Field	MINOR	A field that's only used by one method could perhaps be replaced by a local variable.
385	Avoid Assert As Identifier	MAJOR	Finds all places 'assert' is used as an identifier is used.
386	Unused formal parameter	MAJOR	Avoid passing parameters to methods or constructors and then not using those parameters.
387	Unused Private Field	MAJOR	Detects when a private field is declared and/or assigned a value, but not used.
388	String To String	MAJOR	Avoid calling toString() on String objects; this is unnecessary.
389	Big Integer Instantiation	MAJOR	Don't create instances of already existing BigInteger (BigInteger.ZERO, BigInteger.ONE) and for 1.5 on, BigInteger.TEN and BigDecimal (BigDecimal.ZERO, BigDecimal.ONE, BigDecimal.TEN)
390	String Buffer Instantiation With Char	MAJOR	StringBuffer sb = new StringBuffer('c'); The char will be converted into int to intialize StringBuffer size.
391	Integer Instantiation	MAJOR	In JDK 1.5, calling new Integer() causes memory allocation. Integer.valueOf() is more memory friendly.
392	Empty Finalizer	MAJOR	If the finalize() method is empty, then it does not need to exist.
393	String Instantiation	MAJOR	Avoid instantiating String objects; this is usually unnecessary.
394	Ncss Type Count	MAJOR	This rule uses the NCSS (Non Commenting Source Statements) algorithm to determine the number of lines of code for a give type. NCSS ignores comments, and counts actual statements. Using this algorithm, lines of code that are split are counted as
395	Missing Static Method In Non Instantiatable Class	MAJOR	A class that has private constructors and does not have any static methods or fields cannot be used.
396	Class Cast Exception With To Array	MAJOR	if you need to get an array of a class from your Collection, you should pass an array of the desidered class as the parameter of the toArray method. Otherwise you will get a ClassCastException.

397	Use Arrays As List	MAJOR	The class java.util.Arrays has a asList method that should be use when you want to create a new List from an array of object It is faster than executing a loop to cpy all the elements of the array one by one
398	Boolean Instantiation	MAJOR	Avoid instantiating Boolean objects; you can reference Boolean.TRUE, Boolean.FALSE, or call Boolean.valueOf() instead.
399	Compare Objects With Equals	MAJOR	Use equals() to compare object references; avoid comparing them with ==.
400	Use String Buffer Length	MINOR	Use StringBuffer.length() to determine StringBuffer length rather than using StringBuffer.toString().equals() or StringBuffer.toS ==.
401	Unused local variable	MAJOR	Detects when a local variable is declared and/or assigned, but not used.
402	Use Index Of Char	MAJOR	Use String.indexOf(char) when checking for the index of a single character; it executes faster.
403	Unused Modifier	INFO	Fields in interfaces are automatically public static final, and methods are public abstract. Classes or interfaces nested in an interface are automatically public and static (all nested interfaces are automatically static). For historical reasons, modif accepted by the compiler, but are superfluous.
404	Security - Array is stored directly	CRITICAL	Constructors and methods receiving arrays should clone objects and store the copy. This prevents that future changes from the user affect the internal functionality.

405	Close Resource	MAJOR	Ensure that resources (like Connection, Statement, and ResultSet objects) are always closed after use. It does this by lookin for code patterned like :
			`Connection c = openConnection();`
			`try {`
			`  // do stuff, and maybe catch something`
			`} finally {`
			`  c.close();`
			`}`

406	Constructor Calls Overridable Method	MAJOR	Calling overridable methods during construction poses a risk of invoking methods on an incompletely constructed object and can be difficult to discern. It may leave the sub-class unable to construct its superclass or forced to replicate
			
			the construction process completely within itself, losing the ability to call super().
			If the default constructor contains a call to an overridable method, the subclass may be completely uninstantiable.
			
			Note that this includes method calls throughout the control flow graph - i.e., if a constructor Foo() calls
			a private method bar() that calls a public method buz(), this denotes a problem.
			
			Example :
			`public class SeniorClass {`
			`  public SeniorClass(){`
			`      toString(); //may throw NullPointerException if overridden`
			`  }`
			`  public String toString(){`
			`    return "IAmSeniorClass";`
			`  }`
			`}`
			`public class JuniorClass extends SeniorClass {`
			`  private String name;`
			`  public JuniorClass(){`
			`    super(); //Automatic call leads to NullPointerException`
			`    name = "JuniorClass";`
			`  }`
			`  public String toString(){`
			`    return name.toUpperCase();`
			`  }`
			`}`

407	Avoid Throwing Null Pointer	MAJOR	Avoid throwing a NullPointerException - it's confusing because most people will assume that the virtual machine threw it.

	Exception		Consider using an IllegalArgumentException instead; this will be clearly seen as a programmer-initiated exception.
408	Avoid Catching NPE	MAJOR	Code should never throw NPE under normal circumstances. A catch block may hide the original error, causing other more su errors in its wake.
409	Avoid Decimal Literals In Big Decimal Constructor	MAJOR	One might assume that new BigDecimal(.1) is exactly equal to .1, but it is actually equal to .100000000000000005551115123 This is so because .1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). in to the constructor is not exactly equal to .1, appearances notwithstanding. The (String) constructor, on the other hand, is p is exactly equal to .1, as one would expect. Therefore, it is generally recommended that the (String) constructor be used in p

410	Idempotent Operations	MAJOR	Avoid idempotent operations - they are have no effect. Example :
			`int x = 2;`
			`x = x;`

411	Simplify Conditional	MAJOR	No need to check for null before an instanceof; the instanceof keyword returns false when given a null argument.
412	Avoid Array Loops	MAJOR	Instead of copying data between two arrays, use System.arrayCopy method
413	Unnecessary Local Before Return	MAJOR	Avoid unnecessarily creating local variables
414	Useless Operation On Immutable	CRITICAL	An operation on an Immutable object (BigDecimal or BigInteger) won't change the object itself. The result of the operation is a new object. Therefore, ignoring the operation result is an error.
415	Instantiation To Get Class	MAJOR	Avoid instantiating an object just to call getClass() on it; use the .class public member instead. Example : replace `Class c = new String().getClass();` with `Class c = String.class;`
416	Unused Null Check In Equals	MAJOR	After checking an object reference for null, you should invoke equals() on that object rather than passing it to another object's equals() method.
417	Inefficient String Buffering	MAJOR	Avoid concatenating non literals in a StringBuffer constructor or append().
418	Broken Null Check	CRITICAL	The null check is broken since it will throw a Nullpointer itself. The reason is that a method is called on the object when it is null. It is likely that you used

419	Redundant pairs of parentheses should be removed	MAJOR	The use of parentheses, even those not required to enforce a desired order of operations, can clarify the intent behind a piece of code. But
			redundant pairs of parentheses could be misleading, and should be removed.
			Noncompliant Code Example
			`int x = (y / 2 + 1);   //Compliant even if the parenthesis are ignored by the compiler`
			
			`if (a && ((x+y > 0))) {  // Noncompliant`
			`  //...`
			`}`
			
			`return ((x + 1));  // Noncompliant`
			Compliant Solution
			`int x = (y / 2 + 1);`
			
			`if (a && (x+y > 0)) {`
			`  //...`
			`}`
			
			`return (x + 1);`

			Calling `System.gc()` or `Runtime.getRuntime().gc()` is a bad idea for a simple reason: there is no way to know exactly what

420	Execution of the Garbage Collector should be triggered only by the JVM	CRITICAL	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.

