Android Rules

 CallSuperFirst: Super should be called at the start of the method

 CallSuperLast: Super should be called at the end of the method

 ProtectLogD: Log.d calls should be protected by checking Config.LOGD first

 ProtectLogV: Log.v calls should be protected by checking Config.LOGV first

Basic JSF rules

 DontNestJsfInJstlIteration: Do not nest JSF component custom actions inside a custom action that iterates over its body.

Basic JSP rules

 NoLongScripts: Scripts should be part of Tag Libraries, rather than part of JSP pages.

 NoScriptlets: Scriptlets should be factored into Tag Libraries or JSP declarations, rather than being part of JSP pages.

 NoInlineStyleInformation: Style information should be put in CSS files, not in JSPs. Therefore, don't use <B> or <FONT> tags, or attributes like "align='center'".

 NoClassAttribute: Do not use an attribute called 'class'. Use "styleclass" for CSS styles.

 NoJspForward: Do not do a forward from within a JSP file.

 IframeMissingSrcAttribute: IFrames which are missing a src element can cause security information popups in IE if you are accessing the page through SSL. See http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q261188

 NoHtmlComments: In a production system, HTML comments increase the payload between the application server to the client, and serve little other purpose. Consider switching to JSP comments.

 DuplicateJspImports: Avoid duplicate import statements inside JSP's.

 JspEncoding: A missing 'meta' tag or page directive will trigger this rule, as well as a non-UTF-8 charset.

Basic Rules

 EmptyCatchBlock: Empty Catch Block finds instances where an exception is caught, but nothing is done. In most circumstances, this swallows an exception which should either be acted on or reported.

 EmptyIfStmt: Empty If Statement finds instances where a condition is checked but nothing is done about it.

 EmptyWhileStmt: Empty While Statement finds all instances where a while statement does nothing. If it is a timing loop, then you should use Thread.sleep() for it; if it's a while loop that does a lot in the exit expression, rewrite it to make it clearer.

 EmptyTryBlock: Avoid empty try blocks - what's the point?

 EmptyFinallyBlock: Avoid empty finally blocks - these can be deleted.

 EmptySwitchStatements: Avoid empty switch statements.

 JumbledIncrementer: Avoid jumbled loop incrementers - it's usually a mistake, and it's confusing even if it's what's intended.

 ForLoopShouldBeWhileLoop: Some for loops can be simplified to while loops - this makes them more concise.

 UnnecessaryConversionTemporary: Avoid unnecessary temporaries when converting primitives to Strings

 OverrideBothEqualsAndHashcode: Override both public boolean Object.equals(Object other), and public int Object.hashCode(), or override neither. Even if you are inheriting a hashCode() from a parent class, consider implementing hashCode and explicitly delegating to your superclass.

 DoubleCheckedLocking: Partially created objects can be returned by the Double Checked Locking pattern when used in Java. An optimizing JRE may assign a reference to the baz variable before it creates the object the reference is intended to point to. For more details see http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-double.html.

 ReturnFromFinallyBlock: Avoid returning from a finally block - this can discard exceptions.

 EmptySynchronizedBlock: Avoid empty synchronized blocks - they're useless.

 UnnecessaryReturn: Avoid unnecessary return statements

 EmptyStaticInitializer: An empty static initializer was found.

 UnconditionalIfStatement: Do not use "if" statements that are always true or always false.

 EmptyStatementNotInLoop: An empty statement (aka a semicolon by itself) that is not used as the sole body of a for loop or while loop is probably a bug. It could also be a double semicolon, which is useless and should be removed.

 BooleanInstantiation: Avoid instantiating Boolean objects; you can reference Boolean.TRUE, Boolean.FALSE, or call Boolean.valueOf() instead.

 UnnecessaryFinalModifier: When a class has the final modifier, all the methods are automatically final.

 CollapsibleIfStatements: Sometimes two 'if' statements can be consolidated by separating their conditions with a boolean short-circuit operator.

 UselessOverridingMethod: The overriding method merely calls the same method defined in a superclass

 ClassCastExceptionWithToArray: if you need to get an array of a class from your Collection, you should pass an array of the desidered class as the parameter of the toArray method. Otherwise you will get a ClassCastException.

 AvoidDecimalLiteralsInBigDecimalConstructor: One might assume that "new BigDecimal(.1)" is exactly equal to .1, but it is actually equal to .1000000000000000055511151231257827021181583404541015625. This is so because .1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the long value that is being passed in to the constructor is not exactly equal to .1, appearances notwithstanding. The (String) constructor, on the other hand, is perfectly predictable: 'new BigDecimal(".1")' is exactly equal to .1, as one would expect. Therefore, it is generally recommended that the (String) constructor be used in preference to this one.

