FindBugs Bug Descriptions

This document lists the standard bug patterns reported by FindBugs version 3.0.1.

Summary

Description	Category
BC: Equals method should not assume anything about the type of its argument	Bad practice
BIT: Check for sign of bitwise operation	Bad practice
CN: Class implements Cloneable but does not define or use clone method	Bad practice
CN: clone method does not call super.clone()	Bad practice
CN: Class defines clone() but doesn't implement Cloneable	Bad practice
CNT: Rough value of known constant found	Bad practice
Co: Abstract class defines covariant compareTo() method	Bad practice
Co: compareTo()/compare() incorrectly handles float or double value	Bad practice
Co: compareTo()/compare() returns Integer.MIN_VALUE	Bad practice
Co: Covariant compareTo() method defined	Bad practice
DE: Method might drop exception	Bad practice
DE: Method might ignore exception	Bad practice
DMI: Adding elements of an entry set may fail due to reuse of Entry objects	Bad practice
DMI: Random object created and used only once	Bad practice
DMI: Don't use removeAll to clear a collection	Bad practice
Dm: Method invokes System.exit(...)	Bad practice
Dm: Method invokes dangerous method runFinalizersOnExit	Bad practice
ES: Comparison of String parameter using == or !=	Bad practice
ES: Comparison of String objects using == or !=	Bad practice
Eq: Abstract class defines covariant equals() method	Bad practice
Eq: Equals checks for incompatible operand	Bad practice

Eq: Class defines compareTo(...) and uses Object.equals()	Bad practice
Eq: equals method fails for subtypes	Bad practice
Eq: Covariant equals() method defined	Bad practice
FI: Empty finalizer should be deleted	Bad practice
FI: Explicit invocation of finalizer	Bad practice
FI: Finalizer nulls fields	Bad practice
FI: Finalizer only nulls fields	Bad practice
FI: Finalizer does not call superclass finalizer	Bad practice
FI: Finalizer nullifies superclass finalizer	Bad practice
FI: Finalizer does nothing but call superclass finalizer	Bad practice
FS: Format string should use %n rather than \n	Bad practice
GC: Unchecked type in generic call	Bad practice
HE: Class defines equals() but not hashCode()	Bad practice
HE: Class defines equals() and uses Object.hashCode()	Bad practice
HE: Class defines hashCode() but not equals()	Bad practice
HE: Class defines hashCode() and uses Object.equals()	Bad practice
HE: Class inherits equals() and uses Object.hashCode()	Bad practice
IC: Superclass uses subclass during initialization	Bad practice
IMSE: Dubious catching of IllegalMonitorStateException	Bad practice
ISC: Needless instantiation of class that only supplies static methods	Bad practice
It: Iterator next() method can't throw NoSuchElementException	Bad practice
J2EE: Store of non serializable object into HttpSession	Bad practice
JCIP: Fields of immutable classes should be final	Bad practice
ME: Public enum method unconditionally sets its field	Bad practice
ME: Enum field is public and mutable	Bad practice
NP: Method with Boolean return type returns explicit null	Bad practice
NP: Clone method may return null	Bad practice
NP: equals() method does not check for null argument	Bad practice
NP: toString method may return null	Bad practice
Nm: Class names should start with an upper case letter	Bad practice
Nm: Class is not derived from an Exception, even though it is named as such	Bad practice

Nm: Confusing method names	Bad practice
Nm: Field names should start with a lower case letter	Bad practice
Nm: Use of identifier that is a keyword in later versions of Java	Bad practice
Nm: Use of identifier that is a keyword in later versions of Java	Bad practice
Nm: Method names should start with a lower case letter	Bad practice
Nm: Class names shouldn't shadow simple name of implemented interface	Bad practice
Nm: Class names shouldn't shadow simple name of superclass	Bad practice
Nm: Very confusing method names (but perhaps intentional)	Bad practice
Nm: Method doesn't override method in superclass due to wrong package for parameter	Bad practice
ODR: Method may fail to close database resource	Bad practice
ODR: Method may fail to close database resource on exception	Bad practice
OS: Method may fail to close stream	Bad practice
OS: Method may fail to close stream on exception	Bad practice
PZ: Don't reuse entry objects in iterators	Bad practice
RC: Suspicious reference comparison to constant	Bad practice
RC: Suspicious reference comparison of Boolean values	Bad practice
RR: Method ignores results of InputStream.read()	Bad practice
RR: Method ignores results of InputStream.skip()	Bad practice
RV: Negating the result of compareTo()/compare()	Bad practice
RV: Method ignores exceptional return value	Bad practice
SI: Static initializer creates instance before all static final fields assigned	Bad practice
SW: Certain swing methods needs to be invoked in Swing thread	Bad practice
Se: Non-transient non-serializable instance field in serializable class	Bad practice
Se: Non-serializable class has a serializable inner class	Bad practice
Se: Non-serializable value stored into instance field of a serializable class	Bad practice

Se: Comparator doesn't implement Serializable	Bad practice
Se: Serializable inner class	Bad practice
Se: serialVersionUID isn't final	Bad practice
Se: serialVersionUID isn't long	Bad practice
Se: serialVersionUID isn't static	Bad practice
Se: Class is Serializable but its superclass doesn't define a void constructor	Bad practice
Se: Class is Externalizable but doesn't define a void constructor	Bad practice
Se: The readResolve method must be declared with a return type of Object.	Bad practice
Se: Transient field that isn't set by deserialization.	Bad practice
SnVI: Class is Serializable, but doesn't define serialVersionUID	Bad practice
UI: Usage of GetResource may be unsafe if class is extended	Bad practice
BC: Impossible cast	Correctness
BC: Impossible downcast	Correctness
BC: Impossible downcast of toArray() result	Correctness
BC: instanceof will always return false	Correctness
BIT: Bitwise add of signed byte value	Correctness
BIT: Incompatible bit masks	Correctness
BIT: Check to see if ((...) & 0) == 0	Correctness
BIT: Incompatible bit masks	Correctness
BIT: Bitwise OR of signed byte value	Correctness
BIT: Check for sign of bitwise operation	Correctness
BOA: Class overrides a method implemented in super class Adapter wrongly	Correctness
BSHIFT: Possible bad parsing of shift operation	Correctness
BSHIFT: 32 bit int shifted by an amount not in the range -31..31	Correctness
DLS: Useless increment in return statement	Correctness
DLS: Dead store of class literal	Correctness
DLS: Overwritten increment	Correctness
DMI: Reversed method arguments	Correctness

DMI: Bad constant value for month	Correctness
DMI: BigDecimal constructed from double that isn't represented precisely	Correctness
DMI: hasNext method invokes next	Correctness
DMI: Collections should not contain themselves	Correctness
DMI: D'oh! A nonsensical method invocation	Correctness
DMI: Invocation of hashCode on an array	Correctness
DMI: Double.longBitsToDouble invoked on an int	Correctness
DMI: Vacuous call to collections	Correctness
Dm: Can't use reflection to check for presence of annotation without runtime retention	Correctness
Dm: Futile attempt to change max pool size of ScheduledThreadPoolExecutor	Correctness
Dm: Creation of ScheduledThreadPoolExecutor with zero core threads	Correctness
Dm: Useless/vacuous call to EasyMock method	Correctness
Dm: Incorrect combination of Math.max and Math.min	Correctness
EC: equals() used to compare array and nonarray	Correctness
EC: Invocation of equals() on an array, which is equivalent to ==	Correctness
EC: equals(...) used to compare incompatible arrays	Correctness
EC: Call to equals(null)	Correctness
EC: Call to equals() comparing unrelated class and interface	Correctness
EC: Call to equals() comparing different interface types	Correctness
EC: Call to equals() comparing different types	Correctness
EC: Using pointer equality to compare different types	Correctness
Eq: equals method always returns false	Correctness
Eq: equals method always returns true	Correctness
Eq: equals method compares class names rather than class objects	Correctness
Eq: Covariant equals() method defined for enum	Correctness
Eq: equals() method defined that doesn't override equals(Object)	Correctness
Eq: equals() method defined that doesn't override Object.equals(Object)	Correctness

Eq: equals method overrides equals in superclass and may not be symmetric	Correctness
Eq: Covariant equals() method defined, Object.equals(Object) inherited	Correctness
FE: Doomed test for equality to NaN	Correctness
FS: Format string placeholder incompatible with passed argument	Correctness
FS: The type of a supplied argument doesn't match format specifier	Correctness
FS: MessageFormat supplied where printf style format expected	Correctness
FS: More arguments are passed than are actually used in the format string	Correctness
FS: Illegal format string	Correctness
FS: Format string references missing argument	Correctness
FS: No previous argument for format string	Correctness
GC: No relationship between generic parameter and method argument	Correctness
HE: Signature declares use of unhashable class in hashed construct	Correctness
HE: Use of class without a hashCode() method in a hashed data structure	Correctness
ICAST: int value converted to long and used as absolute time	Correctness
ICAST: Integral value cast to double and then passed to Math.ceil	Correctness
ICAST: int value cast to float and then passed to Math.round	Correctness
IJU: JUnit assertion in run method will not be noticed by JUnit	Correctness
IJU: TestCase declares a bad suite method	Correctness
IJU: TestCase has no tests	Correctness
IJU: TestCase defines setUp that doesn't call super.setUp()	Correctness
IJU: TestCase implements a non-static suite method	Correctness
IJU: TestCase defines tearDown that doesn't call super.tearDown()	Correctness
IL: A collection is added to itself	Correctness

IL: An apparent infinite loop	Correctness
IL: An apparent infinite recursive loop	Correctness
IM: Integer multiply of result of integer remainder	Correctness
INT: Bad comparison of int value with long constant	Correctness
INT: Bad comparison of nonnegative value with negative constant or zero	Correctness
INT: Bad comparison of signed byte	Correctness
IO: Doomed attempt to append to an object output stream	Correctness
IP: A parameter is dead upon entry to a method but overwritten	Correctness
MF: Class defines field that masks a superclass field	Correctness
MF: Method defines a variable that obscures a field	Correctness
NP: Null pointer dereference	Correctness
NP: Null pointer dereference in method on exception path	Correctness
NP: Method does not check for null argument	Correctness
NP: close() invoked on a value that is always null	Correctness
NP: Null value is guaranteed to be dereferenced	Correctness
NP: Value is null and guaranteed to be dereferenced on exception path	Correctness
NP: Non-null field is not initialized	Correctness
NP: Method call passes null to a non-null parameter	Correctness
NP: Method may return null, but is declared @Nonnull	Correctness
NP: A known null value is checked to see if it is an instance of a type	Correctness
NP: Possible null pointer dereference	Correctness
NP: Possible null pointer dereference in method on exception path	Correctness
NP: Method call passes null for non-null parameter	Correctness
NP: Method call passes null for non-null parameter	Correctness
NP: Non-virtual method call passes null for non-null parameter	Correctness
NP: Method with Optional return type returns explicit null	Correctness
NP: Store of null value into field annotated @Nonnull	Correctness
NP: Read of unwritten field	Correctness
Nm: Class defines equal(Object); should it be	Correctness

equals(Object)?	
Nm: Class defines hashcode(); should it be hashCode()?	Correctness
Nm: Class defines tostring(); should it be toString()?	Correctness
Nm: Apparent method/constructor confusion	Correctness
Nm: Very confusing method names	Correctness
Nm: Method doesn't override method in superclass due to wrong package for parameter	Correctness
QBA: Method assigns boolean literal in boolean expression	Correctness
RANGE: Array index is out of bounds	Correctness
RANGE: Array length is out of bounds	Correctness
RANGE: Array offset is out of bounds	Correctness
RANGE: String index is out of bounds	Correctness
RC: Suspicious reference comparison	Correctness
RCN: Nullcheck of value previously dereferenced	Correctness
RE: Invalid syntax for regular expression	Correctness
RE: File.separator used for regular expression	Correctness
RE: "." or "|" used for regular expression	Correctness
RV: Random value from 0 to 1 is coerced to the integer 0	Correctness
RV: Bad attempt to compute absolute value of signed 32-bit hashcode	Correctness
RV: Bad attempt to compute absolute value of signed random integer	Correctness
RV: Code checks for specific values returned by compareTo	Correctness
RV: Exception created and dropped rather than thrown	Correctness
RV: Method ignores return value	Correctness
RpC: Repeated conditional tests	Correctness
SA: Self assignment of field	Correctness
SA: Self comparison of field with itself	Correctness
SA: Nonsensical self computation involving a field (e.g., x & x)	Correctness
SA: Self assignment of local rather than assignment to field	Correctness
SA: Self comparison of value with itself	Correctness
SA: Nonsensical self computation involving a variable	