421	Thread.run() should not be called directly	MAJOR	The purpose of the `Thread.run()` method is to execute code in a separate, dedicated thread. Calling this method directly doesn't make sense because it causes its code to be executed in the current thread.
			To get the expected behavior, call the `Thread.start()` method instead.
			Noncompliant Code Example
			`Thread myThread = new Thread(runnable);`
			`myThread.run(); // Noncompliant`
			Compliant Solution
			`Thread myThread = new Thread(runnable);`
			`myThread.start(); // Compliant`
			See
			- MITRE, CWE-572 - Call to Thread run() instead of start()
			- CERT THI00-J. - Do not invoke Thread.run()

			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**

<table>
<tr>
<td>422</td>
<td>"switch" statements should not contain non-case labels</td>
<td>BLOCKER</td>
<td>

```
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 < X ; i++) {  // Noncompliant; the code is correct and behaves as expec
         /* ... /
        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;

    / ... */
}
```

See

- MISRA C:2004, 15.0 - The MISRA C *switch* syntax shall be used.
- MISRA C++:2008, 6-4-3 - A switch statement shall be a well-formed switch statement.
- MISRA C:2012, 16.1 - All switch statements shall be well-formed

</td>
</tr>
</table>

423	Loops should not contain more than a single "break" or "continue" statement	MINOR	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 <= 10; i++) {     // Noncompliant - 2 continue - one might be tempted to add so`
			`  if (i % 2 == 0) {`
			`    continue;`
			`  }`
			
			`  if (i % 3 == 0) {`
			`    continue;`
			`  }`
			
			`  System.out.println("i = " + i);`
			`}`

424	Enumeration should not be implemented	MAJOR	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 Enumera
			
			Noncompliant Code Example
			
			`public class MyClass implements Enumeration {  // Non-Compliant`
			`  /* ... /`
			`}`
			
			Compliant Solution
			
			`public class MyClass implements Iterator {     // Compliant`
			`  / ... */`
			`}`

			The `switch` statement should be used only to clearly define some new branches in the control flow. As soon as a `case` clause contains too many statements this highly decreases the readability of the overall control flow statement. In such

<table>
<tr>
<td>425</td>
<td>"switch case" clauses should not have too many lines of code</td>
<td>MAJOR</td>
<td>

case, the content of the `case` clause should be extracted into a dedicated method.

Noncompliant Code Example

With the default threshold of 5:

```
switch (myVariable) {
  case 0: // 6 lines till next case
    methodCall1("");
    methodCall2("");
    methodCall3("");
    methodCall4("");
    break;
  case 1:
  ...
}
```

Compliant Solution

```
switch (myVariable) {
  case 0:
    doSomething()
    break;
  case 1:
  ...
}
...
private void doSomething(){
    methodCall1("");
    methodCall2("");
    methodCall3("");
    methodCall4("");
}
```

</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>

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

</td>
</tr>
</table>

<table>
<tr>
<td>426</td>
<td>"equals(Object obj)" should be overridden along with the "compareTo(T obj)" method</td>
<td>MINOR</td>
<td>

6 it has relied on `equals()` .

Noncompliant Code Example

```
public class Foo implements Comparable<Foo> {
  @Override
  public int compareTo(Foo foo) { /* ... / }      // Noncompliant as the equals(Object obj) metho
}
```

Compliant Solution

```
public class Foo implements Comparable<Foo> {
  @Override
  public int compareTo(Foo foo) { / ... / }      // Compliant

  @Override
  public boolean equals(Object obj) { / ... */ }
}
```

</td>
</tr>
<tr>
<td>427</td>
<td>Constants should not be defined in interfaces</td>
<td>CRITICAL</td>
<td>

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 commi
> 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;
}
```

</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td></td>
<td>

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;
}
```

</td>
</tr>
<tr>
<td>428</td>
<td>The Object.finalize() method should not be called</td>
<td>MAJOR</td>
<td>

According to the official javadoc documentation, this Object.finalize() is called by the garbage collector on an object when garbage collection
determines that there are no more references to the object. Calling this method explicitly breaks this contract and so is misleading.

Noncompliant Code Example

```
public void dispose() throws Throwable {
  this.finalize();                        // Noncompliant
}
```

See

- MITRE, CWE-586 - Explicit Call to Finalize()
- CERT, MET12-J. - Do not use finalizers

</td>
</tr>
<tr>
<td>429</td>
<td>Primitive wrappers should not be instantiated only for "toString" or "compareTo" calls</td>
<td>MINOR</td>
<td>

Creating temporary primitive wrapper objects only for `String` conversion or the use of the `compareTo` method is
inefficient.

Instead, the static `toString()` or `compare` method of the primitive wrapper class should be used.

Noncompliant Code Example

```
new Integer(myInteger).toString();  // Noncompliant
```

</td>
</tr>
</table>

			Compliant Solution `Integer.toString(myInteger);        // Compliant`

430	Case insensitive string comparisons should be made without intermediate upper or lower casing	MINOR	Using `toLowerCase()` or `toUpperCase()` to make case insensitive comparisons is inefficient because it requires the creation of temporary, intermediate `String` objects.
			Noncompliant Code Example
			`boolean result1 = foo.toUpperCase().equals(bar);             // Noncompliant`
			`boolean result2 = foo.equals(bar.toUpperCase());             // Noncompliant`
			`boolean result3 = foo.toLowerCase().equals(bar.LowerCase()); // Noncompliant`
			Compliant Solution
			`boolean result = foo.equalsIgnoreCase(bar);                  // Compliant`

431	Collection.isEmpty() should be used to test for emptiness	MINOR	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()) {`
			`  / ... */`
			`}`

			Appending `String.valueOf()` to a `String` decreases the code readability.

<table>
<tr>
<td>432</td>
<td>String.valueOf() should not be appended to a String</td>
<td>MINOR</td>
<td>

The argument passed to `String.valueOf()` should be directly appended instead.

Noncompliant Code Example

```
public void display(int i){
  System.out.println("Output is " + String.valueOf(i));    // Noncompliant
}
```

Compliant Solution

```
public void display(int i){
  System.out.println("Output is " + i);                    // Compliant
}
```

</td>
</tr>
<tr>
<td>433</td>
<td>Comments should not be located at the end of lines of code</td>
<td>MINOR</td>
<td>

This rule verifies that single-line comments are not located at the ends of lines of code. The main idea behind this rule is that in order to be
really readable, trailing comments would have to be properly written and formatted (correct alignment, no interference with the visual structure of
the code, not too long to be visible) but most often, automatic code formatters would not handle this correctly: the code would end up less readable.
Comments are far better placed on the previous empty line of code, where they will always be visible and properly formatted.

Noncompliant Code Example

```
int a1 = b + c; // This is a trailing comment that can be very very long
```

Compliant Solution

```
// This very long comment is better placed before the line of code
int a2 = b + c;
```

</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>

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

</td>
</tr>
</table>

434	Switch cases should end with an unconditional "break" statement	BLOCKER

```
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 gr
  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

- MISRA C:2004, 15.0 - The MISRA C *switch* syntax shall be used.
- MISRA C:2004, 15.2 - An unconditional break statement shall terminate every non-empty switch clause
- MISRA C++:2008, 6-4-3 - A switch statement shall be a well-formed switch statement.
- MISRA C++:2008, 6-4-5 - An unconditional throw or break statement shall terminate every non-empty switch-clause

<table>
<tr><td></td><td></td><td></td><td>

- MISRA C:2012, 16.1 - All switch statements shall be well-formed
- MISRA C:2012, 16.3 - An unconditional break statement shall terminate every switch-clause
- 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, MSC18-CPP. - 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

</td></tr>
<tr><td>435</td><td>Unused "private" methods should be removed</td><td>MAJOR</td><td>

`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 di
 class.
  public static void doSomething(){
    Foo foo = new Foo();
    ...
  }
  private void unusedPrivateMethod(){...}
  private void writeObject(ObjectOutputStream s){...}  //Compliant, relates to the java serializat
  private void readObject(ObjectInputStream in){...}  //Compliant, relates to the java serializati
}
```

Compliant Solution

```
public class Foo implements Serializable
{
  private Foo(){}     //Compliant, private empty constructor intentionally used to prevent any di
 class.
  public static void doSomething(){
    Foo foo = new Foo();
    ...
  }

  private void writeObject(ObjectOutputStream s){...}  //Compliant, relates to the java serializat

  private void readObject(ObjectInputStream in){...}  //Compliant, relates to the java serializati
}
```

</td></tr>
</table>

Exceptions

This rule doesn't raise any issue on annotated methods.