 UselessOperationOnImmutable: An operation on an Immutable object (String, BigDecimal or BigInteger) won't change the object itself. The result of the operation is a new object. Therefore, ignoring the operation result is an error.

 MisplacedNullCheck: The null check here is misplaced. if the variable is null you'll get a NullPointerException. Either the check is useless (the variable will never be "null") or it's incorrect.

 UnusedNullCheckInEquals: After checking an object reference for null, you should invoke equals() on that object rather than passing it to another object's equals() method.

 AvoidThreadGroup: Avoid using ThreadGroup; although it is intended to be used in a threaded environment it contains methods that are not thread safe.

 BrokenNullCheck: The null check is broken since it will throw a NullPointerException itself. It is likely that you used || instead of && or vice versa.

 BigIntegerInstantiation: Don't create instances of already existing BigInteger (BigInteger.ZERO, BigInteger.ONE) and for 1.5 on, BigInteger.TEN and BigDecimal (BigDecimal.ZERO, BigDecimal.ONE, BigDecimal.TEN)

 AvoidUsingOctalValues: Integer literals should not start with zero. Zero means that the rest of literal will be interpreted as an octal value.

 AvoidUsingHardCodedIP: An application with hard coded IP may become impossible to deploy in some case. It never hurts to externalize IP adresses.

 CheckResultSet: Always check the return of one of the navigation method (next,previous,first,last) of a ResultSet. Indeed, if the value return is 'false', the developer should deal with it !

 AvoidMultipleUnaryOperators: Using multiple unary operators may be a bug, and/or is confusing. Check the usage is not a bug, or consider simplifying the expression.

 EmptyInitializer: An empty initializer was found.

Braces Rules

 IfStmtsMustUseBraces: Avoid using if statements without using curly braces.

 WhileLoopsMustUseBraces: Avoid using 'while' statements without using curly braces.

 IfElseStmtsMustUseBraces: Avoid using if..else statements without using curly braces.

 ForLoopsMustUseBraces: Avoid using 'for' statements without using curly braces.

Clone Implementation Rules

 ProperCloneImplementation: Object clone() should be implemented with super.clone().

 CloneThrowsCloneNotSupportedException: The method clone() should throw a CloneNotSupportedException.

 CloneMethodMustImplementCloneable: The method clone() should only be implemented if the class implements the Cloneable interface with the exception of a final method that only throws CloneNotSupportedException.

Code Size Rules

 NPathComplexity: The NPath complexity of a method is the number of acyclic execution paths through that method. A threshold of 200 is generally considered the point where measures should be taken to reduce complexity.

 ExcessiveMethodLength: Violations of this rule usually indicate that the method is doing too much. Try to reduce the method size by creating helper methods and removing any copy/pasted code.

 ExcessiveParameterList: Long parameter lists can indicate that a new object should be created to wrap the numerous parameters. Basically, try to group the parameters together.

 ExcessiveClassLength: Long Class files are indications that the class may be trying to do too much. Try to break it down, and reduce the size to something manageable.

 CyclomaticComplexity: Complexity is determined by the number of decision points in a method plus one for the method entry. The decision points are 'if', 'while', 'for', and 'case labels'. Generally, 1-4 is low complexity, 5-7 indicates moderate complexity, 8-10 is high complexity, and 11+ is very high complexity.

 ExcessivePublicCount: A large number of public methods and attributes declared in a class can indicate the class may need to be broken up as increased effort will be required to thoroughly test it.

 TooManyFields: Classes that have too many fields could be redesigned to have fewer fields, possibly through some nested object grouping of some of the information. For example, a class with city/state/zip fields could instead have one Address field.

 NcssMethodCount: This rule uses the NCSS (Non Commenting Source Statements) algorithm to determine the number of lines of code for a given method. NCSS ignores comments, and counts actual statements. Using this algorithm, lines of code that are split are counted as one.

 NcssTypeCount: This rule uses the NCSS (Non Commenting Source Statements) algorithm to determine the number of lines of code for a given type. NCSS ignores comments, and counts actual statements. Using this algorithm, lines of code that are split are counted as one.

 NcssConstructorCount: This rule uses the NCSS (Non Commenting Source Statements) algorithm to determine the number of lines of code for a given constructor. NCSS ignores comments, and counts actual statements. Using this algorithm, lines of code that are split are counted as one.

 TooManyMethods: A class with too many methods is probably a good suspect for refactoring, in order to reduce its complexity and find a way to have more fine grained objects.

Controversial Rules

 UnnecessaryConstructor: This rule detects when a constructor is not necessary; i.e., when there's only one constructor, it's public, has an empty body, and takes no arguments.