(e.g., x & x)	Correctness
SF: Dead store due to switch statement fall through	Correctness
SF: Dead store due to switch statement fall through to throw	Correctness
SIC: Deadly embrace of non-static inner class and thread local	Correctness
SIO: Unnecessary type check done using instanceof operator	Correctness
SQL: Method attempts to access a prepared statement parameter with index 0	Correctness
SQL: Method attempts to access a result set field with index 0	Correctness
STI: Unneeded use of currentThread() call, to call interrupted()	Correctness
STI: Static Thread.interrupted() method invoked on thread instance	Correctness
Se: Method must be private in order for serialization to work	Correctness
Se: The readResolve method must not be declared as a static method.	Correctness
TQ: Value annotated as carrying a type qualifier used where a value that must not carry that qualifier is required	Correctness
TQ: Comparing values with incompatible type qualifiers	Correctness
TQ: Value that might not carry a type qualifier is always used in a way requires that type qualifier	Correctness
TQ: Value that might carry a type qualifier is always used in a way prohibits it from having that type qualifier	Correctness
TQ: Value annotated as never carrying a type qualifier used where value carrying that qualifier is required	Correctness
TQ: Value without a type qualifier used where a value is required to have that qualifier	Correctness
UMAC: Uncallable method defined in anonymous class	Correctness
UR: Uninitialized read of field in constructor	Correctness
UR: Uninitialized read of field method called from constructor of superclass	Correctness
USELESS_STRING: Invocation of toString on an unnamed array	Correctness

USELESS_STRING: Invocation of toString on an array	Correctness
USELESS_STRING: Array formatted in useless way using format string	Correctness
UwF: Field only ever set to null	Correctness
UwF: Unwritten field	Correctness
VA: Primitive array passed to function expecting a variable number of object arguments	Correctness
LG: Potential lost logger changes due to weak reference in OpenJDK	Experimental
OBL: Method may fail to clean up stream or resource	Experimental
OBL: Method may fail to clean up stream or resource on checked exception	Experimental
Dm: Consider using Locale parameterized version of invoked method	Internationalization
Dm: Reliance on default encoding	Internationalization
DP: Classloaders should only be created inside doPrivileged block	Malicious code vulnerability
DP: Method invoked that should be only be invoked inside a doPrivileged block	Malicious code vulnerability
EI: May expose internal representation by returning reference to mutable object	Malicious code vulnerability
EI2: May expose internal representation by incorporating reference to mutable object	Malicious code vulnerability
FI: Finalizer should be protected, not public	Malicious code vulnerability
MS: May expose internal static state by storing a mutable object into a static field	Malicious code vulnerability
MS: Field isn't final and can't be protected from malicious code	Malicious code vulnerability
MS: Public static method may expose internal representation by returning array	Malicious code vulnerability
MS: Field should be both final and package protected	Malicious code vulnerability
MS: Field is a mutable array	Malicious code vulnerability
MS: Field is a mutable collection	Malicious code vulnerability
MS: Field is a mutable collection which should be package	Malicious code

protected	vulnerability
MS: Field is a mutable Hashtable	Malicious code vulnerability
MS: Field should be moved out of an interface and made package protected	Malicious code vulnerability
MS: Field should be package protected	Malicious code vulnerability
MS: Field isn't final but should be	Malicious code vulnerability
MS: Field isn't final but should be refactored to be so	Malicious code vulnerability
AT: Sequence of calls to concurrent abstraction may not be atomic	Multithreaded correctness
DC: Possible double check of field	Multithreaded correctness
DC: Possible exposure of partially initialized object	Multithreaded correctness
DL: Synchronization on Boolean	Multithreaded correctness
DL: Synchronization on boxed primitive	Multithreaded correctness
DL: Synchronization on interned String	Multithreaded correctness
DL: Synchronization on boxed primitive values	Multithreaded correctness
Dm: Monitor wait() called on Condition	Multithreaded correctness
Dm: A thread was created using the default empty run method	Multithreaded correctness
ESync: Empty synchronized block	Multithreaded correctness
IS: Inconsistent synchronization	Multithreaded correctness
IS: Field not guarded against concurrent access	Multithreaded correctness
JLM: Synchronization performed on Lock	Multithreaded correctness
JLM: Synchronization performed on util.concurrent instance	Multithreaded correctness

JLM: Using monitor style wait methods on util.concurrent abstraction	Multithreaded correctness
LI: Incorrect lazy initialization of static field	Multithreaded correctness
LI: Incorrect lazy initialization and update of static field	Multithreaded correctness
ML: Synchronization on field in futile attempt to guard that field	Multithreaded correctness
ML: Method synchronizes on an updated field	Multithreaded correctness
MSF: Mutable servlet field	Multithreaded correctness
MWN: Mismatched notify()	Multithreaded correctness
MWN: Mismatched wait()	Multithreaded correctness
NN: Naked notify	Multithreaded correctness
NP: Synchronize and null check on the same field.	Multithreaded correctness
No: Using notify() rather than notifyAll()	Multithreaded correctness
RS: Class's readObject() method is synchronized	Multithreaded correctness
RV: Return value of putIfAbsent ignored, value passed to putIfAbsent reused	Multithreaded correctness
Ru: Invokes run on a thread (did you mean to start it instead?)	Multithreaded correctness
SC: Constructor invokes Thread.start()	Multithreaded correctness
SP: Method spins on field	Multithreaded correctness
STCAL: Call to static Calendar	Multithreaded correctness
STCAL: Call to static DateFormat	Multithreaded correctness
STCAL: Static Calendar field	Multithreaded correctness
	Multithreaded

STCAL: Static DateFormat	correctness
SWL: Method calls Thread.sleep() with a lock held	Multithreaded correctness
TLW: Wait with two locks held	Multithreaded correctness
UG: Unsynchronized get method, synchronized set method	Multithreaded correctness
UL: Method does not release lock on all paths	Multithreaded correctness
UL: Method does not release lock on all exception paths	Multithreaded correctness
UW: Unconditional wait	Multithreaded correctness
VO: An increment to a volatile field isn't atomic	Multithreaded correctness
VO: A volatile reference to an array doesn't treat the array elements as volatile	Multithreaded correctness
WL: Synchronization on getClass rather than class literal	Multithreaded correctness
WS: Class's writeObject() method is synchronized but nothing else is	Multithreaded correctness
Wa: Condition.await() not in loop	Multithreaded correctness
Wa: Wait not in loop	Multithreaded correctness
Bx: Primitive value is boxed and then immediately unboxed	Performance
Bx: Primitive value is boxed then unboxed to perform primitive coercion	Performance
Bx: Primitive value is unboxed and coerced for ternary operator	Performance
Bx: Boxed value is unboxed and then immediately reboxed	Performance
Bx: Boxing a primitive to compare	Performance
Bx: Boxing/unboxing to parse a primitive	Performance
Bx: Method allocates a boxed primitive just to call toString	Performance
Bx: Method invokes inefficient floating-point Number constructor; use static valueOf instead	Performance

Bx: Method invokes inefficient Number constructor; use static valueOf instead	Performance
Dm: The equals and hashCode methods of URL are blocking	Performance
Dm: Maps and sets of URLs can be performance hogs	Performance
Dm: Method invokes inefficient Boolean constructor; use Boolean.valueOf(...) instead	Performance
Dm: Explicit garbage collection; extremely dubious except in benchmarking code	Performance
Dm: Method allocates an object, only to get the class object	Performance
Dm: Use the nextInt method of Random rather than nextDouble to generate a random integer	Performance
Dm: Method invokes inefficient new String(String) constructor	Performance
Dm: Method invokes toString() method on a String	Performance
Dm: Method invokes inefficient new String() constructor	Performance
HSC: Huge string constants is duplicated across multiple class files	Performance
SBSC: Method concatenates strings using + in a loop	Performance
SIC: Should be a static inner class	Performance
SIC: Could be refactored into a named static inner class	Performance
SIC: Could be refactored into a static inner class	Performance
SS: Unread field: should this field be static?	Performance
UM: Method calls static Math class method on a constant value	Performance
UPM: Private method is never called	Performance
UrF: Unread field	Performance
UuF: Unused field	Performance
WMI: Inefficient use of keySet iterator instead of entrySet iterator	Performance
Dm: Hardcoded constant database password	Security
Dm: Empty database password	Security
HRS: HTTP cookie formed from untrusted input	Security
HRS: HTTP Response splitting vulnerability	Security
PT: Absolute path traversal in servlet	Security

PT: Relative path traversal in servlet	Security
SQL: Nonconstant string passed to execute or addBatch method on an SQL statement	Security
SQL: A prepared statement is generated from a nonconstant String	Security
XSS: JSP reflected cross site scripting vulnerability	Security
XSS: Servlet reflected cross site scripting vulnerability in error page	Security
XSS: Servlet reflected cross site scripting vulnerability	Security
BC: Questionable cast to abstract collection	Dodgy code
BC: Questionable cast to concrete collection	Dodgy code
BC: Unchecked/unconfirmed cast	Dodgy code
BC: Unchecked/unconfirmed cast of return value from method	Dodgy code
BC: instanceof will always return true	Dodgy code
BSHIFT: Unsigned right shift cast to short/byte	Dodgy code
CI: Class is final but declares protected field	Dodgy code
DB: Method uses the same code for two branches	Dodgy code
DB: Method uses the same code for two switch clauses	Dodgy code
DLS: Dead store to local variable	Dodgy code
DLS: Useless assignment in return statement	Dodgy code
DLS: Dead store of null to local variable	Dodgy code
DLS: Dead store to local variable that shadows field	Dodgy code
DMI: Code contains a hard coded reference to an absolute pathname	Dodgy code
DMI: Non serializable object written to ObjectOutput	Dodgy code
DMI: Invocation of substring(0), which returns the original value	Dodgy code
Dm: Thread passed where Runnable expected	Dodgy code
Eq: Class doesn't override equals in superclass	Dodgy code
Eq: Unusual equals method	Dodgy code
FE: Test for floating point equality	Dodgy code
FS: Non-Boolean argument formatted using %b format specifier	Dodgy code
IA: Potentially ambiguous invocation of either an inherited	Dodgy code

or outer method	
IC: Initialization circularity	Dodgy code
ICAST: Integral division result cast to double or float	Dodgy code
ICAST: Result of integer multiplication cast to long	Dodgy code
IM: Computation of average could overflow	Dodgy code
IM: Check for oddness that won't work for negative numbers	Dodgy code
INT: Integer remainder modulo 1	Dodgy code
INT: Vacuous bit mask operation on integer value	Dodgy code
INT: Vacuous comparison of integer value	Dodgy code
MTIA: Class extends Servlet class and uses instance variables	Dodgy code
MTIA: Class extends Struts Action class and uses instance variables	Dodgy code
NP: Dereference of the result of readLine() without nullcheck	Dodgy code
NP: Immediate dereference of the result of readLine()	Dodgy code
NP: Load of known null value	Dodgy code
NP: Method tightens nullness annotation on parameter	Dodgy code
NP: Method relaxes nullness annotation on return value	Dodgy code
NP: Possible null pointer dereference due to return value of called method	Dodgy code
NP: Possible null pointer dereference on branch that might be infeasible	Dodgy code
NP: Parameter must be non-null but is marked as nullable	Dodgy code
NP: Read of unwritten public or protected field	Dodgy code
NS: Potentially dangerous use of non-short-circuit logic	Dodgy code
NS: Questionable use of non-short-circuit logic	Dodgy code
PZLA: Consider returning a zero length array rather than null	Dodgy code
QF: Complicated, subtle or wrong increment in for-loop	Dodgy code
RCN: Redundant comparison of non-null value to null	Dodgy code
RCN: Redundant comparison of two null values	Dodgy code
RCN: Redundant nullcheck of value known to be non-null	Dodgy code
RCN: Redundant nullcheck of value known to be null	Dodgy code

REC: Exception is caught when Exception is not thrown	Dodgy code
RI: Class implements same interface as superclass	Dodgy code
RV: Method checks to see if result of String.indexOf is positive	Dodgy code
RV: Method discards result of readLine after checking if it is non-null	Dodgy code
RV: Remainder of hashCode could be negative	Dodgy code
RV: Remainder of 32-bit signed random integer	Dodgy code
RV: Method ignores return value, is this OK?	Dodgy code
RV: Return value of method without side effect is ignored	Dodgy code
SA: Double assignment of field	Dodgy code
SA: Double assignment of local variable	Dodgy code
SA: Self assignment of local variable	Dodgy code
SF: Switch statement found where one case falls through to the next case	Dodgy code
SF: Switch statement found where default case is missing	Dodgy code
ST: Write to static field from instance method	Dodgy code
Se: Private readResolve method not inherited by subclasses	Dodgy code
Se: Transient field of class that isn't Serializable.	Dodgy code
TQ: Value required to have type qualifier, but marked as unknown	Dodgy code
TQ: Value required to not have type qualifier, but marked as unknown	Dodgy code
UC: Condition has no effect	Dodgy code
UC: Condition has no effect due to the variable type	Dodgy code
UC: Useless object created	Dodgy code
UC: Useless object created on stack	Dodgy code
UC: Useless non-empty void method	Dodgy code
UCF: Useless control flow	Dodgy code
UCF: Useless control flow to next line	Dodgy code
UrF: Unread public/protected field	Dodgy code
UuF: Unused public or protected field	Dodgy code
UwF: Field not initialized in constructor but dereferenced without null check	Dodgy code

UwF: Unwritten public or protected field	Dodgy code
XFB: Method directly allocates a specific implementation of xml interfaces	Dodgy code

Descriptions

BC: Equals method should not assume anything about the type of its argument (BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS)

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`.