See

- CERT, MSC07-CPP. - Detect and remove dead code

436	Method parameters, caught exceptions and foreach variables should not be reassigned	MINOR

While it is technically correct to assign to parameters from within method bodies, it reduces code readability because developers won't be able to
tell whether the original parameter or some temporary variable is being accessed without going through the whole method. Moreover, some developers
might also expect assignments of method parameters to be visible to callers, which is not the case, and this lack of visibility could confuse them.
Instead, all parameters, caught exceptions, and foreach parameters should be treated as `final`.

Noncompliant Code Example

```
class MyClass {
  public String name;

  public MyClass(String name) {
    name = name;                      // Noncompliant - useless identity assignment
  }

  public int add(int a, int b) {
    a = a + b;                        // Noncompliant

    /* additional logic */

    return a;                         // Seems like the parameter is returned as is, what is the poi
  }

  public static void main(String[] args) {
    MyClass foo = new MyClass();
    int a = 40;
    int b = 2;
    foo.add(a, b);                    // Variable "a" will still hold 40 after this call
  }
}
```

Compliant Solution

```
class MyClass {
  public String name;

  public MyClass(String name) {
    this.name = name;               // Compliant
  }

  public int add(int a, int b) {
    return a + b;                   // Compliant
  }

  public static void main(String[] args) {
    MyClass foo = new MyClass();
    int a = 40;
    int b = 2;
    foo.add(a, b);
  }
}
```

See

- MISRA C:2012, 17.8 - A function parameter should not be modified

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.

437	Assignments should not be made from within sub-expressions	MAJOR	`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
			- MISRA C:2004, 13.1 - Assignment operators shall not be used in expressions that yield a Boolean value
			- MISRA C++:2008, 6-2-1 - Assignment operators shall not be used in sub-expressions
			- MISRA C:2012, 13.4 - The result of an assignment operator should not be used
			- 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
			- CERT, EXP19-CPP. - Do not perform assignments in conditional expressions
			- CERT, MSC02-CPP. - Avoid errors of omission

438	Methods should not be named "hashcode" or "equal"	MAJOR	Naming a method `hashcode()` or `equal` is either: - A bug in the form of a typo. Overriding `Object.hashCode()` (note the camelCasing) or `Object.equals` (note the 's' on the end) was meant, and the application does not behave as expected. - Done on purpose. The name however will confuse every other developer, who may not notice the naming difference, or who will think it is a bug. In both cases, the method should be renamed. **Noncompliant Code Example**

			`public int hashcode() { /* ... / }  // Noncompliant`
			`public boolean equal(Object obj) { / ... / }  // Noncompliant`
			Compliant Solution
			`@Override`
			`public int hashCode() { / ... / }`
			`public boolean equals(Object obj) { / ... */ }`

439	Public methods should throw at most one checked exception	MAJOR	Using checked exceptions forces method callers to deal with errors, either by propagating them or by handling them. Throwir exceptions makes them fully part of the API of the method.
			But to keep the complexity for callers reasonable, methods should not throw more than one kind of checked exception.
			Noncompliant Code Example
			`public void delete() throws IOException, SQLException {      // Noncompliant`
			`  /* ... /`
			`}`
			Compliant Solution
			`public void delete() throws SomeApplicationLevelException {`
			`  / ... */`
			`}`
			Exceptions
			Overriding methods are not checked by this rule and are allowed to throw several checked exceptions.

			According to the Java Language Specification: Unnamed packages are provided by the Java platform principally for convenience when developing small or temporary ap or when just beginning development.

440	The default unnamed package should not be used	MINOR	To enforce this best practice, classes located in default package can no longer be accessed from named ones since Java 1.4.
			Noncompliant Code Example
			`public class MyClass { /* ... / }`
			Compliant Solution
			`package org.example;`
			`public class MyClass{ / ... */ }`

441	Class names should comply with a naming convention	MINOR	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. 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 {...}`

			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(){...}`

442	Method names should comply with a naming convention	MINOR	**Compliant Solution**
			`public int doSomething(){...}`
			Exceptions
			Overriding methods are excluded.
			`@Override`
			`public int Do_Something(){...}`

443	Non-constructor methods should not have the same name as the enclosing class	MAJOR	Having a class and some of its methods sharing the same name is misleading, and leaves others to wonder whether it was done that way on purpose, or
			was the methods supposed to be a constructor.
			Noncompliant Code Example
			`public class Foo {`
			`   public Foo() {...}`
			`   public void Foo(String label) {...}  // Noncompliant`
			`}`
			Compliant Solution
			`public class Foo {`
			`   public Foo() {...}`
			`   public void foo(String label) {...}  // Compliant`
			`}`

			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**

<table>
<tr>
<td>444</td>
<td>Empty arrays and collections should be returned instead of null</td>
<td>MAJOR</td>
<td>

```
public static List<Result> 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<Result> 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

</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>

Shadowing fields with a local variable is a bad practice that reduces code readability: it makes it confusing to know whether the field or the variable is being used.

Noncompliant Code Example

</td>
</tr>
</table>

<table>
<tr><th></th><th></th><th></th><th></th></tr>
<tr>
<td>445</td>
<td>Local variables should not shadow class fields</td>
<td>MAJOR</td>
<td>
<pre><code>class Foo {
  public int myField;

  public void doSomething() {
    int myField = 0;
    ...
  }
}</code></pre>

<h2>See</h2>

<ul><li>CERT, DCL51-J. - Do not shadow or obscure identifiers in subscopes</li></ul>
</td>
</tr>
<tr>
<td>446</td>
<td>Exceptions should not be thrown in finally blocks</td>
<td>CRITICAL</td>
<td>
<p>Throwing an exception from within a finally block will mask any exception which was previously thrown in the <code>try</code> or <code>catch</code> block, and the masked's exception message and stack trace will be lost.</p>

<h2>Noncompliant Code Example</h2>

<pre><code>try {
  /* some work which end up throwing an exception /
  throw new IllegalArgumentException();
} finally {
  / clean up /
  throw new RuntimeException();   // Noncompliant; masks the IllegalArgumentException
}</code></pre>

<h2><em>Compliant Solution</em></h2>

<pre><code>try {
  / some work which end up throwing an exception /
  throw new IllegalArgumentException();
} finally {
  / clean up */
}</code></pre>

<h2>See</h2>

<ul><li>CERT, ERR05-J. - Do not let checked exceptions escape from a finally block</li></ul>
</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>When handling a caught exception, the original exception's message and stack trace should be logged or passed forward.</td>
</tr>
</table>

447	Exception handlers should preserve the original exceptions	MAJOR

Noncompliant Code Example

```
try {
  /* ... /
} catch (Exception e) {   // Noncompliant - exception is lost
  LOGGER.info("context");
}

try {
  / ... /
} catch (Exception e) {  // Noncompliant - exception is lost (only message is preserved)
  LOGGER.info(e.getMessage());
}

try {
  / ... /
} catch (Exception e) {  // Noncompliant - original exception is lost
  throw new RuntimeException("context");
}
```

Compliant Solution

```
try {
  / ... /
} catch (Exception e) {
  LOGGER.info(e);  // exception is logged
}

try {
  / ... /
} catch (Exception e) {
  throw new RuntimeException(e);   // exception stack trace is propagated
}

try {
  / ... */
} catch (RuntimeException e) {
  doSomething();
  throw e;  // original exception passed forward
} catch (Exception e) {
  throw new RuntimeException(e);  // Conversion into unchecked exception is also allowed
}
```

Exceptions

`InterruptedException` , `NumberFormatException` , `DateTimeParseException` , `ParseException`
and
`MalformedURLException` exceptions are arguably used to indicate nonexceptional outcomes. Similarly, handling `NoSuchMethodException` is often required when dealing with the Java reflection API.

Because they are part of Java, developers have no choice but to deal with them. This rule does not verify that those particula
exceptions are
correctly handled.