 NullAssignment: Assigning a "null" to a variable (outside of its declaration) is usually bad form. Some times, the assignment is an indication that the programmer doesn't completely understand what is going on in the code. NOTE: This sort of assignment may in rare cases be useful to encourage garbage collection. If that's what you're using it for, by all means, disregard this rule :-)

 OnlyOneReturn: A method should have only one exit point, and that should be the last statement in the method.

 UnusedModifier: Fields in interfaces are automatically public static final, and methods are public abstract. Classes or interfaces nested in an interface are automatically public and static (all nested interfaces are automatically static). For historical reasons, modifiers which are implied by the context are accepted by the compiler, but are superfluous.

 AssignmentInOperand: Avoid assignments in operands; this can make code more complicated and harder to read.

 AtLeastOneConstructor: Each class should declare at least one constructor.

 DontImportSun: Avoid importing anything from the 'sun.\*' packages. These packages are not portable and are likely to change.

 SuspiciousOctalEscape: A suspicious octal escape sequence was found inside a String literal. The Java language specification (section 3.10.6) says an octal escape sequence inside a literal String shall consist of a backslash followed by: OctalDigit | OctalDigit OctalDigit | ZeroToThree OctalDigit OctalDigit Any octal escape sequence followed by non-octal digits can be confusing, e.g. "\038" is interpreted as the octal escape sequence "\03" followed by the literal character "8".

 CallSuperInConstructor: It is a good practice to call super() in a constructor. If super() is not called but another constructor (such as an overloaded constructor) is called, this rule will not report it.

 UnnecessaryParentheses: Sometimes expressions are wrapped in unnecessary parentheses, making them look like a function call.

 DefaultPackage: Use explicit scoping instead of the default package private level.

 BooleanInversion: Use bitwise inversion to invert boolean values - it's the fastest way to do this. See http://www.javaspecialists.co.za/archive/newsletter.do?issue=042&locale=en\_US for specific details

 DataflowAnomalyAnalysis: The dataflow analysis tracks local definitions, undefinitions and references to variables on different paths on the data flow. From those informations there can be found various problems. 1. UR - Anomaly: There is a reference to a variable that was not defined before. This is a bug and leads to an error. 2. DU - Anomaly: A recently defined variable is undefined. These anomalies may appear in normal source text. 3. DD - Anomaly: A recently defined variable is redefined. This is ominous but don't have to be a bug.

 AvoidFinalLocalVariable: Avoid using final local variables, turn them into fields.

 AvoidUsingShortType: Java uses the 'short' type to reduce memory usage, not to optimize calculation. In fact, the jvm does not have any arithmetic capabilities for the short type: the jvm must convert the short into an int, do the proper caculation and convert the int back to a short. So, the use of the 'short' type may have a greater impact than memory usage.

 AvoidUsingVolatile: Use of the keyword 'volatile' is general used to fine tune a Java application, and therefore, requires a good expertise of the Java Memory Model. Moreover, its range of action is somewhat misknown. Therefore, the volatile keyword should not be used for maintenance purpose and portability.

 AvoidUsingNativeCode: As JVM and Java language offer already many help in creating application, it should be very rare to have to rely on non-java code. Even though, it is rare to actually have to use Java Native Interface (JNI). As the use of JNI make application less portable, and harder to maintain, it is not recommended.

 AvoidAccessibilityAlteration: Methods such as getDeclaredConstructors(), getDeclaredConstructor(Class[]) and setAccessible(), as the interface PrivilegedAction, allow to alter, at runtime, the visilibilty of variable, classes, or methods, even if they are private. Obviously, no one should do so, as such behavior is against everything encapsulation principal stands for.

 DoNotCallGarbageCollectionExplicitly: Calls to System.gc(), Runtime.getRuntime().gc(), and System.runFinalization() are not advised. Code should have the same behavior whether the garbage collection is disabled using the option -Xdisableexplicitgc or not. Moreover, "modern" jvms do a very good job handling garbage collections. If memory usage issues unrelated to memory leaks develop within an application, it should be dealt with JVM options rather than within the code itself.

Coupling Rules

 CouplingBetweenObjects: This rule counts unique attributes, local variables and return types within an object. A number higher than specified threshold can indicate a high degree of coupling.

 ExcessiveImports: A high number of imports can indicate a high degree of coupling within an object. Rule counts the number of unique imports and reports a violation if the count is above the user defined threshold.