BIT: Check for sign of bitwise operation (BIT_SIGNED_CHECK)

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

CN: Class implements Cloneable but does not define or use clone method (CN_IDIOM)

Class implements Cloneable but does not define or use the clone method.

CN: clone method does not call super.clone()

(CN_IDIOM_NO_SUPER_CALL)

This non-final class defines a clone() method that does not call super.clone(). If this class ("*A*") is extended by a subclass ("*B*"), and the subclass *B* calls super.clone(), then it is likely that *B*'s clone() method will return an object of type *A*, which violates the standard contract for clone().

If all clone() methods call super.clone(), then they are guaranteed to use Object.clone(), which always returns an object of the correct type.

CN: Class defines clone() but doesn't implement Cloneable (CN_IMPLEMENTS_CLONE_BUT_NOT_CLONEABLE)

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.

CNT: Rough value of known constant found (CNT_ROUGH_CONSTANT_VALUE)

It's recommended to use the predefined library constant for code clarity and better precision.

Co: Abstract class defines covariant compareTo() method (CO_ABSTRACT_SELF)

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`.

Co: compareTo()/compare() incorrectly handles float or double value (CO_COMPARETO_INCORRECT_FLOATING)

This method compares double or float values using pattern like this: val1 > val2 ? 1 : val1 < val2 ? -1 : 0. This pattern works incorrectly for -0.0 and NaN values which may result in incorrect sorting result or broken collection (if compared values are used as keys). Consider using Double.compare or Float.compare static methods which handle all the special cases correctly.

Co: compareTo()/compare() returns Integer.MIN_VALUE (CO_COMPARETO_RESULTS_MIN_VALUE)

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.

Co: Covariant compareTo() method defined (CO_SELF_NO_OBJECT)

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`.

DE: Method might drop exception (DE_MIGHT_DROP)

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.

DE: Method might ignore exception (DE_MIGHT_IGNORE)

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.

DMI: Adding elements of an entry set may fail due to reuse of Entry objects (DMI_ENTRY_SETS_MAY_REUSE_ENTRY_OBJECTS)

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.

DMI: Random object created and used only once (DMI_RANDOM_USED_ONLY_ONCE)

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).

DMI: Don't use removeAll to clear a collection (DMI_USING_REMOVEALL_TO_CLEAR_COLLECTION)

If you want to remove all elements from a collection `c`, use `c.clear`, not `c.removeAll(c)`. Calling `c.removeAll(c)` to clear a collection is less clear, susceptible to errors from typos, less efficient and for some collections, might throw a `ConcurrentModificationException`.

Dm: Method invokes System.exit(...) (DM_EXIT)

Invoking System.exit shuts down the entire Java virtual machine. This should only been done when it is appropriate. Such calls make it hard or impossible for your code to be invoked by other code. Consider throwing a RuntimeException instead.

Dm: Method invokes dangerous method runFinalizersOnExit (DM_RUN_FINALIZERS_ON_EXIT)

Never call System.runFinalizersOnExit or Runtime.runFinalizersOnExit for any reason: they are among the most dangerous methods in the Java libraries. -- Joshua Bloch

ES: Comparison of String parameter using == or != (ES_COMPARING_PARAMETER_STRING_WITH_EQ)

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.

ES: Comparison of String objects using == or != (ES_COMPARING_STRINGS_WITH_EQ)

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.

Eq: Abstract class defines covariant equals() method (EQ_ABSTRACT_SELF)

This class defines a covariant version of `equals()`. To correctly override the `equals()` method in `java.lang.Object`, the parameter of `equals()` must have type `java.lang.Object`.

Eq: Equals checks for incompatible operand (EQ_CHECK_FOR_OPERAND_NOT_COMPATIBLE_WITH_THIS)

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.

Eq: Class defines compareTo(...) and uses Object.equals() (EQ_COMPARETO_USE_OBJECT_EQUALS)

This class defines a `compareTo(...)` method but inherits its `equals()`

method from `java.lang.Object`. Generally, the value of compareTo should return zero if and only if equals returns true. If this is violated, weird and unpredictable failures will occur in classes such as PriorityQueue. In Java 5 the PriorityQueue.remove method uses the compareTo method, while in Java 6 it uses the equals method.

From the JavaDoc for the compareTo method in the Comparable interface:

> 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."

Eq: equals method fails for subtypes (EQ_GETCLASS_AND_CLASS_CONSTANT)

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()`.

Eq: Covariant equals() method defined (EQ_SELF_NO_OBJECT)

This class defines a covariant version of `equals()`. To correctly override the `equals()` method in `java.lang.Object`, the parameter of `equals()` must have type `java.lang.Object`.

FI: Empty finalizer should be deleted (FI_EMPTY)

Empty `finalize()` methods are useless, so they should be deleted.

FI: Explicit invocation of finalizer (FI_EXPLICIT_INVOCATION)

This method contains an explicit invocation of the `finalize()` method on an object. Because finalizer methods are supposed to be executed once, and only by the VM, this is a bad idea.

If a connected set of objects beings finalizable, then the VM will invoke the finalize method on all the finalizable object, possibly at the same time in different threads. Thus, it is a particularly bad idea, in the finalize method for a class X, invoke finalize on objects referenced by X, because they may already be getting finalized in a separate thread.

FI: Finalizer nulls fields (FI_FINALIZER_NULLS_FIELDS)

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.

FI: Finalizer only nulls fields (FI_FINALIZER_ONLY_NULLS_FIELDS)

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.

FI: Finalizer does not call superclass finalizer (FI_MISSING_SUPER_CALL)

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()`.

FI: Finalizer nullifies superclass finalizer (FI_NULLIFY_SUPER)

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.

FI: Finalizer does nothing but call superclass finalizer (FI_USELESS)

The only thing this `finalize()` method does is call the superclass's `finalize()` method, making it redundant. Delete it.

FS: Format string should use %n rather than \n (VA_FORMAT_STRING_USES_NEWLINE)

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.

GC: Unchecked type in generic call (GC_UNCHECKED_TYPE_IN_GENERIC_CALL)

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.

HE: Class defines equals() but not hashCode() (HE_EQUALS_NO_HASHCODE)

This class overrides `equals(Object)`, but does not override `hashCode()`. Therefore, the class may violate the invariant that equal objects must have equal hashcodes.

HE: Class defines equals() and uses Object.hashCode() (HE_EQUALS_USE_HASHCODE)

This class overrides `equals(Object)`, but does not override `hashCode()`, and inherits the implementation of `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 think instances of this class will ever be inserted into a HashMap/HashTable, the recommended `hashCode` implementation to use is:

```
public int hashCode() {
  assert false : "hashCode not designed";
  return 42; // any arbitrary constant will do
  }
```

HE: Class defines hashCode() but not equals() (HE_HASHCODE_NO_EQUALS)

This class defines a `hashCode()` method but not an `equals()` method. Therefore, the class may violate the invariant that equal objects must have equal hashcodes.

HE: Class defines hashCode() and uses Object.equals() (HE_HASHCODE_USE_OBJECT_EQUALS)

This class defines a `hashCode()` method but inherits its `equals()` method from `java.lang.Object` (which defines equality by comparing object references). Although this will probably satisfy the contract that equal objects must have equal hashcodes, it is probably not what was intended by overriding the `hashCode()` method. (Overriding `hashCode()` implies that the object's identity is based on criteria more complicated than simple reference equality.)

If you don't think instances of this class will ever be inserted into a HashMap/HashTable, the recommended `hashCode` implementation to use is:

```
public int hashCode() {
  assert false : "hashCode not designed";
  return 42; // any arbitrary constant will do
  }
```

HE: Class inherits equals() and uses Object.hashCode() (HE_INHERITS_EQUALS_USE_HASHCODE)

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`.

IC: Superclass uses subclass during initialization (IC_SUPERCLASS_USES_SUBCLASS_DURING_INITIALIZATION)