```
int myInteger;
try {
  myInteger = Integer.parseInt(myString);
} catch (NumberFormatException e) {
  // It is perfectly acceptable to not handle "e" here
  myInteger = 0;
}
```

See

- CERT, ERR00-J. - Do not suppress or ignore checked exceptions

448	Exception classes should be immutable	MINOR

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

<table>
<tr>
<td></td>
<td></td>
<td></td>
<td>

```
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;
  }

}
```

</td>
</tr>
<tr>
<td>449</td>
<td>Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply</td>
<td>CRITICAL</td>
<td>

Nested `if` , `for` , `while` , `switch` , and `try` statements are key ingredients for making what's known as "Spaghetti code".

Such code is hard to read, refactor and therefore maintain.

Noncompliant Code Example

With the default threshold of 3:

```
if (condition1) {                  // Compliant - depth = 1
  /* ... /
  if (condition2) {                // Compliant - depth = 2
    / ... /
    for(int i = 0; i < 10; i++) {  // Compliant - depth = 3, not exceeding the limit
      / ... /
      if (condition4) {            // Noncompliant - depth = 4
        if (condition5) {          // Depth = 5, exceeding the limit, but issues are only reporte
          / ... */
        }
        return;
      }
    }
  }
}
```

</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>

Making a `public` constant just `final` as opposed to `static final` leads to duplicating its value for every instance of the class, uselessly increasing the amount of memory required to execute the application.

Further, when a non- `public` , `final` field isn't also `static` , it implies that different instances can have different values. However, initializing a non- `static final` field in its declaration forces every instance to have the same value. So such fields should either be made `static` or initialized in the constructor.

Noncompliant Code Example

</td>
</tr>
</table>

450	Public constants and fields initialized at declaration should be "static final" rather than merely "final"	MINOR	`public class Myclass {`
			`  public final int THRESHOLD = 3;`
			`}`
			Compliant Solution
			`public class Myclass {`
			`  public static final int THRESHOLD = 3;    // Compliant`
			`}`
			Exceptions
			No issues are reported on final fields of inner classes whose type is not a primitive or a String. Indeed according to the Java specification:
			> An inner class is a nested class that is not explicitly or implicitly declared static. Inner classes may not declare static initializers (§8.7)
			> or member interfaces. Inner classes may not declare static members, unless they are compile-time constant fields (§15.28

451	Only static class initializers should be used	MAJOR	Non-static initializers are rarely used, and can be confusing for most developers because they only run when new class instances are created. When
			possible, non-static initializers should be refactored into standard constructors or field initializers.
			Noncompliant Code Example
			`class MyClass {`
			`  private static final Map<String, String> MYMAP = new HashMap<String, String>() {`
			
			`    // Noncompliant - HashMap should be extended only to add behavior, not for initialization`
			`    {`
			`      put("a", "b");`
			`    }`
			
			`  };`
			`}`
			Compliant Solution

			`class MyClass {`
			`  private static final Map<String, String> MYMAP = new HashMap<String, String>();`
			
			`  static {`
			`    MYMAP.put("a", "b");`
			`  }`
			`}`
			
			or using Guava:
			
			`class MyClass {`
			`  // Compliant`
			`  private static final Map<String, String> MYMAP = ImmutableMap.of("a", "b");`
			`}`

452	The signature of "finalize()" should match that of "Object.finalize()"	CRITICAL	`Object.finalize()` is called by the Garbage Collector at some point after the object becomes unreferenced.
			
			In general, overloading `Object.finalize()` is a bad idea because:
			
			- The overload may not be called by the Garbage Collector.
			- Users are not expected to call `Object.finalize()` and will get confused.
			
			But beyond that it's a terrible idea to name a method "finalize" if it doesn't actually override `Object.finalize()`.
			
			Noncompliant Code Example
			
			`public int finalize(int someParameter) {        // Noncompliant`
			`  /* ... /`
			`}`
			
			Compliant Solution
			
			`public int someBetterName(int someParameter) {  // Compliant`
			`  / ... */`
			`}`

			The contract of the `Object.finalize()` method is clear: only the Garbage Collector is supposed to call this method. Making this method public is misleading, because it implies that any caller can use it.

<table>
<tr>
<td>453</td>
<td>"Object.finalize()" should remain protected (versus public) when overriding</td>
<td>CRITICAL</td>
<td>
<h2>Noncompliant Code Example</h2>
<pre>
public class MyClass {

  @Override
  public void finalize() {   // Noncompliant
    /* ... */
  }
}
</pre>
<h2>See</h2>
<ul>
<li>MITRE, CWE-583 - finalize() Method Declared Public</li>
<li>CERT, MET12-J. - Do not use finalizers</li>
</ul>
</td>
</tr>
<tr>
<td>454</td>
<td>"throws" declarations should not be superfluous</td>
<td>MINOR</td>
<td>
<p>An exception in a <code>throws</code> declaration in Java is superfluous if it is:</p>
<ul>
<li>listed multiple times</li>
<li>a subclass of another listed exception</li>
<li>a <code>RuntimeException</code> , or one of its descendants</li>
<li>completely unnecessary because the declared exception type cannot actually be thrown</li>
</ul>
<h2>Noncompliant Code Example</h2>
<pre>
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
</pre>
<h2>Compliant Solution</h2>
<pre>
void foo() throws MyException {}
void bar() throws Throwable {}
void baz() {}
</pre>
</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>
<p>Labels are not commonly used in Java, and many developers do not understand how they work. Moreover, their usage make
the control flow harder to
follow, which reduces the code's readability.</p>
</td>
</tr>
</table>

455 Labels should not be used MAJOR

Noncompliant Code Example

```
int matrix[][] = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};

outer: for (int row = 0; row < matrix.length; row++) {   // Non-Compliant
  for (int col = 0; col < matrix[row].length; col++) {
    if (col == row) {
      continue outer;
    }
    System.out.println(matrix[row][col]);                // Prints the elements under the diagona
  }
}
```

Compliant Solution

```
for (int row = 1; row < matrix.length; row++) {          // Compliant
  for (int col = 0; col < row; col++) {
    System.out.println(matrix[row][col]);                // Also prints 4, 7 and 8
  }
}
```

456 "for" loop stop conditions should be invariant MAJOR

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 < 10; i++) {
  ...
  i = i - 1; // Noncompliant; counter updated in the body of the loop
  ...
}
```

Compliant Solution

```
for (int i = 0; i < 10; i++) {...}
```

See

- MISRA C:2004, 13.6 - Numeric variables being used within a *for* loop for iteration counting shall not be modified in the body of the loop.
- MISRA C++:2008, 6-5-3 - The *loop-counter* shall not be modified within *condition* or *statement*.