 LooseCoupling: Avoid using implementation types (i.e., HashSet); use the interface (i.e, Set) instead

Design Rules

 UseSingleton: If you have a class that has nothing but static methods, consider making it a Singleton. Note that this doesn't apply to abstract classes, since their subclasses may well include non-static methods. Also, if you want this class to be a Singleton, remember to add a private constructor to prevent instantiation.

 SimplifyBooleanReturns: Avoid unnecessary if..then..else statements when returning a boolean.

 SimplifyBooleanExpressions: Avoid unnecessary comparisons in boolean expressions - this complicates simple code.

 SwitchStmtsShouldHaveDefault: Switch statements should have a default label.

 AvoidDeeplyNestedIfStmts: Deeply nested if..then statements are hard to read.

 AvoidReassigningParameters: Reassigning values to parameters is a questionable practice. Use a temporary local variable instead.

 SwitchDensity: A high ratio of statements to labels in a switch statement implies that the switch statement is doing too much work. Consider moving the statements into new methods, or creating subclasses based on the switch variable.

 ConstructorCallsOverridableMethod: Calling overridable methods during construction poses a risk of invoking methods on an incompletely constructed object and can be difficult to discern. It may leave the sub-class unable to construct its superclass or forced to replicate the construction process completely within itself, losing the ability to call super(). If the default constructor contains a call to an overridable method, the subclass may be completely uninstantiable. Note that this includes method calls throughout the control flow graph - i.e., if a constructor Foo() calls a private method bar() that calls a public method buz(), this denotes a problem.

 AccessorClassGeneration: Instantiation by way of private constructors from outside of the constructor's class often causes the generation of an accessor. A factory method, or non-privitization of the constructor can eliminate this situation. The generated class file is actually an interface. It gives the accessing class the ability to invoke a new hidden package scope constructor that takes the interface as a supplementary parameter. This turns a private constructor effectively into one with package scope, and is challenging to discern.

 FinalFieldCouldBeStatic: If a final field is assigned to a compile-time constant, it could be made static, thus saving overhead in each object at runtime.

 CloseResource: Ensure that resources (like Connection, Statement, and ResultSet objects) are always closed after use.

 NonStaticInitializer: A nonstatic initializer block will be called any time a constructor is invoked (just prior to invoking the constructor). While this is a valid language construct, it is rarely used and is confusing.

 DefaultLabelNotLastInSwitchStmt: By convention, the default label should be the last label in a switch statement.

 NonCaseLabelInSwitchStatement: A non-case label (e.g. a named break/continue label) was present in a switch statement. This legal, but confusing. It is easy to mix up the case labels and the non-case labels.

 OptimizableToArrayCall: A call to Collection.toArray can use the Collection's size vs an empty Array of the desired type.

 BadComparison: Avoid equality comparisons with Double.NaN - these are likely to be logic errors.

 EqualsNull: Inexperienced programmers sometimes confuse comparison concepts and use equals() to compare to null.

 ConfusingTernary: In an "if" expression with an "else" clause, avoid negation in the test. For example, rephrase: if (x != y) diff(); else same(); as: if (x == y) same(); else diff(); Most "if (x != y)" cases without an "else" are often return cases, so consistent use of this rule makes the code easier to read. Also, this resolves trivial ordering problems, such as "does the error case go first?" or "does the common case go first?".

 InstantiationToGetClass: Avoid instantiating an object just to call getClass() on it; use the .class public member instead.

 IdempotentOperations: Avoid idempotent operations - they are have no effect.

 SimpleDateFormatNeedsLocale: Be sure to specify a Locale when creating a new instance of SimpleDateFormat.

 ImmutableField: Identifies private fields whose values never change once they are initialized either in the declaration of the field or by a constructor. This aids in converting existing classes to immutable classes.

 UseLocaleWithCaseConversions: When doing a String.toLowerCase()/toUpperCase() call, use a Locale. This avoids problems with certain locales, i.e. Turkish.

 AvoidProtectedFieldInFinalClass: Do not use protected fields in final classes since they cannot be subclassed. Clarify your intent by using private or package access modifiers instead.

 AssignmentToNonFinalStatic: Identifies a possible unsafe usage of a static field.

 MissingStaticMethodInNonInstantiatableClass: A class that has private constructors and does not have any static methods or fields cannot be used.

 AvoidSynchronizedAtMethodLevel: Method level synchronization can backfire when new code is added to the method. Block-level synchronization helps to ensure that only the code that needs synchronization gets it.

 MissingBreakInSwitch: A switch statement without an enclosed break statement may be a bug.

 UseNotifyAllInsteadOfNotify: Thread.notify() awakens a thread monitoring the object. If more than one thread is monitoring, then only one is chosen. The thread chosen is arbitrary; thus it's usually safer to call notifyAll() instead.