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. 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;
}
```

IMSE: Dubious catching of IllegalMonitorStateException (IMSE_DONT_CATCH_IMSE)

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).

ISC: Needless instantiation of class that only supplies static methods (ISC_INSTANTIATE_STATIC_CLASS)

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.

It: Iterator next() method can't throw NoSuchElementException (IT_NO_SUCH_ELEMENT)

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.

J2EE: Store of non serializable object into HttpSession (J2EE_STORE_OF_NON_SERIALIZABLE_OBJECT_INTO_SESSION)

This code seems to be storing a non-serializable object into an HttpSession. If this session is passivated or migrated, an error will result.

JCIP: Fields of immutable classes should be final (JCIP_FIELD_ISNT_FINAL_IN_IMMUTABLE_CLASS)

The class is annotated with net.jcip.annotations.Immutable or javax.annotation.concurrent.Immutable, and the rules for those annotations require that all fields are final. .

ME: Public enum method unconditionally sets its field (ME_ENUM_FIELD_SETTER)

This public method declared in public enum unconditionally sets enum field, thus this field can be changed by malicious code or by accident from another package. Though mutable enum fields may be used for lazy initialization, it's a bad practice to expose them to the outer world. Consider removing this method or declaring it package-private.

ME: Enum field is public and mutable (ME_MUTABLE_ENUM_FIELD)

A mutable public field is defined inside a public enum, thus can be changed by malicious code or by accident from another package. Though mutable enum fields may be used for lazy initialization, it's a bad practice to expose them to the outer world. Consider declaring this field final and/or package-private.

NP: Method with Boolean return type returns explicit null (NP_BOOLEAN_RETURN_NULL)

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.

NP: Clone method may return null (NP_CLONE_COULD_RETURN_NULL)

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.

NP: equals() method does not check for null argument (NP_EQUALS_SHOULD_HANDLE_NULL_ARGUMENT)

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.

NP: toString method may return null (NP_TOSTRING_COULD_RETURN_NULL)

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.

Nm: Class names should start with an upper case letter (NM_CLASS_NAMING_CONVENTION)

Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

Nm: Class is not derived from an Exception, even though it is named as such (NM_CLASS_NOT_EXCEPTION)

This class is not derived from another exception, but ends with 'Exception'. This will be confusing to users of this class.

Nm: Confusing method names (NM_CONFUSING)

The referenced methods have names that differ only by capitalization.

Nm: Field names should start with a lower case letter (NM_FIELD_NAMING_CONVENTION)

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.

Nm: Use of identifier that is a keyword in later versions of Java (NM_FUTURE_KEYWORD_USED_AS_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.

Nm: Use of identifier that is a keyword in later versions of Java (NM_FUTURE_KEYWORD_USED_AS_MEMBER_IDENTIFIER)

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.

Nm: Method names should start with a lower case letter (NM_METHOD_NAMING_CONVENTION)

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

Nm: Class names shouldn't shadow simple name of implemented interface (NM_SAME_SIMPLE_NAME_AS_INTERFACE)

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 accidentally define methods that do not override methods in their superclasses.

Nm: Class names shouldn't shadow simple name of superclass (NM_SAME_SIMPLE_NAME_AS_SUPERCLASS)

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 accidentally define methods that do not override methods in their superclasses.

Nm: Very confusing method names (but perhaps intentional) (NM_VERY_CONFUSING_INTENTIONAL)

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.

Nm: Method doesn't override method in superclass due to wrong package for parameter (NM_WRONG_PACKAGE_INTENTIONAL)

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.

ODR: Method may fail to close database resource (ODR_OPEN_DATABASE_RESOURCE)

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.

ODR: Method may fail to close database resource on exception (ODR_OPEN_DATABASE_RESOURCE_EXCEPTI

ON_PATH)

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.

OS: Method may fail to close stream (OS_OPEN_STREAM)

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.

OS: Method may fail to close stream on exception (OS_OPEN_STREAM_EXCEPTION_PATH)

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.

PZ: Don't reuse entry objects in iterators (PZ_DONT_REUSE_ENTRY_OBJECTS_IN_ITERATORS)

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.

RC: Suspicious reference comparison to constant (RC_REF_COMPARISON_BAD_PRACTICE)

This method compares a reference value to a constant using the == or != operator, where the correct way to compare instances of this type is generally with the equals() method. It is possible to create distinct instances that are equal but do not compare as == since they are different objects. Examples of classes which should generally not be compared by reference are java.lang.Integer, java.lang.Float, etc.

RC: Suspicious reference comparison of Boolean values (RC_REF_COMPARISON_BAD_PRACTICE_BOOLEAN)

This method compares two Boolean values using the == or != operator. Normally, there are only two Boolean values (Boolean.TRUE and Boolean.FALSE), but it is possible to create other Boolean objects using the `new Boolean(b)` constructor. It is best to avoid such objects, but if they do exist, then checking Boolean objects for equality using == or != will give results than are different than you would get using `.equals(...)`

RR: Method ignores results of InputStream.read() (RR_NOT_CHECKED)

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.

RR: Method ignores results of InputStream.skip() (SR_NOT_CHECKED)

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, skip() will only skip data in the buffer, and will routinely fail to skip the requested number of bytes.

RV: Negating the result of compareTo()/compare() (RV_NEGATING_RESULT_OF_COMPARETO)

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.

RV: Method ignores exceptional return value (RV_RETURN_VALUE_IGNORED_BAD_PRACTICE)

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.

SI: Static initializer creates instance before all

static final fields assigned (SI_INSTANCE_BEFORE_FINALS_ASSIGNED)

The class's static initializer creates an instance of the class before all of the static final fields are assigned.

SW: Certain swing methods needs to be invoked in Swing thread (SW_SWING_METHODS_INVOKED_IN_SWING_THREAD)

(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.

Se: Non-transient non-serializable instance field in serializable class (SE_BAD_FIELD)

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.

Se: Non-serializable class has a serializable inner class (SE_BAD_FIELD_INNER_CLASS)

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.

Se: Non-serializable value stored into instance field of a serializable class (SE_BAD_FIELD_STORE)

A non-serializable value is stored into a non-transient field of a serializable class.

Se: Comparator doesn't implement Serializable (SE_COMPARATOR_SHOULD_BE_SERIALIZABLE)

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.

Se: Serializable inner class (SE_INNER_CLASS)

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.

Se: serialVersionUID isn't final

(SE_NONFINAL_SERIALVERSIONID)

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.

Se: serialVersionUID isn't long (SE_NONLONG_SERIALVERSIONID)

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.

Se: serialVersionUID isn't static (SE_NONSTATIC_SERIALVERSIONID)

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.

Se: Class is Serializable but its superclass doesn't define a void constructor (SE_NO_SUITABLE_CONSTRUCTOR)

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.

Se: Class is Externalizable but doesn't define a void constructor (SE_NO_SUITABLE_CONSTRUCTOR_FOR_EXTERNALIZATION)

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.

Se: The readResolve method must be declared with a return type of Object. (SE_READ_RESOLVE_MUST_RETURN_OBJECT)

In order for the readResolve method to be recognized by the serialization mechanism, it must be declared to have a return type of Object.

Se: Transient field that isn't set by deserialization. (SE_TRANSIENT_FIELD_NOT_RESTORED)

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 value in any deserialized instance of the class.

SnVI: Class is Serializable, but doesn't define serialVersionUID (SE_NO_SERIALVERSIONID)

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.

UI: Usage of GetResource may be unsafe if class is extended (UI_INHERITANCE_UNSAFE_GETRESOURCE)

Calling `this.getClass().getResource(...)` could give results other than expected if this class is extended by a class in another package.

BC: Impossible cast (BC_IMPOSSIBLE_CAST)

This cast will always throw a ClassCastException. FindBugs tracks type information from instanceof checks, and also uses more precise information about the types of values returned from methods and loaded from fields. Thus, it may have more precise information that just the declared type of a variable, and can use this to determine that a cast will always throw an exception at runtime.

BC: Impossible downcast (BC_IMPOSSIBLE_DOWNCAST)

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.

BC: Impossible downcast of toArray() result (BC_IMPOSSIBLE_DOWNCAST_OF_TOARRAY)

This code is casting the result of calling `toArray()` on a collection to a type more specific than `Object[]`, as in:

```
String[] getAsArray(Collection<String> 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 collection.

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 `String []`. FindBugs attempts to detect and suppress such cases, but may miss some.

BC: instanceof will always return false (BC_IMPOSSIBLE_INSTANCEOF)

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.

BIT: Bitwise add of signed byte value (BIT_ADD_OF_SIGNED_BYTE)

Adds a byte value and a value which is known to have 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));
```

BIT: Incompatible bit masks (BIT_AND)

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.

BIT: Check to see if ((...) & 0) == 0 (BIT_AND_ZZ)

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.

BIT: Incompatible bit masks (BIT_IOR)

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.

Typically, this bug occurs because the code wants to perform a membership test in a bit set, but uses the bitwise OR operator ("|") instead of bitwise AND ("&").

BIT: Bitwise OR of signed byte value (BIT_IOR_OF_SIGNED_BYTE)

Loads a byte value (e.g., a value loaded from a byte array or returned by a method with return type byte) and performs a bitwise OR with that value. Byte values 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));
```

BIT: Check for sign of bitwise operation (BIT_SIGNED_CHECK_HIGH_BIT)

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

BOA: Class overrides a method implemented in super class Adapter wrongly (BOA_BADLY_OVERRIDDEN_ADAPTER)

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.

BSHIFT: Possible bad parsing of shift operation

(BSHIFT_WRONG_ADD_PRIORITY)

The code performs an operation like (x << 8 + y). Although this might be correct, probably it was meant to perform (x << 8) + y, but shift operation has a lower precedence, so it's actually parsed as x << (8 + y).

BSHIFT: 32 bit int shifted by an amount not in the range -31..31 (ICAST_BAD_SHIFT_AMOUNT)

The code performs shift of a 32 bit int by a constant amount outside the range -31..31. The effect of this is to use the lower 5 bits of the integer value to decide how much to shift by (e.g., shifting by 40 bits is the same as shifting by 8 bits, and shifting by 32 bits is the same as shifting by zero bits). This probably isn't what was expected, and it is at least confusing.

DLS: Useless increment in return statement (DLS_DEAD_LOCAL_INCREMENT_IN_RETURN)

This statement has a return such as `return x++;`. A postfix increment/decrement does not impact the value of the expression, so this increment/decrement has no effect. Please verify that this statement does the right thing.

DLS: Dead store of class literal (DLS_DEAD_STORE_OF_CLASS_LITERAL)

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. 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.

DLS: Overwritten increment

(DLS_OVERWRITTEN_INCREMENT)

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.

DMI: Reversed method arguments (DMI_ARGUMENTS_WRONG_ORDER)

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.

DMI: Bad constant value for month (DMI_BAD_MONTH)

This code passes a constant month value outside the expected range of 0..11 to a method.

DMI: BigDecimal constructed from double that isn't represented precisely (DMI_BIGDECIMAL_CONSTRUCTED_FROM_DOUBLE)

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 0.1000000000000000055511151231257827021181583404541015625. 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).

DMI: hasNext method invokes next (DMI_CALLING_NEXT_FROM_HASNEXT)

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.

DMI: Collections should not contain themselves (DMI_COLLECTIONS_SHOULD_NOT_CONTAIN_THEMSELVES)

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.

DMI: D'oh! A nonsensical method invocation (DMI_DOH)

This partical method invocation doesn't make sense, for reasons that should be apparent from inspection.

DMI: Invocation of hashCode on an array (DMI_INVOKING_HASHCODE_ON_ARRAY)

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)`.

DMI: Double.longBitsToDouble invoked on an int (DMI_LONG_BITS_TO_DOUBLE_INVOKED_ON_INT)

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.

DMI: Vacuous call to collections (DMI_VACUOUS_SELF_COLLECTION_CALL)

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.

Dm: Can't use reflection to check for presence of annotation without runtime retention (DMI_ANNOTATION_IS_NOT_VISIBLE_TO_REFLECTION)

Unless an annotation has itself been annotated with @Retention(RetentionPolicy.RUNTIME), the annotation can't be observed using reflection (e.g., by using the isAnnotationPresent method). .

Dm: Futile attempt to change max pool size of ScheduledThreadPoolExecutor (DMI_FUTILE_ATTEMPT_TO_CHANGE_MAXPOOL_SIZE_OF_SCHEDULED_THREAD_POOL_EXECUTOR)

(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 to maximumPoolSize have no useful effect.

Dm: Creation of ScheduledThreadPoolExecutor with zero core threads (DMI_SCHEDULED_THREAD_POOL_EXECUTOR_WITH_ZERO_CORE_THREADS)

(Javadoc) A ScheduledThreadPoolExecutor with zero core threads will never execute anything; changes to the max pool size are ignored.

Dm: Useless/vacuous call to EasyMock method (DMI_VACUOUS_CALL_TO_EASYMOCK_METHOD)

This call doesn't pass any objects to the EasyMock method, so the call doesn't do anything.

Dm: Incorrect combination of Math.max and Math.min (DM_INVALID_MIN_MAX)

This code tries to limit the value bounds using the construct like Math.min(0, Math.max(100, value)). However the order of the constants is incorrect: it should be Math.min(100, Math.max(0, value)). As the result this code always produces the same result (or NaN if the value is NaN).

EC: equals() used to compare array and nonarray (EC_ARRAY_AND_NONARRAY)

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[]).

EC: Invocation of equals() on an array, which is equivalent to == (EC_BAD_ARRAY_COMPARE)

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[])`. To compare the addresses of the arrays, it would be less confusing to explicitly check pointer equality using ==.

EC: equals(...) used to compare incompatible arrays (EC_INCOMPATIBLE_ARRAY_COMPARE)

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 only checks to see if they are the same array, and ignores the contents of the arrays.

EC: Call to equals(null) (EC_NULL_ARG)

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`.

EC: Call to equals() comparing unrelated class and interface (EC_UNRELATED_CLASS_AND_INTERFACE)

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.

EC: Call to equals() comparing different interface types (EC_UNRELATED_INTERFACES)

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.

EC: Call to equals() comparing different types (EC_UNRELATED_TYPES)

This method calls equals(Object) on two references of different class types and analysis suggests they will be to objects of different classes at runtime. Further, examination of the equals methods that would be invoked suggest that either this call will always return false, or else the equals method is not be symmetric (which is a property required by the contract for equals in class Object).

EC: Using pointer equality to compare different types (EC_UNRELATED_TYPES_USING_POINTER_EQUALITY)

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.