"equals(Object obj)" and

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

457	"hashCode()" should be overridden in pairs	MINOR	

```
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

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. When the `switch` covers all current values of an `enum` - and especially when it
doesn't - a `default` case should still be used because there is no guarantee that the `enum` won't be extended.

Noncompliant Code Example

<table>
<tr>
<td>458</td>
<td>"switch" statements should end with "default" clauses</td>
<td>CRITICAL</td>
<td>

```
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;
}
```

</td>
</tr>
</table>

See

- MISRA C:2004, 15.0 - The MISRA C *switch* syntax shall be used.
- MISRA C:2004, 15.3 - The final clause of a switch statement shall be the default clause
- MISRA C++:2008, 6-4-3 - A switch statement shall be a well-formed switch statement.
- MISRA C++:2008, 6-4-6 - The final clause of a switch statement shall be the default-clause
- MISRA C:2012, 16.1 - All switch statements shall be well-formed
- MISRA C:2012, 16.4 - Every *switch* statement shall have a *default* label
- MISRA C:2012, 16.5 - A *default* label shall appear as either the first or the last *switch label* of a *switch* statement
- MITRE, CWE-478 - Missing Default Case in Switch Statement
- CERT, MSC01-C. - Strive for logical completeness
- CERT, MSC01-CPP. - Strive for logical completeness

459

"equals" method overrides should accept "Object" parameters

MAJOR

"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 && 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`

460	Modifiers should be declared in the correct order	MINOR	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`
			`}`

461	Classes should not be too complex	CRITICAL	The Cyclomatic Complexity is measured by the number of `&&` and

462	Throwable and Error should not be caught	MAJOR	`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

			`Cloneable` is the marker `Interface` that indicates that `clone()` may be called on an object. Overriding `clone()` without implementing `Cloneable` can be useful if you want to control how subclasses clone themselves, but otherwise, it's probably a mistake.

463	Classes that override "clone" should be "Cloneable" and call "super.clone()"	MINOR

The usual convention for `Object.clone()` according to Oracle's Javadoc is:

1. `x.clone() != x`
2. `x.clone().getClass() == x.getClass()`
3. `x.clone().equals(x)`

Obtaining the object that will be returned by calling `super.clone()` helps to satisfy those invariants:

1. `super.clone()` returns a new object instance
2. `super.clone()` returns an object of the same type as the one `clone()` was called on
3. `Object.clone()` performs a shallow copy of the object's state

Noncompliant Code Example

```
class BaseClass {  // Noncompliant; should implement Cloneable
  @Override
  public Object clone() throws CloneNotSupportedException {    // Noncompliant; should return the
    return new BaseClass();
  }
}

class DerivedClass extends BaseClass implements Cloneable {
  /* Does not override clone() /

  public void sayHello() {
    System.out.println("Hello, world!");
  }
}

class Application {
  public static void main(String[] args) throws Exception {
    DerivedClass instance = new DerivedClass();
    ((DerivedClass) instance.clone()).sayHello();              // Throws a ClassCastException bec
  }
}
```

Compliant Solution

```
class BaseClass implements Cloneable {
  @Override
  public Object clone() throws CloneNotSupportedException {    // Compliant
    return super.clone();
  }
}

class DerivedClass extends BaseClass implements Cloneable {
  / Does not override clone() */

  public void sayHello() {
    System.out.println("Hello, world!");
  }
}

class Application {
  public static void main(String[] args) throws Exception {
    DerivedClass instance = new DerivedClass();
    ((DerivedClass) instance.clone()).sayHello();              // Displays "Hello, world!" as expe
  }
}
```

See

- MITRE, CWE-580 - clone() Method Without super.clone()
- CERT, MET53-J. - Ensure that the clone() method calls super.clone()

464 Methods should not be empty CRITICAL

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

<table>
<tr>
<td></td>
<td></td>
<td></td>
<td>
<pre><code>@Override
public void doSomething() {
  // Do nothing because of X and Y.
}

@Override
public void doSomethingElse() {
  throw new UnsupportedOperationException();
}</code></pre>
<h2>Exceptions</h2>
<p>Default (no-argument) constructors are ignored when there are other constructors in the class, as are empty methods in abstract classes.</p>
<pre><code>public abstract class Animal {
  void speak() { // default implementation ignored
  }
}</code></pre>
</td>
</tr>
<tr>
<td>465</td>
<td>Overriding methods should do more than simply call the same method in the super class</td>
<td>MINOR</td>
<td>
<p>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 <code>final</code> overriding methods, where the effect is to lock in the parent class behavior.
This rule ignores such
overrides of <code>equals</code>, <code>hashCode</code> and <code>toString</code>.</p>
<h2>Noncompliant Code Example</h2>
<pre><code>public void doSomething() {
  super.doSomething();
}

@Override
public boolean isLegal(Action action) {
  return super.isLegal(action);
}</code></pre>
<h2>Compliant Solution</h2>
<pre><code>@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
  return super.getId();
}</code></pre>
</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Anonymous classes and lambdas (with Java 8) are a very convenient and compact way to inject a behavior without having</td>
</tr>
</table>

466	Lambdas and anonymous classes should not have too many lines of code	MAJOR	to create a dedicated class. But those anonymous inner classes and lambdas should be used only if the behavior to be injected can be defined in a few lines of code, otherwise the source code can quickly become unreadable.

467	Useless imports should be removed	MINOR	The imports part of a file should be handled by the Integrated Development Environment (IDE), not manually by the develope
			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 importe`
			`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.

			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 h references should be refactored. **Noncompliant Code Example**

```
class MyClass {

  @Deprecated
  public void foo1() {
  }

  /
    * @deprecated
    */
  public void foo2() {    // Noncompliant
  }

}
```

468	Deprecated elements should have both the annotation and the Javadoc tag	MAJOR

Compliant Solution

```
class MyClass {

  /
    * @deprecated (when, why, refactoring advice...)
    /
  @Deprecated
  public void foo1() {
  }

  /*
    * @deprecated (when, why, refactoring advice...)
    /
  @Deprecated
  public void foo2() {
  }

}
```

Exceptions

The members and methods of a deprecated class or interface are ignored by this rule. The classes and interfaces themselve 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();

}
```

469	"switch" statements should have at least 3 "case" clauses	MINOR	

`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();
}
```

See

- MISRA C:2004, 15.5 - Every switch statement shall have at least one case clause.
- MISRA C++:2008, 6-4-8 - Every switch statement shall have at least one case-clause.
- MISRA C:2012, 16.6 - Every switch statement shall have at least two switch-clauses

470	The Object.finalize() method should not be overriden	MAJOR	

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

471	IP addresses should not be hardcoded	MINOR	Hardcoding an IP address into source code is a bad idea for several reasons:
			- a recompile is required if the address changes
			- it forces the same address to be used in every environment (dev, sys, qa, prod)
			- it places the responsibility of setting the value to use in production on the shoulders of the developer
			- it allows attackers to decompile the code and thereby discover a potentially sensitive address
			Noncompliant Code Example
			`String ip = "127.0.0.1";`
			`Socket socket = new Socket(ip, 6667);`
			Compliant Solution
			`String ip = System.getProperty("myapplication.ip");`
			`Socket socket = new Socket(ip, 6667);`
			See
			- CERT, MSC03-J. - Never hard code sensitive information

			Loggers should be:

			- `private` : not accessible outside of their parent classes. If another class needs to log something, it should instantiate its own logger.
			- `static` : not dependent on an instance of a class (an object). When logging something, contextual information can of course be provided in the messages but the logger should be created at class level to prevent creating a logger along with each ob
			- `final` : created once and only once per class.
			Noncompliant Code Example
			With a default regular expression of `LOG(?:GER)?` :
			`public Logger logger = LoggerFactory.getLogger(Foo.class);  // Noncompliant`
			Compliant Solution
			`private static final Logger LOGGER = LoggerFactory.getLogger(Foo.class);`
			Exceptions
			Variables of type `org.apache.maven.plugin.logging.Log` are ignored.

472	Loggers should be "private static final" and should share a naming convention	MINOR	

	Utility classes should not have		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;`
			`  }`
			`}`

<table>
<tr>
<td>473</td>
<td>public constructors</td>
<td>MAJOR</td>
<td>

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.

</td>
</tr>
<tr>
<td>474</td>
<td>Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"</td>
<td>MINOR</td>
<td>

The purpose of the Java Collections API is to provide a well defined hierarchy of interfaces in order to hide implementation details.

Implementing classes must be used to instantiate new collections, but the result of an instantiation should ideally be stored in a variable whose
type is a Java Collection interface.

This rule raises an issue when an implementation class:

- is returned from a `public` method.
- is accepted as an argument to a `public` method.
- is exposed as a `public` member.

Noncompliant Code Example