 AvoidInstanceofChecksInCatchClause: Each caught exception type should be handled in its own catch clause.

 AbstractClassWithoutAbstractMethod: The abstract class does not contain any abstract methods. An abstract class suggests an incomplete implementation, which is to be completed by subclasses implementing the abstract methods. If the class is intended to be used as a base class only (not to be instantiated direcly) a protected constructor can be provided prevent direct instantiation.

 SimplifyConditional: No need to check for null before an instanceof; the instanceof keyword returns false when given a null argument.

 CompareObjectsWithEquals: Use equals() to compare object references; avoid comparing them with ==.

 PositionLiteralsFirstInComparisons: Position literals first in String comparisons - that way if the String is null you won't get a NullPointerException, it'll just return false.

 UnnecessaryLocalBeforeReturn: Avoid unnecessarily creating local variables

 NonThreadSafeSingleton: Non-thread safe singletons can result in bad state changes. Eliminate static singletons if possible by instantiating the object directly. Static singletons are usually not needed as only a single instance exists anyway. Other possible fixes are to synchronize the entire method or to use an initialize-on-demand holder class (do not use the double-check idiom). See Effective Java, item 48.

 UncommentedEmptyMethod: Uncommented Empty Method finds instances where a method does not contain statements, but there is no comment. By explicitly commenting empty methods it is easier to distinguish between intentional (commented) and unintentional empty methods.

 UncommentedEmptyConstructor: Uncommented Empty Constructor finds instances where a constructor does not contain statements, but there is no comment. By explicitly commenting empty constructors it is easier to distinguish between intentional (commented) and unintentional empty constructors.

 AvoidConstantsInterface: An interface should be used only to model a behaviour of a class: using an interface as a container of constants is a poor usage pattern.

 UnsynchronizedStaticDateFormatter: SimpleDateFormat is not synchronized. Sun recomends separate format instances for each thread. If multiple threads must access a static formatter, the formatter must be synchronized either on method or block level.

 PreserveStackTrace: Throwing a new exception from a catch block without passing the original exception into the new exception will cause the true stack trace to be lost, and can make it difficult to debug effectively.

 UseCollectionIsEmpty: The isEmpty() method on java.util.Collection is provided to see if a collection has any elements. Comparing the value of size() to 0 merely duplicates existing behavior.

 ClassWithOnlyPrivateConstructorsShouldBeFinal: A class with only private constructors should be final, unless the private constructor is called by a inner class.

 EmptyMethodInAbstractClassShouldBeAbstract: An empty method in an abstract class should be abstract instead, as developer may rely on this empty implementation rather than code the appropriate one.

 SingularField: This field is used in only one method and the first usage is assigning a value to the field. This probably means that the field can be changed to a local variable.

 ReturnEmptyArrayRatherThanNull: For any method that returns an array, it's a better behavior to return an empty array rather than a null reference.

 AbstractClassWithoutAnyMethod: If the abstract class does not provides any methods, it may be just a data container that is not to be instantiated. In this case, it's probably better to use a private or a protected constructor in order to prevent instantiation than make the class misleadingly abstract.

 TooFewBranchesForASwitchStatement: Swith are designed complex branches, and allow branches to share treatement. Using a switch for only a few branches is ill advised, as switches are not as easy to understand as if. In this case, it's most likely is a good idea to use a if statement instead, at least to increase code readability.

Finalizer Rules

 EmptyFinalizer: If the finalize() method is empty, then it does not need to exist.

 FinalizeOnlyCallsSuperFinalize: If the finalize() is implemented, it should do something besides just calling super.finalize().

 FinalizeOverloaded: Methods named finalize() should not have parameters. It is confusing and probably a bug to overload finalize(). It will not be called by the VM.

 FinalizeDoesNotCallSuperFinalize: If the finalize() is implemented, its last action should be to call super.finalize.

 FinalizeShouldBeProtected: If you override finalize(), make it protected. If you make it public, other classes may call it.

 AvoidCallingFinalize: Object.finalize() is called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

Import Statement Rules

 DuplicateImports: Avoid duplicate import statements.

 DontImportJavaLang: Avoid importing anything from the package 'java.lang'. These classes are automatically imported (JLS 7.5.3).

 UnusedImports: Avoid unused import statements.

 ImportFromSamePackage: No need to import a type that lives in the same package.

 TooManyStaticImports: If you overuse the static import feature, it can make your program unreadable and unmaintainable, polluting its namespace with all the static members you import. Readers of your code (including you, a few months after you wrote it) will not know which class a static member comes from (Sun 1.5 Language Guide).

J2EE Rules

 UseProperClassLoader: In J2EE getClassLoader() might not work as expected. Use Thread.currentThread().getContextClassLoader() instead.