Eq: equals method always returns false (EQ_ALWAYS_FALSE)

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; }
```

Eq: equals method always returns true (EQ_ALWAYS_TRUE)

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.

Eq: equals method compares class names rather than class objects (EQ_COMPARING_CLASS_NAMES)

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.

Eq: Covariant equals() method defined for enum (EQ_DONT_DEFINE_EQUALS_FOR_ENUM)

This class defines an enumeration, and equality on enumerations are defined using object identity. Defining a covariant equals method for an enumeration value is exceptionally bad practice, since it would likely result in having two different enumeration values that compare as equals using the covariant enum method, and as not equal when compared normally. Don't do it.

Eq: equals() method defined that doesn't override

equals(Object) (EQ_OTHER_NO_OBJECT)

This class defines an `equals()` method, that doesn't override the normal `equals(Object)` method defined in the base `java.lang.Object` class. Instead, it inherits an `equals(Object)` method from a superclass. The class should probably define a `boolean equals(Object)` method.

Eq: equals() method defined that doesn't override Object.equals(Object) (EQ_OTHER_USE_OBJECT)

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.

Eq: equals method overrides equals in superclass and may not be symmetric (EQ_OVERRIDING_EQUALS_NOT_SYMMETRIC)

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.

Eq: Covariant equals() method defined, Object.equals(Object) inherited (EQ_SELF_USE_OBJECT)

This class defines a covariant version of the `equals()` method, but inherits the normal `equals(Object)` method defined in the base `java.lang.Object` class. The class should probably define a `boolean equals(Object)` method.

FE: Doomed test for equality to NaN (FE_TEST_IF_EQUAL_TO_NOT_A_NUMBER)

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).

FS: Format string placeholder incompatible with passed argument (VA_FORMAT_STRING_BAD_ARGUMENT)

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.

FS: The type of a supplied argument doesn't match format specifier (VA_FORMAT_STRING_BAD_CONVERSION)

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.

FS: MessageFormat supplied where printf style format expected (VA_FORMAT_STRING_EXPECTED_MESSAGE_

FORMAT_SUPPLIED)

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 MessageFormat string when a printf-style format string is required. At runtime, all of the arguments will be ignored and the format string will be returned exactly as provided without any formatting.

FS: More arguments are passed than are actually used in the format string (VA_FORMAT_STRING_EXTRA_ARGUMENTS_PASSED)

A format-string method with a variable number of arguments is called, 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.

FS: Illegal format string (VA_FORMAT_STRING_ILLEGAL)

The format string is syntactically invalid, and a runtime exception will occur when this statement is executed.

FS: Format string references missing argument (VA_FORMAT_STRING_MISSING_ARGUMENT)

Not enough arguments are passed to satisfy a placeholder in the format string. A runtime exception will occur when this statement is executed.

FS: No previous argument for format string (VA_FORMAT_STRING_NO_PREVIOUS_ARGUMENT)

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.

GC: No relationship between generic parameter and method argument (GC_UNRELATED_TYPES)

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.

HE: Signature declares use of unhashable class in hashed construct (HE_SIGNATURE_DECLARES_HASHING_OF_UNHASHABLE_CLASS)

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.

HE: Use of class without a hashCode() method in a hashed data structure (HE_USE_OF_UNHASHABLE_CLASS)

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.

ICAST: int value converted to long and used as absolute time (ICAST_INT_2_LONG_AS_INSTANT)

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 epoch 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); }
```

ICAST: Integral value cast to double and then passed to Math.ceil (ICAST_INT_CAST_TO_DOUBLE_PASSED_TO_CEIL)

This code converts an integral value (e.g., int or long) 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. 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.

ICAST: int value cast to float and then passed to Math.round (ICAST_INT_CAST_TO_FLOAT_PASSED_TO_ROUND)

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.

IJU: JUnit assertion in run method will not be noticed by JUnit (IJU_ASSERT_METHOD_INVOKED_FROM_RUN_METHOD)

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.

IJU: TestCase declares a bad suite method (IJU_BAD_SUITE_METHOD)

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()
```

IJU: TestCase has no tests (IJU_NO_TESTS)

Class is a JUnit TestCase but has not implemented any test methods

IJU: TestCase defines setUp that doesn't call super.setUp() (IJU_SETUP_NO_SUPER)

Class is a JUnit TestCase and implements the setUp method. The setUp method should call super.setUp(), but doesn't.

IJU: TestCase implements a non-static suite method (IJU_SUITE_NOT_STATIC)

Class is a JUnit TestCase and implements the suite() method. The suite method should be declared as being static, but isn't.

IJU: TestCase defines tearDown that doesn't call super.tearDown() (IJU_TEARDOWN_NO_SUPER)

Class is a JUnit TestCase and implements the tearDown method. The tearDown method should call super.tearDown(), but doesn't.

IL: A collection is added to itself (IL_CONTAINER_ADDED_TO_ITSELF)

A collection is added to itself. As a result, computing the hashCode of this set will throw a StackOverflowException.

IL: An apparent infinite loop (IL_INFINITE_LOOP)

This loop doesn't seem to have a way to terminate (other than by perhaps throwing an exception).

IL: An apparent infinite recursive loop (IL_INFINITE_RECURSIVE_LOOP)

This method unconditionally invokes itself. This would seem to indicate an infinite recursive loop that will result in a stack overflow.

IM: Integer multiply of result of integer remainder (IM_MULTIPLYING_RESULT_OF_IREM)

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).

INT: Bad comparison of int value with long constant (INT_BAD_COMPARISON_WITH_INT_VALUE)

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.

INT: Bad comparison of nonnegative value with negative constant or zero (INT_BAD_COMPARISON_WITH_NONNEGATIVE_VALUE)

This code compares a value that is guaranteed to be non-negative with a negative constant or zero.

INT: Bad comparison of signed byte (INT_BAD_COMPARISON_WITH_SIGNED_BYTE)

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`

IO: Doomed attempt to append to an object output stream (IO_APPENDING_TO_OBJECT_OUTPUT_STREAM)

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.

IP: A parameter is dead upon entry to a method but overwritten

(IP_PARAMETER_IS_DEAD_BUT_OVERWRITTEN)

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.

MF: Class defines field that masks a superclass field (MF_CLASS_MASKS_FIELD)

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.

MF: Method defines a variable that obscures a field (MF_METHOD_MASKS_FIELD)

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.

NP: Null pointer dereference (NP_ALWAYS_NULL)

A null pointer is dereferenced here. This will lead to a `NullPointerException` when the code is executed.

NP: Null pointer dereference in method on exception path (NP_ALWAYS_NULL_EXCEPTION)

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.

NP: Method does not check for null argument (NP_ARGUMENT_MIGHT_BE_NULL)

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.

NP: close() invoked on a value that is always null (NP_CLOSING_NULL)

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.

NP: Null value is guaranteed to be dereferenced (NP_GUARANTEED_DEREF)

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).

Note that a check such as `if (x == null) throw new NullPointerException();` is treated as a dereference of `x`.

NP: Value is null and guaranteed to be dereferenced on exception path (NP_GUARANTEED_DEREF_ON_EXCEPTION_PATH)

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).

NP: Non-null field is not initialized (NP_NONNULL_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR)

The field is marked as non-null, but isn't written to by the constructor. The field might be initialized elsewhere during constructor, or might always be initialized before use.

NP: Method call passes null to a non-null parameter (NP_NONNULL_PARAM_VIOLATION)

This method passes a null value as the parameter of a method which must be non-null. Either this parameter has been explicitly marked as @Nonnull, or analysis has determined that this parameter is always dereferenced.

NP: Method may return null, but is declared @Nonnull (NP_NONNULL_RETURN_VIOLATION)

This method may return a null value, but the method (or a superclass method which it overrides) is declared to return @Nonnull.

NP: A known null value is checked to see if it is an instance of a type (NP_NULL_INSTANCEOF)

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.

NP: Possible null pointer dereference (NP_NULL_ON_SOME_PATH)

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.

NP: Possible null pointer dereference in method on exception path (NP_NULL_ON_SOME_PATH_EXCEPTION)

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.

NP: Method call passes null for non-null parameter (NP_NULL_PARAM_DEREF)

This method call passes a null value for a non-null method parameter. Either the parameter is annotated as a parameter that should always be non-null, or analysis has shown that it will always be dereferenced.

NP: Method call passes null for non-null parameter (NP_NULL_PARAM_DEREF_ALL_TARGETS_DANGEROUS)

A possibly-null value is passed at a call site where all known target methods require the parameter to be non-null. Either the parameter is annotated as a parameter that should always be non-null, or analysis has shown that it will always be dereferenced.

NP: Non-virtual method call passes null for non-null parameter (NP_NULL_PARAM_DEREF_NONVIRTUAL)

A possibly-null value is passed to a non-null method parameter. Either the parameter is annotated as a parameter that should always be non-null, or analysis has shown that it will always be dereferenced.

NP: Method with Optional return type returns explicit null (NP_OPTIONAL_RETURN_NULL)

The usage of Optional return type (java.util.Optional or com.google.common.base.Optiona) always mean that explicit null returns were not desired by design. Returning a null value in such case is a contract violation and will most likely break clients code.

NP: Store of null value into field annotated @Nonnull (NP_STORE_INTO_NONNULL_FIELD)

A value that could be null is stored into a field that has been annotated as @Nonnull.

NP: Read of unwritten field (NP_UNWRITTEN_FIELD)

The program is dereferencing a 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.

Nm: Class defines equal(Object); should it be equals(Object)? (NM_BAD_EQUAL)

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.

Nm: Class defines hashcode(); should it be hashCode()? (NM_LCASE_HASHCODE)

This class defines a method called `hashcode()`. This method does not override the `hashCode()` method in `java.lang.Object`, which is probably what was intended.

Nm: Class defines tostring(); should it be toString()? (NM_LCASE_TOSTRING)

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.

Nm: Apparent method/constructor confusion (NM_METHOD_CONSTRUCTOR_CONFUSION)

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 accidentally 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.

Nm: Very confusing method names (NM_VERY_CONFUSING)

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.

Nm: Method doesn't override method in superclass due to wrong package for parameter (NM_WRONG_PACKAGE)

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; }
}
```

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.

QBA: Method assigns boolean literal in boolean expression (QBA_QUESTIONABLE_BOOLEAN_ASSIGNMENT)

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 =.

RANGE: Array index is out of bounds (RANGE_ARRAY_INDEX)

Array operation is performed, but array index is out of bounds, which will result in ArrayIndexOutOfBoundsException at runtime.

RANGE: Array length is out of bounds (RANGE_ARRAY_LENGTH)

Method is called with array parameter and length parameter, but the length is out of bounds. This will result in IndexOutOfBoundsException at runtime.

RANGE: Array offset is out of bounds (RANGE_ARRAY_OFFSET)

Method is called with array parameter and offset parameter, but the offset is out of bounds. This will result in IndexOutOfBoundsException at runtime.

RANGE: String index is out of bounds (RANGE_STRING_INDEX)

String method is called and specified string index is out of bounds. This will result in StringIndexOutOfBoundsException at runtime.

RC: Suspicious reference comparison (RC_REF_COMPARISON)

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. It is possible to create distinct instances that are equal but do not compare as == since they are different objects. Examples of classes which should generally not be compared by reference are java.lang.Integer, java.lang.Float, etc.

RCN: Nullcheck of value previously dereferenced (RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE)

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.

RE: Invalid syntax for regular expression (RE_BAD_SYNTAX_FOR_REGULAR_EXPRESSI

ON)

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.

RE: File.separator used for regular expression (RE_CANT_USE_FILE_SEPARATOR_AS_REGULAR_EXPRESSION)

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`

RE: "." or "|" used for regular expression (RE_POSSIBLE_UNINTENDED_PATTERN)

A String function is being invoked and "." or "|" 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
- s.split(".") *always* returns a zero length array of String
- "ab|cd".replaceAll("|", "/") will return "/a/b/|/c/d/"
- "ab|cd".split("|") will return array with six (!) elements: [, a, b, |, c, d]

RV: Random value from 0 to 1 is coerced to the integer 0 (RV_01_TO_INT)

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.

RV: Bad attempt to compute absolute value of signed 32-bit hashcode (RV_ABSOLUTE_VALUE_OF_HASHCODE)

This code generates a hashcode and then computes the absolute value of that hashcode. If the hashcode is `Integer.MIN_VALUE`, then the result will be negative as well (since `Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE`).

One out of 2^32 strings have a hashCode of Integer.MIN_VALUE, including "polygenelubricants" "GydZG_" and ""DESIGNING WORKHOUSES".

RV: Bad attempt to compute absolute value of signed random integer (RV_ABSOLUTE_VALUE_OF_RANDOM_INT)

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.MIN_VALUE`, then the result will be negative as well (since `Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE`). (Same problem arised for long values as well).

RV: Code checks for specific values returned by compareTo (RV_CHECK_COMPARETO_FOR_SPECIFIC_RETURN_VALUE)

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.

RV: Exception created and dropped rather than

thrown (RV_EXCEPTION_NOT_THROWN)

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

```
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");
```

RV: Method ignores return value (RV_RETURN_VALUE_IGNORED)

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();
```

RpC: Repeated conditional tests (RpC_REPEATED_CONDITIONAL_TEST)

The code contains a conditional test is performed twice, one right after the other (e.g., `x == 0 || x == 0`). Perhaps the second occurrence is intended to be something else (e.g., `x == 0 || y == 0`).

SA: Self assignment of field (SA_FIELD_SELF_ASSIGNMENT)

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.

SA: Self comparison of field with itself (SA_FIELD_SELF_COMPARISON)

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.

SA: Nonsensical self computation involving a field (e.g., x & x) (SA_FIELD_SELF_COMPUTATION)

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.

SA: Self assignment of local rather than assignment to field (SA_LOCAL_SELF_ASSIGNMENT_INSTEAD_OF_FIELD)

This method contains a self assignment of a local variable, and there is a field with an identical name. assignment appears to have been ; e.g.