```
public class Employees {
  private HashSet<Employee> employees = new HashSet<Employee>();  // Noncompliant - "employees" sh
 "Set" rather than "HashSet"

  public HashSet<Employee> getEmployees() {                       // Noncompliant
    return employees;
  }
}
```

Compliant Solution

</td>
</tr>
</table>

<table>
<tr><th></th><th></th><th></th><th></th></tr>
<tr><td></td><td></td><td></td><td><pre>
public class Employees {
  private Set<Employee> employees = new HashSet<Employee>();   // Compliant

  public Set<Employee> getEmployees() {   // Compliant
    return employees;
  }
}
</pre></td></tr>
<tr><td>475</td><td>Sections of code should not be "commented out"</td><td>MAJOR</td><td>
<p>Programmers should not comment out code as it bloats programs and reduces readability.</p>
<p>Unused code should be deleted and can be retrieved from source control history if required.</p>
<h2>See</h2>
<ul>
<li>MISRA C:2004, 2.4 - Sections of code should not be "commented out".</li>
<li>MISRA C++:2008, 2-7-2 - Sections of code shall not be "commented out" using C-style comments.</li>
<li>MISRA C++:2008, 2-7-3 - Sections of code should not be "commented out" using C++ comments.</li>
<li>MISRA C:2012, Dir. 4.4 - Sections of code should not be "commented out"</li>
</ul>
</td></tr>
<tr><td>476</td><td>Octal values should not be used</td><td>BLOCKER</td><td>
<p>Integer literals starting with a zero are octal rather than decimal values. While using octal values is fully supported, most developers do not
have experience with them. They may not recognize octal values as such, mistaking them instead for decimal values.</p>
<h2>Noncompliant Code Example</h2>
<pre>
int myNumber = 010;   // Noncompliant. myNumber will hold 8, not 10 - was this really expected?
</pre>
<h2>Compliant Solution</h2>
<pre>
int myNumber = 8;
</pre>
<h2>See</h2>
<ul>
<li>MISRA C:2004, 7.1 - Octal constants (other than zero) and octal escape sequences shall not be used.</li>
<li>MISRA C++:2008, 2-13-2 - Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used</li>
</ul>
</td></tr>
</table>

			- MISRA C:2012, 7.1 - Octal constants shall not be used - CERT, DCL18-C. - Do not begin integer constants with 0 when specifying a decimal value - CERT, DCL50-J. - Use visually distinct identifiers

477	Boolean literals should not be redundant	MINOR	Redundant Boolean literals should be removed from expressions to improve readability.
			Noncompliant Code Example
			`if (booleanMethod() == true) { /* ... / }`
			`if (booleanMethod() == false) { / ... */ }`
			`if (booleanMethod()`

478	Return of boolean expressions should not be wrapped into an "if-then-else" statement	MINOR	Return of boolean literal statements wrapped into `if-then-else` ones should be simplified.
			Noncompliant Code Example
			`if (expression) {`
			`  return true;`
			`} else {`
			`  return false;`
			`}`
			Compliant Solution
			`return expression;`

			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**

479	Class variable fields should not have public accessibility	MINOR

```
public class MyClass {

  public static final int SOMECONSTANT = 0;     // Compliant - constants are not checked

  public String firstName;                       // Noncompliant

}
```

Compliant Solution

```
public class MyClass {

  public static final int SOMECONSTANT = 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

Shared coding conventions make it possible for a team to collaborate efficiently.

This rule makes it mandatory to place closing curly braces on the same line as the next `else` , `catch` or `finally` keywords.

Noncompliant Code Example

480	Close curly brace and the next "else", "catch" and "finally" keywords should be located on the same line	MINOR	

```
public void myMethod() {
  if(something) {
    executeTask();
  } else if (somethingElse) {
    doSomethingElse();
  }
  else {                                    // Noncompliant
     generateError();
  }

  try {
    generateOrder();
  } catch (Exception e) {
    log(e);
  }
  finally {                                 // Noncompliant
    closeConnection();
  }
}
```

Compliant Solution

```
public void myMethod() {
  if(something) {
    executeTask();
  } else if (somethingElse) {
    doSomethingElse();
  } else {
     generateError();
  }

  try {
    generateOrder();
  } catch (Exception e) {
    log(e);
  } finally {
    closeConnection();
  }
}
```

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");
}
```

481	Generic exceptions should never be thrown	MAJOR	**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

482	Collapsible "if" statements should be merged	MAJOR	Merging collapsible `if` statements increases the code's readability.
			Noncompliant Code Example
			`if (file != null) {`
			`  if (file.isFile()`

483	Expressions should not be too complex	CRITICAL	The complexity of an expression is defined by the number of `&&` ,
			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. `;;` .

<table>
<tr>
<td>484</td>
<td>Empty statements should be removed</td>
<td>MINOR</td>
<td>

Noncompliant Code Example

```
void doSomething() {
  ;                                                          // Noncompliant - was used as a kind of
}

void doSomethingElse() {
  System.out.println("Hello, world!");;                        // Noncompliant - double ;
  ...
  for (int i = 0; i < 3; System.out.println(i), i++);        // Noncompliant - Rarely, they are use
 body of a loop. It is a bad practice to have side-effects outside of the loop body
  ...
}
```

Compliant Solution