 MDBAndSessionBeanNamingConvention: The EJB Specification state that any MessageDrivenBean or SessionBean should be suffixed by Bean.

 RemoteSessionInterfaceNamingConvention: Remote Home interface of a Session EJB should be suffixed by 'Home'.

 LocalInterfaceSessionNamingConvention: The Local Interface of a Session EJB should be suffixed by 'Local'.

 LocalHomeNamingConvention: The Local Home interface of a Session EJB should be suffixed by 'LocalHome'.

 RemoteInterfaceNamingConvention: Remote Interface of a Session EJB should NOT be suffixed.

 DoNotCallSystemExit: Web applications should not call System.exit(), since only the web container or the application server should stop the JVM.

 StaticEJBFieldShouldBeFinal: According to the J2EE specification (p.494), an EJB should not have any static fields with write access. However, static read only fields are allowed. This ensures proper behavior especially when instances are distributed by the container on several JREs.

 DoNotUseThreads: The J2EE specification explicitly forbid use of threads.

JavaBean Rules

 BeanMembersShouldSerialize: If a class is a bean, or is referenced by a bean directly or indirectly it needs to be serializable. Member variables need to be marked as transient, static, or have accessor methods in the class. Marking variables as transient is the safest and easiest modification. Accessor methods should follow the Java naming conventions, i.e.if you have a variable foo, you should provide getFoo and setFoo methods.

 MissingSerialVersionUID: Classes that are serializable should provide a serialVersionUID field.

JUnit Rules

 JUnitStaticSuite: The suite() method in a JUnit test needs to be both public and static.

 JUnitSpelling: Some JUnit framework methods are easy to misspell.

 JUnitAssertionsShouldIncludeMessage: JUnit assertions should include a message - i.e., use the three argument version of assertEquals(), not the two argument version.

 JUnitTestsShouldIncludeAssert: JUnit tests should include at least one assertion. This makes the tests more robust, and using assert with messages provide the developer a clearer idea of what the test does.

 TestClassWithoutTestCases: Test classes end with the suffix Test. Having a non-test class with that name is not a good practice, since most people will assume it is a test case. Test classes have test methods named testXXX.

 UnnecessaryBooleanAssertion: A JUnit test assertion with a boolean literal is unnecessary since it always will eval to the same thing. Consider using flow control (in case of assertTrue(false) or similar) or simply removing statements like assertTrue(true) and assertFalse(false). If you just want a test to halt, use the fail method.

 UseAssertEqualsInsteadOfAssertTrue: This rule detects JUnit assertions in object equality. These assertions should be made by more specific methods, like assertEquals.

 UseAssertSameInsteadOfAssertTrue: This rule detects JUnit assertions in object references equality. These assertions should be made by more specific methods, like assertSame, assertNotSame.

 UseAssertNullInsteadOfAssertTrue: This rule detects JUnit assertions in object references equality. These assertions should be made by more specific methods, like assertNull, assertNotNull.

 SimplifyBooleanAssertion: Avoid negation in an assertTrue or assertFalse test. For example, rephrase: assertTrue(!expr); as: assertFalse(expr);

Jakarta Commons Logging Rules

 UseCorrectExceptionLogging: To make sure the full stacktrace is printed out, use the logging statement with 2 arguments: a String and a Throwable.

 ProperLogger: A logger should normally be defined private static final and have the correct class. Private final Log log; is also allowed for rare cases where loggers need to be passed around, with the restriction that the logger needs to be passed into the constructor.

Java Logging Rules

 MoreThanOneLogger: Normally only one logger is used in each class.

 LoggerIsNotStaticFinal: In most cases, the Logger can be declared static and final.

 SystemPrintln: System.(out|err).print is used, consider using a logger.

 AvoidPrintStackTrace: Avoid printStackTrace(); use a logger call instead.

Migration Rules

 ReplaceVectorWithList: Consider replacing Vector usages with the newer java.util.ArrayList if expensive threadsafe operation is not required.

 ReplaceHashtableWithMap: Consider replacing this Hashtable with the newer java.util.Map

 ReplaceEnumerationWithIterator: Consider replacing this Enumeration with the newer java.util.Iterator

 AvoidEnumAsIdentifier: Finds all places where 'enum' is used as an identifier.

 AvoidAssertAsIdentifier: Finds all places where 'assert' is used as an identifier.

 IntegerInstantiation: In JDK 1.5, calling new Integer() causes memory allocation. Integer.valueOf() is more memory friendly.

 ByteInstantiation: In JDK 1.5, calling new Byte() causes memory allocation. Byte.valueOf() is more memory friendly.