```
int foo;
public void setFoo(int foo) {
  foo = foo;
}
```

The assignment is useless. Did you mean to assign to the field instead?

SA: Self comparison of value with itself (SA_LOCAL_SELF_COMPARISON)

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.

SA: Nonsensical self computation involving a variable (e.g., x & x) (SA_LOCAL_SELF_COMPUTATION)

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.

SF: Dead store due to switch statement fall through (SF_DEAD_STORE_DUE_TO_SWITCH_FALLTHROUGH)

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.

SF: Dead store due to switch statement fall through to throw (SF_DEAD_STORE_DUE_TO_SWITCH_FALLTH

ROUGH_TO_THROW)

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.

SIC: Deadly embrace of non-static inner class and thread local (SIC_THREADLOCAL_DEADLY_EMBRACE)

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 reference to the outer class. If the thread local contains a reference to an instance of the inner class, the inner and outer instance will both be reachable and not eligible for garbage collection.

SIO: Unnecessary type check done using instanceof operator (SIO_SUPERFLUOUS_INSTANCEOF)

Type check performed using the instanceof operator where it can be statically determined whether the object is of the type requested.

SQL: Method attempts to access a prepared statement parameter with index 0 (SQL_BAD_PREPARED_STATEMENT_ACCESS)

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.

SQL: Method attempts to access a result set field with index 0 (SQL_BAD_RESULTSET_ACCESS)

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.

STI: Unneeded use of currentThread() call, to call interrupted() (STI_INTERRUPTED_ON_CURRENTTHREAD)

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().

STI: Static Thread.interrupted() method invoked on thread instance (STI_INTERRUPTED_ON_UNKNOWNTHREAD)

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.

Se: Method must be private in order for serialization to work (SE_METHOD_MUST_BE_PRIVATE)

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.

Se: The readResolve method must not be declared as a static method. (SE_READ_RESOLVE_IS_STATIC)

In order for the readResolve method to be recognized by the serialization mechanism, it must not be declared as a static method.

TQ: Value annotated as carrying a type qualifier used where a value that must not carry that qualifier is required (TQ_ALWAYS_VALUE_USED_WHERE_NEVER_REQUIRED)

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;
}
```

TQ: Comparing values with incompatible type qualifiers (TQ_COMPARING_VALUES_WITH_INCOMPATIBLE_TYPE_QUALIFIERS)

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 va
    return value1.equals(value2);
}
```

TQ: Value that might not carry a type qualifier is always used in a way requires that type qualifier (TQ_MAYBE_SOURCE_VALUE_REACHES_ALWAYS_SINK)

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.

TQ: Value that might carry a type qualifier is always used in a way prohibits it from having that type qualifier (TQ_MAYBE_SOURCE_VALUE_REACHES_NEVER_SINK)

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.

TQ: Value annotated as never carrying a type qualifier used where value carrying that qualifier is required (TQ_NEVER_VALUE_USED_WHERE_ALWAYS_REQUIRED)

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

TQ: Value without a type qualifier used where a value is required to have that qualifier (TQ_UNKNOWN_VALUE_USED_WHERE_ALWAYS_STRICTLY_REQUIRED)

A value is being used in a way that requires the value be annotation with a type qualifier. The type qualifier is strict, so the tool rejects any values that do not have the appropriate annotation.

To coerce a value to have a strict annotation, define an identity function where the return value is annotated with the strict annotation. This is the only way to turn a non-annotated value into a value with a strict type qualifier annotation.

UMAC: Uncallable method defined in anonymous class (UMAC_UNCALLABLE_METHOD_OF_ANONYMOUS_CLASS)

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.

UR: Uninitialized read of field in constructor (UR_UNINIT_READ)

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.

UR: Uninitialized read of field method called from constructor of superclass (UR_UNINIT_READ_CALLED_FROM_SUPER_CONSTRUCTOR)

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:

```
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;
  }
  }
```

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`

USELESS_STRING: Invocation of toString on an unnamed array (DMI_INVOKING_TOSTRING_ON_ANONYMOUS_ARRAY)

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.

USELESS_STRING: Invocation of toString on an array (DMI_INVOKING_TOSTRING_ON_ARRAY)

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.

USELESS_STRING: Array formatted in useless way using format string (VA_FORMAT_STRING_BAD_CONVERSION_FROM_ARRAY)

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.

UwF: Field only ever set to null (UWF_NULL_FIELD)

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.

UwF: Unwritten field (UWF_UNWRITTEN_FIELD)

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.

VA: Primitive array passed to function expecting a variable number of object arguments (VA_PRIMITIVE_ARRAY_PASSED_TO_OBJECT_VARARG)

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.

LG: Potential lost logger changes due to weak reference in OpenJDK (LG_LOST_LOGGER_DUE_TO_WEAK_REFERENCE)

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 configurati
 logger.setUseParentHandlers(false); // another call to change logger confi
}
```

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
}
```

Ulf Ochsenfahrt and Eric Fellheimer

OBL: Method may fail to clean up stream or resource (OBL_UNSATISFIED_OBLIGATION)

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, but is based on a different (and hopefully better) static analysis technique. We are interested is getting feedback about the usefulness of this bug pattern. To send feedback, either:

- send email to findbugs@cs.umd.edu
- file a bug report: http://findbugs.sourceforge.net/reportingBugs.html

In particular, the false-positive suppression heuristics for this bug pattern have not been extensively tuned, so reports about false positives are helpful to us.

See Weimer and Necula, *Finding and Preventing Run-Time Error Handling Mistakes*, for a description of the analysis technique.

OBL: Method may fail to clean up stream or resource on checked exception (OBL_UNSATISFIED_OBLIGATION_EXCEPTION_EDGE)

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, but is based on a different (and hopefully better) static analysis technique. We are interested is getting feedback about the usefulness of this bug pattern. To send feedback, either:

- send email to findbugs@cs.umd.edu
- file a bug report: http://findbugs.sourceforge.net/reportingBugs.html

In particular, the false-positive suppression heuristics for this bug pattern have not been extensively tuned, so reports about false positives are helpful to us.

See Weimer and Necula, *Finding and Preventing Run-Time Error Handling Mistakes*, for a description of the analysis technique.

Dm: Consider using Locale parameterized version of invoked method (DM_CONVERT_CASE)

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.

Dm: Reliance on default encoding (DM_DEFAULT_ENCODING)

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.

DP: Classloaders should only be created inside doPrivileged block (DP_CREATE_CLASSLOADER_INSIDE_DO_PRIVILEGED)

This code creates a classloader, which needs permission if a security manage is installed. If this code might be invoked by code that does not have security permissions, then the classloader creation needs to occur inside a doPrivileged block.

DP: Method invoked that should be only be invoked inside a doPrivileged block (DP_DO_INSIDE_DO_PRIVILEGED)

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.

EI: May expose internal representation by returning reference to mutable object (EI_EXPOSE_REP)

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.

EI2: May expose internal representation by

incorporating reference to mutable object (EI_EXPOSE_REP2)

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.

FI: Finalizer should be protected, not public (FI_PUBLIC_SHOULD_BE_PROTECTED)

A class's `finalize()` method should have protected access, not public.

MS: May expose internal static state by storing a mutable object into a static field (EI_EXPOSE_STATIC_REP2)

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.

MS: Field isn't final and can't be protected from malicious code (MS_CANNOT_BE_FINAL)

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.

MS: Public static method may expose internal representation by returning array (MS_EXPOSE_REP)

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.

MS: Field should be both final and package protected (MS_FINAL_PKGPROTECT)

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 this vulnerability.

MS: Field is a mutable array (MS_MUTABLE_ARRAY)

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.

MS: Field is a mutable collection (MS_MUTABLE_COLLECTION)

A mutable collection instance is assigned to a final static field, thus can be changed by malicious code or by accident from another package. Consider wrapping this field into Collections.unmodifiableSet/List/Map/etc. to avoid this vulnerability.

MS: Field is a mutable collection which should be package protected (MS_MUTABLE_COLLECTION_PKGPROTECT)

A mutable collection instance is assigned to a final static field, thus can be changed by malicious code or by accident from another package. The field could be made package protected to avoid this vulnerability. Alternatively you may wrap this field into Collections.unmodifiableSet/List/Map/etc. to avoid this vulnerability.

MS: Field is a mutable Hashtable (MS_MUTABLE_HASHTABLE)

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.

MS: Field should be moved out of an interface and made package protected (MS_OOI_PKGPROTECT)

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

MS: Field should be package protected (MS_PKGPROTECT)

A mutable static field could be changed by malicious code or by accident. The field could be made package protected to avoid this vulnerability.

MS: Field isn't final but should be (MS_SHOULD_BE_FINAL)

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.

MS: Field isn't final but should be refactored to be so (MS_SHOULD_BE_REFACTORED_TO_BE_FINAL)

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.

AT: Sequence of calls to concurrent abstraction may not be atomic (AT_OPERATION_SEQUENCE_ON_CONCURRENT_ABSTRACTION)

This code contains a sequence of calls to a concurrent abstraction (such as a concurrent hash map). These calls will not be executed atomically.

DC: Possible double check of field (DC_DOUBLECHECK)

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.

DC: Possible exposure of partially initialized object (DC_PARTIALLY_CONSTRUCTED)

Looks like this method uses lazy field initialization with double-checked locking. While the field is correctly declared as volatile, it's possible that the internal structure of the object is changed after the field assignment, thus another thread may see the partially initialized object.

To fix this problem consider storing the object into the local variable first and save it to the volatile field only after it's fully constructed.

DL: Synchronization on Boolean (DL_SYNCHRONIZATION_ON_BOOLEAN)

The code synchronizes on a boxed primitive constant, such as an Boolean.

```
private static Boolean inited = Boolean.FALSE;
...
  synchronized(inited) {
    if (!inited) {
       init();
       inited = Boolean.TRUE;
       }
     }
...
```

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

See CERT [CON08-J. Do not synchronize on objects that may be reused]() for more information.

DL: Synchronization on boxed primitive (DL_SYNCHRONIZATION_ON_BOXED_PRIMITIVE)

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

See CERT [CON08-J. Do not synchronize on objects that may be reused]() for more information.

DL: Synchronization on interned String (DL_SYNCHRONIZATION_ON_SHARED_CONST

ANT)

The code synchronizes on interned String.

```
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.

See CERT CON08-J. Do not synchronize on objects that may be reused for more information.

DL: Synchronization on boxed primitive values (DL_SYNCHRONIZATION_ON_UNSHARED_BOXED_PRIMITIVE)

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.

Dm: Monitor wait() called on Condition (DM_MONITOR_WAIT_ON_CONDITION)

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.

Dm: A thread was created using the default empty run method (DM_USELESS_THREAD)

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.

ESync: Empty synchronized block (ESync_EMPTY_SYNC)

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.

IS: Inconsistent synchronization (IS2_INCONSISTENT_SYNC)

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

- The class contains a mix of locked and unlocked accesses,
- The class is **not** annotated as javax.annotation.concurrent.NotThreadSafe,
- At least one locked access was performed by one of the class's own methods, and
- 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.

IS: Field not guarded against concurrent access (IS_FIELD_NOT_GUARDED)

This field is annotated with net.jcip.annotations.GuardedBy or javax.annotation.concurrent.GuardedBy, but can be accessed in a way that seems to violate those annotations.

JLM: Synchronization performed on Lock (JLM_JSR166_LOCK_MONITORENTER)

This method performs synchronization an object that implements java.util.concurrent.locks.Lock. Such an object is locked/unlocked using `acquire()`/`release()` rather than using the `synchronized (...)` construct.

JLM: Synchronization performed on util.concurrent instance (JLM_JSR166_UTILCONCURRENT_MONITORENTER)

This method performs synchronization an object that is an instance of a class from the java.util.concurrent package (or its subclasses). Instances of these classes have their own concurrency control mechanisms that are orthogonal to the synchronization provided by the Java keyword `synchronized`. For example, synchronizing on an `AtomicBoolean` will not prevent other threads from modifying the `AtomicBoolean`.

Such code may be correct, but should be carefully reviewed and documented, and may confuse people who have to maintain the code at a later date.

JLM: Using monitor style wait methods on util.concurrent abstraction (JML_JSR166_CALLING_WAIT_RATHER_THAN_AWAIT)

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 design, as other developers will find it exceptionally confusing.

LI: Incorrect lazy initialization of static field (LI_LAZY_INIT_STATIC)

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.

LI: Incorrect lazy initialization and update of static field (LI_LAZY_INIT_UPDATE_STATIC)

This method contains an unsynchronized lazy initialization of a static field. After the field is set, the object stored into that location is further updated or 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 you have a *very serious* multithreading bug, unless something else prevents any other thread from accessing the stored object until it is fully initialized.

Even if you feel confident that the method is never called by multiple threads, it might be better to not set the static field until the value you are setting it to is fully populated/initialized.