```
void doSomething() {}

void doSomethingElse() {
  System.out.println("Hello, world!");
  ...
  for (int i = 0; i < 3; i++){
    System.out.println(i);
  }
  ...
}
```

See

- MISRA C:2004, 14.3 - Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that
the first character following the null statement is a white-space character.

- MISRA C++:2008, 6-2-3 - Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided
that the first character following the null statement is a white-space character.

- CERT, MSC12-C. - Detect and remove code that has no effect or is never executed

- CERT, MSC12-CPP. - Detect and remove code that has no effect

- 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

</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The cyclomatic complexity of methods should not exceed a defined threshold.</td>
</tr>
</table>

<table>
<tr>
<td>485</td>
<td>Methods should not be too complex</td>
<td>CRITICAL</td>
<td>
<p>Complex code can perform poorly and will in any case be difficult to understand and therefore to maintain.</p>
<h2>Exceptions</h2>
<p>While having a large number of fields in a class may indicate that it should be split, this rule nonetheless ignores high complexity in
<code>equals</code> and <code>hashCode</code> methods.</p>
</td>
</tr>
<tr>
<td>486</td>
<td>The members of an interface declaration or class should appear in a pre-defined order</td>
<td>MINOR</td>
<td>
<p>According to the Java Code Conventions as defined by Oracle, the parts of a class or interface declaration should appear in the following order in the source files:</p>
<ul>
<li>Class and instance variables</li>
<li>Constructors</li>
<li>Methods</li>
</ul>
<h2>Noncompliant Code Example</h2>
<pre>
public class Foo {
  private int field = 0;
  isTrue() {...}
  public Foo() {...}   // Noncompliant, constructor defined after method
  public static final int OPEN = 4; // Noncompliant, variable defined after method
}
</pre>
<h2>Compliant Solution</h2>
<pre>
public class Foo{ // Compliant
  public static final int OPEN = 4;
  private int field = 0;
  public Foo() {...}
  public boolean isTrue() {...}
}
</pre>
</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>
<p>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.</p>
<h2>Noncompliant Code Example</h2>
<p>With a maximum number of 4 parameters:</p>
</td>
</tr>
</table>

487	Methods should not have too many parameters	MAJOR	`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` may have a lot of parameters, encapsulation being possible. Such methods are therefore ignored.

488	Strings literals should be placed on the left side when checking for equality	MINOR	It is preferable to place string literals on the left-hand side of an `equals()` or `equalsIgnoreCase()` method call.
			This prevents null pointer exceptions from being raised, as a string literal can never be null by definition.
			Noncompliant Code Example
			`String myString = null;`
			`System.out.println("Equal? " + myString.equals("foo"));                        // Noncompliant; wi`
			`System.out.println("Equal? " + (myString != null && myString.equals("foo")));  // Noncompliant; nu`
			` be removed`
			Compliant Solution
			`System.out.println("Equal?" + "foo".equals(myString));                         // properly deals w`

			This rule is meant to be used as a way to track code which is marked as being deprecated. Deprecated code should eventua be removed. **Noncompliant Code Example**

<table>
<tr>
<td>489</td>
<td>Deprecated code should be removed</td>
<td>INFO</td>
<td><pre>
class Foo {
  /**
   * @deprecated
   */
  public void foo() {   // Noncompliant
  }

  @Deprecated   // Noncompliant
  public void bar() {
  }

  public void baz() {   // Compliant
  }
}
</pre></td>
</tr>
<tr>
<td>490</td>
<td>Nested blocks of code should not be left empty</td>
<td>MAJOR</td>
<td><p>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.</p>
<h2>Noncompliant Code Example</h2>
<pre>
for (int i = 0; i < 42; i++){} // Empty on purpose or missing piece of code ?
</pre>
<h2>Exceptions</h2>
<p>When a block contains a comment, this block is not considered to be empty unless it is a <code>synchronized</code> block. <code>synchronized</code> blocks are still considered empty even with comments because they can still affect program flow.</p></td>
</tr>
<tr>
<td>491</td>
<td>Track uses of "FIXME" tags</td>
<td>MAJOR</td>
<td><p><code>FIXME</code> tags are commonly used to mark places where a bug is suspected, but which the developer wants to deal with later.</p>
<p>Sometimes the developer will not have the time or will simply forget to get back to that tag.</p>
<p>This rule is meant to track those tags and to ensure that they do not go unnoticed.</p>
<h2>Noncompliant Code Example</h2>
<pre>
int divide(int numerator, int denominator) {
  return numerator / denominator;   // FIXME denominator value might be 0
}
</pre>
<h2>See</h2>
<ul>
<li>MITRE, CWE-546 - Suspicious Comment</li>
</ul></td>
</tr>
</table>

492	Track uses of "TODO" tags	INFO	`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

493	Statements should be on separate lines	MAJOR	For better readability, do not put more than one statement on a single line.
			Noncompliant Code Example
			`if(someCondition) doSomething();`
			Compliant Solution
			`if(someCondition) {`
			`  doSomething();`
			`}`

			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**

494	Array designators "[]" should be on the type, not the variable	MINOR	`int matrix[][];   // Noncompliant`
			`int[] matrix[];   // Noncompliant`
			Compliant Solution
			`int[][] matrix;   // Compliant`

495	Package names should comply with a naming convention	MINOR	Shared coding conventions allow teams to collaborate efficiently. This rule checks that all package names match a provided regular expression.
			Noncompliant Code Example
			With the default regular expression `^[a-z]+(.[a-z][a-z0-9])$` :
			`package org.exAmple; // Noncompliant`
			Compliant Solution
			`package org.example;`

496	Nested code blocks should not be used	MINOR	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);
}
```

497	Control structures should use curly braces	CRITICAL

While not technically incorrect, the omission of curly braces can be misleading, and may lead to the introduction of errors during maintenance.

Noncompliant Code Example

```
if (condition)  // Noncompliant
  executeSomething();
```

Compliant Solution

```
if (condition) {
  executeSomething();
}
```

See

- MISRA C:2004, 14.8 - The statement forming the body of a switch, while, do ... while or for statement shall be a compou statement
- MISRA C:2004, 14.9 - An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a
  compound statement, or another if statement
- MISRA C++:2008, 6-3-1 - The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement
- MISRA C++:2008, 6-4-1 - An if (condition) construct shall be followed by a compound statement. The else keyword shal be followed by either a
  compound statement, or another if statement
- MISRA C:2012, 15.6 - The body of an iteration-statement or a selection-statement shall be a compound-statement

			- CERT, EXP19-C. - Use braces for the body of an if, for, or while statement - CERT, EXP52-J. - Use braces for the body of an if, for, or while statement

498	Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	MAJOR	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 fro 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() {...}`

			Sharing some coding conventions is a key point to make it possible for a team to efficiently collaborate. This rule makes it mandatory to place open curly braces at the end of lines of code.

499	An open curly brace should be located at the end of a line	MINOR	**Noncompliant Code Example**
			`if(condition)`
			`{`
			`  doSomething();`
			`}`
			Compliant Solution
			`if(condition) {`
			`  doSomething();`
			`}`
			Exceptions
			When blocks are inlined (left and right curly braces on the same line), no issue is triggered.
			`if(condition) {doSomething();}`

500	A close curly brace should be located at the beginning of a line	MINOR	Shared coding conventions make it possible for a team to efficiently collaborate. This rule makes it mandatory to place a close curly brace at the beginning of a line.
			Noncompliant Code Example
			`if(condition) {`
			`  doSomething();}`
			Compliant Solution
			`if(condition) {`
			`  doSomething();`
			`}`
			Exceptions
			When blocks are inlined (open and close curly braces on the same line), no issue is triggered.

			`if(condition) {doSomething();}`
501	Try-catch blocks should not be nested	MAJOR	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.

502	Type parameter names should comply with a naming convention	MINOR	Shared naming conventions make it possible for a team to collaborate efficiently. Following the established convention of single-letter type
			parameter names helps users and maintainers of your code quickly see the difference between a type parameter and a poorl named class.
			
			This rule check that all type parameter names match a provided regular expression. The following code snippets use the default regular
			expression.
			
			Noncompliant Code Example
			
			`public class MyClass<TYPE> { // Noncompliant`
			`  <TYPE> void method(TYPE t) { // Noncompliant`
			`  }`
			`}`
			
			Compliant Solution
			
			`public class MyClass<T> {`
			`  <T> void method(T t) {`
			`  }`
			`}`

			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**

503	Jump statements should not occur in "finally" blocks	MAJOR

```
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 < 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 < 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

504	Classes from "sun.*" packages should not be used	MAJOR	Classes in the `sun.` *or* `com.sun.` packages are considered implementation details, and are not part of the Java API.
			They can cause problems when moving to new versions of Java because there is no backwards compatibility guarantee. Sim they can cause problems when moving to a different Java vendor, such as OpenJDK.
			Such classes are almost always wrapped by Java API classes that should be used instead.
			Noncompliant Code Example
			`import com.sun.jna.Native;     // Noncompliant`
			`import sun.misc.BASE64Encoder; // Noncompliant`

505	Local variable and method parameter names should comply with a naming convention	MINOR	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that all local variable and function parameter names match a 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 = 0; i < limit; i++) {  // Compliant
  // ...
}
```

506

Future keywords should not be used as names

BLOCKER

Through Java's evolution keywords have been added. While code that uses those words as identifiers may be compilable un
older versions of Java,
it will not be under modern versions.

Following keywords are marked as invalid identifiers

Keyword	Added
`_`	9
`enum`	5.0

`assert` and `strictfp` are another example of valid identifiers which became keywords in later
versions, however as
documented in SONARJAVA-285, it is not easily possible to support parsing of the code for such old versions, therefore they
are not supported by this
rule.