 ShortInstantiation: In JDK 1.5, calling new Short() causes memory allocation. Short.valueOf() is more memory friendly.

 LongInstantiation: In JDK 1.5, calling new Long() causes memory allocation. Long.valueOf() is more memory friendly.

 JUnit4TestShouldUseBeforeAnnotation: In JUnit 3, the setUp method was used to set up all data entities required in running tests. JUnit 4 skips the setUp method and executes all methods annotated with @Before before all tests

 JUnit4TestShouldUseAfterAnnotation: In JUnit 3, the tearDown method was used to clean up all data entities required in running tests. JUnit 4 skips the tearDown method and executes all methods annotated with @After after running each test

 JUnit4TestShouldUseTestAnnotation: In JUnit 3, the framework executed all methods which started with the word test as a unit test. In JUnit 4, only methods annotated with the @Test annotation are executed.

 JUnit4SuitesShouldUseSuiteAnnotation: In JUnit 3, test suites are indicated by the suite() method. In JUnit 4, suites are indicated through the @RunWith(Suite.class) annotation.

 JUnitUseExpected:

Migration13

 :

 :

 :

Migration14

 :

Migration15

 :

 :

 :

 :

 :

MigratingToJava4

 :

 :

 :

 :

 :

Naming Rules

 ShortVariable: Detects when a field, local, or parameter has a very short name.

 LongVariable: Detects when a field, formal or local variable is declared with a long name.

 ShortMethodName: Detects when very short method names are used.

 VariableNamingConventions: A variable naming conventions rule - customize this to your liking. Currently, it checks for final variables that should be fully capitalized and non-final variables that should not include underscores.

 MethodNamingConventions: Method names should always begin with a lower case character, and should not contain underscores.

 ClassNamingConventions: Class names should always begin with an upper case character.

 AbstractNaming: Abstract classes should be named 'AbstractXXX'.

 AvoidDollarSigns: Avoid using dollar signs in variable/method/class/interface names.

 MethodWithSameNameAsEnclosingClass: Non-constructor methods should not have the same name as the enclosing class.

 SuspiciousHashcodeMethodName: The method name and return type are suspiciously close to hashCode(), which may mean you are intending to override the hashCode() method.

 SuspiciousConstantFieldName: A field name is all in uppercase characters, which in Sun's Java naming conventions indicate a constant. However, the field is not final.

 SuspiciousEqualsMethodName: The method name and parameter number are suspiciously close to equals(Object), which may mean you are intending to override the equals(Object) method.

 AvoidFieldNameMatchingTypeName: It is somewhat confusing to have a field name matching the declaring class name. This probably means that type and or field names could be more precise.

 AvoidFieldNameMatchingMethodName: It is somewhat confusing to have a field name with the same name as a method. While this is totally legal, having information (field) and actions (method) is not clear naming.

 NoPackage: Detects when a class or interface does not have a package definition.

 PackageCase: Detects when a package definition contains upper case characters.

 MisleadingVariableName: Detects when a non-field has a name starting with 'm\_'. This usually indicates a field and thus is confusing.

 BooleanGetMethodName: Looks for methods named 'getX()' with 'boolean' as the return type. The convention is to name these methods 'isX()'.

Optimization Rules

 LocalVariableCouldBeFinal: A local variable assigned only once can be declared final.

 MethodArgumentCouldBeFinal: A method argument that is never assigned can be declared final.

 AvoidInstantiatingObjectsInLoops: Detects when a new object is created inside a loop

 UseArrayListInsteadOfVector: ArrayList is a much better Collection implementation than Vector.

 SimplifyStartsWith: Since it passes in a literal of length 1, this call to String.startsWith can be rewritten using String.charAt(0) to save some time.

 UseStringBufferForStringAppends: Finds usages of += for appending strings.

 UseArraysAsList: The java.util.Arrays class has a "asList" method that should be used when you want to create a new List from an array of objects. It is faster than executing a loop to copy all the elements of the array one by one

 AvoidArrayLoops: Instead of copying data between two arrays, use System.arraycopy method

 UnnecessaryWrapperObjectCreation: Parsing method should be called directy instead.

 AddEmptyString: Finds empty string literals which are being added. This is an inefficient way to convert any type to a String.

Strict Exception Rules

 AvoidCatchingThrowable: This is dangerous because it casts too wide a net; it can catch things like OutOfMemoryError.

 SignatureDeclareThrowsException: It is unclear which exceptions that can be thrown from the methods. It might be difficult to document and understand the vague interfaces. Use either a class derived from RuntimeException or a checked exception.

 ExceptionAsFlowControl: Using Exceptions as flow control leads to GOTOish code and obscures true exceptions when debugging.