ML: Synchronization on field in futile attempt to guard that field (ML_SYNC_ON_FIELD_TO_GUARD_CHANGING_THAT_FIELD)

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;
 }
```

ML: Method synchronizes on an updated field (ML_SYNC_ON_UPDATED_FIELD)

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.

MSF: Mutable servlet field (MSF_MUTABLE_SERVLET_FIELD)

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.

MWN: Mismatched notify() (MWN_MISMATCHED_NOTIFY)

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.

MWN: Mismatched wait() (MWN_MISMATCHED_WAIT)

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.

NN: Naked notify (NN_NAKED_NOTIFY)

A call to `notify()` or `notifyAll()` 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.

NP: Synchronize and null check on the same field. (NP_SYNC_AND_NULL_CHECK_FIELD)

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.

No: Using notify() rather than notifyAll() (NO_NOTIFY_NOT_NOTIFYALL)

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.

RS: Class's readObject() method is synchronized (RS_READOBJECT_SYNC)

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.

RV: Return value of putIfAbsent ignored, value passed to putIfAbsent reused (RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED)

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 retaining a value that is not the one that 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 program will behave incorrectly.

Ru: Invokes run on a thread (did you mean to start it instead?) (RU_INVOKE_RUN)

This method explicitly invokes `run()` on an object. In general, classes implement the `Runnable` interface because they are going to have their `run()` method invoked in a new thread, in which case `Thread.start()` is the right method to call.

SC: Constructor invokes Thread.start() (SC_START_IN_CTOR)

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.

SP: Method spins on field (SP_SPIN_ON_FIELD)

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).

STCAL: Call to static Calendar (STCAL_INVOKE_ON_STATIC_CALENDAR_INSTANCE)

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.

STCAL: Call to static DateFormat (STCAL_INVOKE_ON_STATIC_DATE_FORMAT_INSTANCE)

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.

STCAL: Static Calendar field (STCAL_STATIC_CALENDAR_INSTANCE)

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.getCalendarDateFromFixedDate().

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.

STCAL: Static DateFormat

(STCAL_STATIC_SIMPLE_DATE_FORMAT_INSTANCE)

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.

SWL: Method calls Thread.sleep() with a lock held (SWL_SLEEP_WITH_LOCK_HELD)

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.

TLW: Wait with two locks held (TLW_TWO_LOCK_WAIT)

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.

UG: Unsynchronized get method, synchronized set method (UG_SYNC_SET_UNSYNC_GET)

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.

UL: Method does not release lock on all paths (UL_UNRELEASED_LOCK)

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();
    }
```

UL: Method does not release lock on all exception paths (UL_UNRELEASED_LOCK_EXCEPTION_PATH)

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();
    }
```

UW: Unconditional wait (UW_UNCOND_WAIT)

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.

VO: An increment to a volatile field isn't atomic (VO_VOLATILE_INCREMENT)

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.

VO: A volatile reference to an array doesn't treat the array elements as volatile (VO_VOLATILE_REFERENCE_TO_ARRAY)

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).

WL: Synchronization on getClass rather than class literal (WL_USING_GETCLASS_RATHER_THAN_CLASS_LITERAL)

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

WS: Class's writeObject() method is synchronized but nothing else is (WS_WRITEOBJECT_SYNC)

This class has a `writeObject()` method which is synchronized; however, no other method of the class is synchronized.

Wa: Condition.await() not in loop (WA_AWAIT_NOT_IN_LOOP)

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.

Wa: Wait not in loop (WA_NOT_IN_LOOP)

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.

Bx: Primitive value is boxed and then immediately

unboxed (BX_BOXING_IMMEDIATELY_UNBOXED)

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.

Bx: Primitive value is boxed then unboxed to perform primitive coercion (BX_BOXING_IMMEDIATELY_UNBOXED_TO_PERFORM_COERCION)

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`).

Bx: Primitive value is unboxed and coerced for ternary operator (BX_UNBOXED_AND_COERCED_FOR_TERNARY_OPERATOR)

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.

Bx: Boxed value is unboxed and then immediately reboxed (BX_UNBOXING_IMMEDIATELY_REBOXED)

A boxed value is unboxed and then immediately reboxed.

Bx: Boxing a primitive to compare (DM_BOXED_PRIMITIVE_FOR_COMPARE)

A boxed primitive is created just to call compareTo method. It's more efficient to use static compare method (for double and float since Java 1.4, for other primitive types since Java 1.7) which works on primitives directly.

Bx: Boxing/unboxing to parse a primitive (DM_BOXED_PRIMITIVE_FOR_PARSING)

A boxed primitive is created from a String, just to extract the unboxed primitive value. It is more efficient to just call the static parseXXX method.

Bx: Method allocates a boxed primitive just to call toString (DM_BOXED_PRIMITIVE_TOSTRING)

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,

Replace...	With this...
new Integer(1).toString()	Integer.toString(1)
new Long(1).toString()	Long.toString(1)
new Float(1.0).toString()	Float.toString(1.0)
new Double(1.0).toString()	Double.toString(1.0)
new Byte(1).toString()	Byte.toString(1)
new Short(1).toString()	Short.toString(1)
new Boolean(true).toString()	Boolean.toString(true)

Bx: Method invokes inefficient floating-point Number constructor; use static valueOf instead (DM_FP_NUMBER_CTOR)

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`.

Bx: Method invokes inefficient Number constructor; use static valueOf instead (DM_NUMBER_CTOR)

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`.

Dm: The equals and hashCode methods of URL are blocking (DMI_BLOCKING_METHODS_ON_URL)

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.

Dm: Maps and sets of URLs can be performance hogs (DMI_COLLECTION_OF_URLS)

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.

Dm: Method invokes inefficient Boolean constructor; use Boolean.valueOf(...) instead (DM_BOOLEAN_CTOR)

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.

Dm: Explicit garbage collection; extremely dubious except in benchmarking code (DM_GC)

Code explicitly invokes garbage collection. Except for specific use in benchmarking, this is very dubious.

In the past, situations where people have explicitly invoked the garbage collector in routines such as close or finalize methods has led to huge performance black holes. Garbage collection can be expensive. Any situation that forces hundreds or thousands of garbage collections will bring the machine to a crawl.

Dm: Method allocates an object, only to get the class object (DM_NEW_FOR_GETCLASS)

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.

Dm: Use the nextInt method of Random rather

than nextDouble to generate a random integer (DM_NEXTINT_VIA_NEXTDOUBLE)

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)`.

The argument to nextInt must be positive. If, for example, you want to generate a random value from -99 to 0, use `-r.nextInt(100)`.

Dm: Method invokes inefficient new String(String) constructor (DM_STRING_CTOR)

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.

Dm: Method invokes toString() method on a String (DM_STRING_TOSTRING)

Calling `String.toString()` is just a redundant operation. Just use the String.

Dm: Method invokes inefficient new String() constructor (DM_STRING_VOID_CTOR)

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.

HSC: Huge string constants is duplicated across multiple class files (HSC_HUGE_SHARED_STRING_CONSTANT)

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.

SBSC: Method concatenates strings using + in a loop (SBSC_USE_STRINGBUFFER_CONCATENATION)

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.

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();
```

SIC: Should be a static inner class (SIC_INNER_SHOULD_BE_STATIC)

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.

SIC: Could be refactored into a named static inner class (SIC_INNER_SHOULD_BE_STATIC_ANON)

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.

SIC: Could be refactored into a static inner class (SIC_INNER_SHOULD_BE_STATIC_NEEDS_THIS)

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.

SS: Unread field: should this field be static? (SS_SHOULD_BE_STATIC)

This class contains an instance final field that is initialized to a compile-time static value. Consider making the field static.

UM: Method calls static Math class method on a

constant value (UM_UNNECESSARY_MATH)

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:

Method	Parameter
abs	-any-
acos	0.0 or 1.0
asin	0.0 or 1.0
atan	0.0 or 1.0
atan2	0.0
cbrt	0.0 or 1.0
ceil	-any-
cos	0.0
cosh	0.0
exp	0.0 or 1.0
expm1	0.0
floor	-any-
log	0.0 or 1.0
log10	0.0 or 1.0
rint	-any-
round	-any-
sin	0.0
sinh	0.0
sqrt	0.0 or 1.0
tan	0.0
tanh	0.0
toDegrees	0.0 or 1.0
toRadians	0.0

UPM: Private method is never called (UPM_UNCALLED_PRIVATE_METHOD)

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.

UrF: Unread field (URF_UNREAD_FIELD)

This field is never read. Consider removing it from the class.

UuF: Unused field (UUF_UNUSED_FIELD)

This field is never used. Consider removing it from the class.

WMI: Inefficient use of keySet iterator instead of entrySet iterator (WMI_WRONG_MAP_ITERATOR)

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.

Dm: Hardcoded constant database password (DMI_CONSTANT_DB_PASSWORD)

This code creates a database connect using a hardcoded, constant password. Anyone with access to either the source code or the compiled code can easily learn the password.

Dm: Empty database password (DMI_EMPTY_DB_PASSWORD)

This code creates a database connect using a blank or empty password. This indicates that the database is not protected by a password.

HRS: HTTP cookie formed from untrusted input (HRS_REQUEST_PARAMETER_TO_COOKIE)

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.

HRS: HTTP Response splitting vulnerability (HRS_REQUEST_PARAMETER_TO_HTTP_HEADER)

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.

PT: Absolute path traversal in servlet (PT_ABSOLUTE_PATH_TRAVERSAL)

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.

PT: Relative path traversal in servlet (PT_RELATIVE_PATH_TRAVERSAL)

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.

SQL: Nonconstant string passed to execute or addBatch method on an SQL statement (SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE)

The method invokes the execute or addBatch 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.

SQL: A prepared statement is generated from a nonconstant String (SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING)

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.

XSS: JSP reflected cross site scripting vulnerability (XSS_REQUEST_PARAMETER_TO_JSP_WRITER)

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.

XSS: Servlet reflected cross site scripting vulnerability in error page (XSS_REQUEST_PARAMETER_TO_SEND_ERROR)

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.

XSS: Servlet reflected cross site scripting vulnerability (XSS_REQUEST_PARAMETER_TO_SERVLET_WRITER)

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.

BC: Questionable cast to abstract collection (BC_BAD_CAST_TO_ABSTRACT_COLLECTION)

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.

BC: Questionable cast to concrete collection (BC_BAD_CAST_TO_CONCRETE_COLLECTION)

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.

BC: Unchecked/unconfirmed cast (BC_UNCONFIRMED_CAST)

This cast is unchecked, and not all instances of the type casted from can be cast to the type it is being cast to. Check that your program logic ensures that this cast will not fail.

BC: Unchecked/unconfirmed cast of return value from method (BC_UNCONFIRMED_CAST_OF_RETURN_VALUE)

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.

BC: instanceof will always return true (BC_VACUOUS_INSTANCEOF)

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.

BSHIFT: Unsigned right shift cast to short/byte (ICAST_QUESTIONABLE_UNSIGNED_RIGHT_SHIFT)

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).

CI: Class is final but declares protected field (CI_CONFUSED_INHERITANCE)

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.

DB: Method uses the same code for two branches (DB_DUPLICATE_BRANCHES)

This method uses the same code to implement two branches of a conditional branch. Check to ensure that this isn't a coding mistake.

DB: Method uses the same code for two switch clauses (DB_DUPLICATE_SWITCH_CLAUSES)

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.

DLS: Dead store to local variable (DLS_DEAD_LOCAL_STORE)

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.

DLS: Useless assignment in return statement (DLS_DEAD_LOCAL_STORE_IN_RETURN)

This statement assigns to a local variable in a return statement. This assignment has effect. Please verify that this statement does the right thing.

DLS: Dead store of null to local variable

(DLS_DEAD_LOCAL_STORE_OF_NULL)

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.

DLS: Dead store to local variable that shadows field (DLS_DEAD_LOCAL_STORE_SHADOWS_FIELD)

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?

DMI: Code contains a hard coded reference to an absolute pathname (DMI_HARDCODED_ABSOLUTE_FILENAME)

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");`

DMI: Non serializable object written to ObjectOutput (DMI_NONSERIALIZABLE_OBJECT_WRITTEN)

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.

DMI: Invocation of substring(0), which returns the original value (DMI_USELESS_SUBSTRING)

This code invokes substring(0) on a String, which returns the original value.

Dm: Thread passed where Runnable expected (DMI_THREAD_PASSED_WHERE_RUNNABLE_EXPECTED)

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.

Eq: Class doesn't override equals in superclass (EQ_DOESNT_OVERRIDE_EQUALS)

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).