Noncompliant Code Example

```
public void doSomething() {
  int enum = 42;            // Noncompliant
  String _ = "";   // Noncompliant
}
```

Compliant Solution

```
public void doSomething() {
  int magic = 42;
}
```

<table>
<tr><th></th><th></th><th></th><th></th></tr>
<tr><td>507</td><td>Field names should comply with a naming convention</td><td>MINOR</td><td>

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;
}
```

</td></tr>
<tr><td>508</td><td>Useless "if(true) {...}" and "if(false){...}" blocks should be removed</td><td>MAJOR</td><td>

`if` statements with conditions that are always false have the effect of making blocks of code non-functional. `if`
statements with conditions that are always true are completely redundant, and make the code less readable.

There are three possible causes for the presence of such code:

- An if statement was changed during debugging and that debug code has been committed.
- Some value was left unset.
- Some logic is not doing what the programmer thought it did.

In any of these cases, unconditional `if` statements should be removed.

Noncompliant Code Example

</td></tr>
</table>

```
if (true) {
  doSomething();
}
...
if (false) {
  doSomethingElse();
}

if (2 < 3 ) { ... }  // Noncompliant; always false

int i = 0;
int j = 0;
// ...
j = foo();

if (j > 0 && i > 0) { ... }  // Noncompliant; always false - i never set after initialization

boolean b = true;
//...
if (b
```

509	Exception types should not be tested using "instanceof" in catch blocks	MAJOR	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

Shared coding conventions allow teams to collaborate efficiently. This rule checks that all constant names match a provided regular expression.

Noncompliant Code Example

<table>
<tr>
<td>510</td>
<td>Constant names should comply with a naming convention</td>
<td>CRITICAL</td>
<td>

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;
}
```

</td>
</tr>
<tr>
<td>511</td>
<td>String literals should not be duplicated</td>
<td>CRITICAL</td>
<td>

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 ti
  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 cha
  return "";                                       // Compliant - literal "" has less than 5 chara
}
```

Compliant Solution

</td>
</tr>
</table>

<table>
<tr><th></th><th></th><th></th><th></th></tr>
<tr><td></td><td></td><td></td><td>
<pre><code>private static final String ACTION1 = "action1"; // Compliant

public void run() {
  prepare(ACTION1);   // Compliant
  execute(ACTION1);
  release(ACTION1);
}
</code></pre>
<h2>Exceptions</h2>
<p>To prevent generating some false-positives, literals having less than 5 characters are excluded.</p>
</td></tr>
<tr><td>512</td><td>Interface names should comply with a naming convention</td><td>MINOR</td><td>
<p>Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that all
interface names match a provided regular expression.</p>
<h2>Noncompliant Code Example</h2>
<p>With the default regular expression <code>^[A-Z][a-zA-Z0-9]*$</code> :</p>
<pre><code>public interface myInterface {...} // Noncompliant
</code></pre>
<h2>Compliant Solution</h2>
<pre><code>public interface MyInterface {...}
</code></pre>
</td></tr>
<tr><td>513</td><td>Exit methods should not be called</td><td>BLOCKER</td><td>
<p>Calling <code>System.exit(int status)</code> or <code>Rutime.getRuntime().exit(int status)</code> calls the shutdown hooks and shuts downs the
entire Java virtual machine. Calling <code>Runtime.getRuntime().halt(int)</code> does an immediate shutdown, without calling the shutdown hooks, and
skipping finalization.</p>
<p>Each of these methods should be used with extreme care, and only when the intent is to stop the whole Java process. For instance, none of them
should be called from applications running in a J2EE container.</p>
<h2>Noncompliant Code Example</h2>
<pre><code>System.exit(0);
Runtime.getRuntime().exit(0);
Runtime.getRuntime().halt(0);
</code></pre>
</td></tr>
</table>

			Exceptions These methods are ignored inside `main`. **See** - MITRE, CWE-382 - Use of System.exit()

514	Standard outputs should not be used directly to log anything	MAJOR	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

			According to the Java Language Specification:

<table>
<tr>
<td>515</td>
<td>Array designators "[]" should be located after the type in method signatures</td>
<td>MINOR</td>
<td>

> For compatibility with older versions of the Java SE platform,
>
> the declaration of a method that returns an array is allowed to place (some or all of) the empty bracket pairs that form the declaration of the
> array type after the formal parameter list.
>
> This obsolescent syntax should not be used in new code.

Noncompliant Code Example

```
public int getVector()[] { /* ... / }    // Noncompliant

public int[] getMatrix()[] { / ... / }  // Noncompliant
```

Compliant Solution

```
public int[] getVector() { / ... / }

public int[][] getMatrix() { / ... */ }
```

</td>
</tr>
<tr>
<td>516</td>
<td>"super.finalize()" should be called at the end of "Object.finalize()" implementations</td>
<td>CRITICAL</td>
<td>

Overriding the `Object.finalize()` method must be done with caution to dispose some system resources.

Calling the `super.finalize()` at the end of this method implementation is highly recommended in case parent implementations must also
dispose some system resources.

Noncompliant Code Example

```
protected void finalize() {   // Noncompliant; no call to super.finalize();
  releaseSomeResources();
}

protected void finalize() {
  super.finalize();  // Noncompliant; this call should come last
  releaseSomeResources();
}
```

Compliant Solution

</td>
</tr>
</table>

```
protected void finalize() {
  releaseSomeResources();
  super.finalize();
}
```

See

- MITRE, CWE-568 - finalize() Method Without super.finalize()
- CERT, MET12-J. - Do not use finalizers

517

Throwable.printStackTrace(...) should not be called

MINOR

`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 `Throwable`s, 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);
}
```

518	"java.lang.Error" should not be extended	MAJOR	`java.lang.Error` and its subclasses represent abnormal conditions, such as `OutOfMemoryError` ,
			which should only be
			encountered by the Java Virtual Machine.
			Noncompliant Code Example
			`public class MyException extends Error { /* ... / }       // Noncompliant`
			Compliant Solution
			`public class MyException extends Exception { / ... */ }   // Compliant`

519	Source files should not have any duplicated blocks	MAJOR	An issue is created on a file as soon as there is at least one block of duplicated code on this file
			Comparisons of dissimilar types will always return false. The comparison and all its dependent code can simply be removed. This includes: - comparing an object with null - comparing an object with an unrelated primitive (E.G. a string with an int) - comparing unrelated classes - comparing an unrelated `class` and `interface` - comparing unrelated `interface` types - comparing an array to a non-array - comparing two arrays Specifically in the case of arrays, since arrays don't override `Object.equals()` , calling `equals` on two arrays is the same as comparing their addresses. This means that `array1.equals(array2)` is equivalent to `array1==array2` . However, some developers might expect `Array.equals(Object obj)` to do more than a simple memory address com comparing for instance the size and content of the two arrays. Instead, the `==` operator or `Arrays.equals(array1, array2)` should always be used with arrays. **Noncompliant Code Example**

<table>
<tr>
<td>520</td>
<td>Silly equality checks should not be made</td>
<td>MAJOR</td>
<td>

```
interface KitchenTool { ... };
interface Plant {...}

public class Spatula implements KitchenTool { ... }
public class Tree implements Plant { ...}
//...

Spatula spatula = new Spatula();
KitchenTool tool = spatula;
KitchenTool [] tools = {tool};

Tree tree = new Tree();
Plant plant = tree;
Tree [] trees = {tree};


if (spatula.equals(tree)) { // Noncompliant; unrelated classes
  // ...
}
else if (spatula.equals(plant)) { // Noncompliant; unrelated class and interface
  // ...
}
else if (tool.equals(plant)) { // Noncompliant; unrelated interfaces
  // ...
}
else if (tool.equals(tools)) { // Noncompliant; array & non-array
  // ...
}
else if (trees.equals(tools)) {  // Noncompliant; incompatible arrays
  // ...
}
else if (tree.equals(null)) {  // Noncompliant
  // ...
}
```

See

- CERT, EXP02-J. - Do not use the Object.equals() method to compare two arrays

</td>
</tr>
</table>