 AvoidCatchingNPE: Code should never throw NPE under normal circumstances. A catch block may hide the original error, causing other more subtle errors in its wake.

 AvoidThrowingRawExceptionTypes: Avoid throwing certain exception types. Rather than throw a raw RuntimeException, Throwable, Exception, or Error, use a subclassed exception or error instead.

 AvoidThrowingNullPointerException: Avoid throwing a NullPointerException - it's confusing because most people will assume that the virtual machine threw it. Consider using an IllegalArgumentException instead; this will be clearly seen as a programmer-initiated exception.

 AvoidRethrowingException: Catch blocks that merely rethrow a caught exception only add to code size and runtime complexity.

 DoNotExtendJavaLangError: Errors are system exceptions. Do not extend them.

 DoNotThrowExceptionInFinally: Throwing exception in a finally block is confusing. It may mask exception or a defect of the code, it also render code cleanup uninstable. Note: This is a PMD implementation of the Lint4j rule "A throw in a finally block"

 AvoidThrowingNewInstanceOfSameException: Catch blocks that merely rethrow a caught exception wrapped inside a new instance of the same type only add to code size and runtime complexity.

String and StringBuffer Rules

 AvoidDuplicateLiterals: Code containing duplicate String literals can usually be improved by declaring the String as a constant field.

 StringInstantiation: Avoid instantiating String objects; this is usually unnecessary.

 StringToString: Avoid calling toString() on String objects; this is unnecessary.

 InefficientStringBuffering: Avoid concatenating non literals in a StringBuffer constructor or append().

 UnnecessaryCaseChange: Using equalsIgnoreCase() is faster than using toUpperCase/toLowerCase().equals()

 UseStringBufferLength: Use StringBuffer.length() to determine StringBuffer length rather than using StringBuffer.toString().equals("") or StringBuffer.toString().length() ==.

 AppendCharacterWithChar: Avoid concatenating characters as strings in StringBuffer.append.

 ConsecutiveLiteralAppends: Consecutively calling StringBuffer.append with String literals

 UseIndexOfChar: Use String.indexOf(char) when checking for the index of a single character; it executes faster.

 InefficientEmptyStringCheck: String.trim().length() is an inefficient way to check if a String is really empty, as it creates a new String object just to check its size. Consider creating a static function that loops through a string, checking Character.isWhitespace() on each character and returning false if a non-whitespace character is found.

 InsufficientStringBufferDeclaration: Failing to pre-size a StringBuffer properly could cause it to re-size many times during runtime. This rule checks the characters that are actually passed into StringBuffer.append(), but represents a best guess "worst case" scenario. An empty StringBuffer constructor initializes the object to 16 characters. This default is assumed if the length of the constructor can not be determined.

 UselessStringValueOf: No need to call String.valueOf to append to a string; just use the valueOf() argument directly.

 StringBufferInstantiationWithChar: StringBuffer sb = new StringBuffer('c'); The char will be converted into int to intialize StringBuffer size.

 UseEqualsToCompareStrings: Using '==' or '!=' to compare strings only works if intern version is used on both sides

 AvoidStringBufferField: StringBuffers can grow quite a lot, and so may become a source of memory leak (if the owning class has a long life time).

Security Code Guidelines

 MethodReturnsInternalArray: Exposing internal arrays directly allows the user to modify some code that could be critical. It is safer to return a copy of the array.

 ArrayIsStoredDirectly: Constructors and methods receiving arrays should clone objects and store the copy. This prevents that future changes from the user affect the internal functionality.

Type Resolution Rules

 LooseCoupling: Avoid using implementation types (i.e., HashSet); use the interface (i.e, Set) instead

 CloneMethodMustImplementCloneable: The method clone() should only be implemented if the class implements the Cloneable interface with the exception of a final method that only throws CloneNotSupportedException. This version uses PMD's type resolution facilities, and can detect if the class implements or extends a Cloneable class

 UnusedImports: Avoid unused import statements. This rule will find unused on demand imports, i.e. import com.foo.\*.

 SignatureDeclareThrowsException: It is unclear which exceptions that can be thrown from the methods. It might be difficult to document and understand the vague interfaces. Use either a class derived from RuntimeException or a checked exception. Junit classes are excluded.

Unused Code Rules

 UnusedPrivateField: Detects when a private field is declared and/or assigned a value, but not used.

 UnusedLocalVariable: Detects when a local variable is declared and/or assigned, but not used.

 UnusedPrivateMethod: Unused Private Method detects when a private method is declared but is unused.

 UnusedFormalParameter: Avoid passing parameters to methods or constructors and then not using those parameters.