Eq: Unusual equals method (EQ_UNUSUAL)

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.

FE: Test for floating point equality (FE_FLOATING_POINT_EQUALITY)

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.

FS: Non-Boolean argument formatted using %b format specifier (VA_FORMAT_STRING_BAD_CONVERSION_TO_BOOLEAN)

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 non-null value, and false for null. This feature of format strings is strange, and may not be what you intended.

IA: Potentially ambiguous invocation of either an inherited or outer method (IA_AMBIGUOUS_INVOCATION_OF_INHERITED_OR_OUTER_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. For example, you invoke `foo(17)`, which is defined in both a superclass and in an outer method. 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 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.

If you call `this.foo(17)`, then the inherited method will be invoked. However, since FindBugs only looks at classfiles, it can't tell the difference between an invocation of `this.foo(17)` and `foo(17)`, it will still complain about a potential ambiguous invocation.

IC: Initialization circularity (IC_INIT_CIRCULARITY)

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.

ICAST: Integral division result cast to double or float (ICAST_IDIV_CAST_TO_DOUBLE)

This code casts the result of an integral division (e.g., int or long 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;
```

ICAST: Result of integer multiplication cast to long (ICAST_INTEGER_MULTIPLY_CAST_TO_LONG)

This code performs integer multiply and then converts the result to a long, as in:

```
long convertDaysToMilliseconds(int days) { return 1000*3600*24*days; }
```

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 1000L*3600*24*days; }
```

or

```
    static final long MILLISECONDS_PER_DAY = 24L*3600*1000;
    long convertDaysToMilliseconds(int days) { return days * MILLISECONDS_P
```

IM: Computation of average could overflow (IM_AVERAGE_COMPUTATION_COULD_OVERFLOW)

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.

IM: Check for oddness that won't work for negative numbers (IM_BAD_CHECK_FOR_ODD)

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.

INT: Integer remainder modulo 1 (INT_BAD_REM_BY_1)

Any expression (exp % 1) is guaranteed to always return zero. Did you mean (exp & 1) or (exp % 2) instead?

INT: Vacuous bit mask operation on integer value (INT_VACUOUS_BIT_OPERATION)

This is an integer bit operation (and, or, or exclusive or) that doesn't do any useful work (e.g., `v & 0xffffffff`).

INT: Vacuous comparison of integer value (INT_VACUOUS_COMPARISON)

There is an integer comparison that always returns the same value (e.g., x <= Integer.MAX_VALUE).

MTIA: Class extends Servlet class and uses instance variables (MTIA_SUSPECT_SERVLET_INSTANCE_FIELD)

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.

MTIA: Class extends Struts Action class and uses instance variables (MTIA_SUSPECT_STRUTS_INSTANCE_FIELD)

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.

NP: Dereference of the result of readLine() without nullcheck (NP_DEREFERENCE_OF_READLINE_VALUE)

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.

NP: Immediate dereference of the result of

readLine() (NP_IMMEDIATE_DEREFERENCE_OF_READLINE)

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.

NP: Load of known null value (NP_LOAD_OF_KNOWN_NULL_VALUE)

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 non-null).

NP: Method tightens nullness annotation on parameter (NP_METHOD_PARAMETER_TIGHTENS_ANNOTATION)

A method should always implement the contract of a method it overrides. Thus, if a method takes a parameter that is marked as @Nullable, you shouldn't override that method in a subclass with a method where that parameter is @Nonnull. Doing so violates the contract that the method should handle a null parameter.

NP: Method relaxes nullness annotation on return value (NP_METHOD_RETURN_RELAXING_ANNOTATION)

A method should always implement the contract of a method it overrides. Thus, if a method takes is annotated as returning a @Nonnull value, you shouldn't override that method in a subclass with a method annotated as returning a @Nullable or @CheckForNull value. Doing so violates the contract that the method shouldn't return null.

NP: Possible null pointer dereference due to return value of called method (NP_NULL_ON_SOME_PATH_FROM_RETURN_VALUE)

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. This may lead to a `NullPointerException` when the code is executed.

NP: Possible null pointer dereference on branch that might be infeasible (NP_NULL_ON_SOME_PATH_MIGHT_BE_INFEASIBLE)

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.

NP: Parameter must be non-null but is marked as nullable (NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULLABLE)

This parameter is always used in a way that requires it to be non-null, but the parameter is explicitly annotated as being Nullable. Either the use of the parameter or the annotation is wrong.

NP: Read of unwritten public or protected field (NP_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD)

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.

NS: Potentially dangerous use of non-short-circuit logic (NS_DANGEROUS_NON_SHORT_CIRCUIT)

This code seems to be using non-short-circuit logic (e.g., & or |) rather than short-circuit logic (&& or ||). In addition, it seem possible that, depending on the value of the left hand side, you might not want to evaluate the right hand side (because it would have side effects, could cause an exception or could be expensive.

Non-short-circuit logic causes both sides of the expression to be evaluated even when the result can be inferred from knowing the left-hand side. This can be less efficient and can result in errors if the left-hand side guards cases when evaluating the right-hand side can generate an error.

See the Java Language Specification for details

NS: Questionable use of non-short-circuit logic (NS_NON_SHORT_CIRCUIT)

This code seems to be using non-short-circuit logic (e.g., & or |) rather than short-circuit logic (&& or ||). Non-short-circuit logic causes both sides of the expression to be evaluated even when the result can be inferred from knowing the left-hand side. This can be less efficient and can result in errors if the left-hand side guards cases when evaluating the right-hand side can generate an error.

See the Java Language Specification for details

PZLA: Consider returning a zero length array rather than null (PZLA_PREFER_ZERO_LENGTH_ARRAYS)

It is often a better design to return a length zero array rather than a null reference to indicate that there are no results (i.e., an empty list of results). This way, no explicit check for null is needed by clients of the method.

On the other hand, using null to indicate "there is no answer to this question" is probably appropriate. For example, `File.listFiles()` returns an empty list if given a directory containing no files, and returns null if the file is not a directory.

QF: Complicated, subtle or wrong increment in for-loop (QF_QUESTIONABLE_FOR_LOOP)

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.

RCN: Redundant comparison of non-null value to null (RCN_REDUNDANT_COMPARISON_OF_NULL_AND_NONNULL_VALUE)

This method contains a reference known to be non-null with another reference known to be null.

RCN: Redundant comparison of two null values (RCN_REDUNDANT_COMPARISON_TWO_NULL_VALUES)

This method contains a redundant comparison of two references known to both be definitely null.

RCN: Redundant nullcheck of value known to be non-null

(RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE)

This method contains a redundant check of a known non-null value against the constant null.

RCN: Redundant nullcheck of value known to be null (RCN_REDUNDANT_NULLCHECK_OF_NULL_VALUE)

This method contains a redundant check of a known null value against the constant null.

REC: Exception is caught when Exception is not thrown (REC_CATCH_EXCEPTION)

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.

A better approach is to either explicitly catch the specific exceptions that are thrown, or to explicitly catch RuntimeException exception, rethrow it, and then catch all non-Runtime Exceptions, as shown below:

```
try {
  ...
} catch (RuntimeException e) {
  throw e;
} catch (Exception e) {
  ... deal with all non-runtime exceptions ...
}
```

RI: Class implements same interface as superclass

(RI_REDUNDANT_INTERFACES)

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.

RV: Method checks to see if result of String.indexOf is positive (RV_CHECK_FOR_POSITIVE_INDEXOF)

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.

RV: Method discards result of readLine after checking if it is non-null (RV_DONT_JUST_NULL_CHECK_READLINE)

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.

RV: Remainder of hashCode could be negative (RV_REM_OF_HASHCODE)

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)`

RV: Remainder of 32-bit signed random integer (RV_REM_OF_RANDOM_INT)

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.

RV: Method ignores return value, is this OK? (RV_RETURN_VALUE_IGNORED_INFERRED)

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.

RV: Return value of method without side effect is ignored (RV_RETURN_VALUE_IGNORED_NO_SIDE_EFFECT)

This code calls a method and ignores the return value. However our analysis shows that the method (including its implementations in subclasses if any) does not produce any effect other than return value. Thus this call can be removed.

We are trying to reduce the false positives as much as possible, but in some cases this warning might be wrong. Common false-positive cases include:

- The method is designed to be overridden and produce a side effect in other projects which are out of the scope of the analysis.

- The method is called to trigger the class loading which may have a side effect.

- The method is called just to get some exception.

If you feel that our assumption is incorrect, you can use a @CheckReturnValue annotation to instruct FindBugs that ignoring the return value of this method is acceptable.

SA: Double assignment of field (SA_FIELD_DOUBLE_ASSIGNMENT)

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.

SA: Double assignment of local variable (SA_LOCAL_DOUBLE_ASSIGNMENT)

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.

SA: Self assignment of local variable (SA_LOCAL_SELF_ASSIGNMENT)

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.

SF: Switch statement found where one case falls through to the next case (SF_SWITCH_FALLTHROUGH)

This method contains a switch statement where one case branch will fall through to the next case. Usually you need to end this case with a break or return.

SF: Switch statement found where default case is missing (SF_SWITCH_NO_DEFAULT)

This method contains a switch statement where default case is missing. Usually you need to provide a default case.

Because the analysis only looks at the generated bytecode, this warning can be incorrect triggered if the default case is at the end of the switch statement and the switch statement doesn't contain break statements for other cases.

ST: Write to static field from instance method (ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD)

This instance method writes to a static field. This is tricky to get correct if multiple instances are being manipulated, and generally bad practice.

Se: Private readResolve method not inherited by subclasses (SE_PRIVATE_READ_RESOLVE_NOT_INHERITED)

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.

Se: Transient field of class that isn't Serializable. (SE_TRANSIENT_FIELD_OF_NONSERIALIZABLE_CLASS)

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.

TQ: Value required to have type qualifier, but marked as unknown (TQ_EXPLICIT_UNKNOWN_SOURCE_VALUE_REACHES_ALWAYS_SINK)

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.

TQ: Value required to not have type qualifier, but marked as unknown (TQ_EXPLICIT_UNKNOWN_SOURCE_VALUE_REACHES_NEVER_SINK)

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.

UC: Condition has no effect (UC_USELESS_CONDITION)

This condition always produces the same result as the value of the involved variable was narrowed before. Probably something else was meant or condition can be removed.

UC: Condition has no effect due to the variable type (UC_USELESS_CONDITION_TYPE)

This condition always produces the same result due to the type range of the involved variable. Probably something else was meant or condition can be removed.

UC: Useless object created (UC_USELESS_OBJECT)

Our analysis shows that this object is useless. It's created and modified, but its value never go outside of the method or produce any side-effect. Either there is a mistake and object was intended to be used or it can be removed.

This analysis rarely produces false-positives. Common false-positive cases include:

- This object used to implicitly throw some obscure exception.

- This object used as a stub to generalize the code.

- This object used to hold strong references to weak/soft-referenced objects.

UC: Useless object created on stack (UC_USELESS_OBJECT_STACK)

This object is created just to perform some modifications which don't have any side-effect. Probably something else was meant or the object can be removed.

UC: Useless non-empty void method (UC_USELESS_VOID_METHOD)

Our analysis shows that this non-empty void method does not actually perform any useful work. Please check it: probably there's a mistake in its code or its body can be fully removed.

We are trying to reduce the false positives as much as possible, but in some cases this warning might be wrong. Common false-positive cases include:

- The method is intended to trigger loading of some class which may have a side effect.

- The method is intended to implicitly throw some obscure exception.

UCF: Useless control flow (UCF_USELESS_CONTROL_FLOW)

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
    }
```

UCF: Useless control flow to next line (UCF_USELESS_CONTROL_FLOW_NEXT_LINE)

This method contains a useless control flow statement in which control flow follows to the same or following line regardless of whether or not 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]);
```

UrF: Unread public/protected field (URF_UNREAD_PUBLIC_OR_PROTECTED_FIELD)

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.

UuF: Unused public or protected field (UUF_UNUSED_PUBLIC_OR_PROTECTED_FIELD)

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.

UwF: Field not initialized in constructor but dereferenced without null check (UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR)

This field is never initialized within any constructor, and is therefore could be null after the object is constructed. Elsewhere, it is loaded and dereferenced without a null check. This could be a either an error or a questionable design, since it means a null pointer exception will be generated if that field is dereferenced before being initialized.

UwF: Unwritten public or protected field (UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD)

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.

XFB: Method directly allocates a specific implementation of xml interfaces (XFB_XML_FACTORY_BYPASS)

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.

Last updated 03/06/2015 13:06:03.

Send comments to findbugs@cs.umd.edu