PARENT PROJECT

PMD

RULE SETS

java

- Android
- Basic
- Braces
- Clone Implementation
- Code Size
- Comments
- Controversial
- Coupling
- Design
- Empty Code
- Finalizer
- Import Statements

Current Rulesets

List of rulesets and rules contained in each ruleset.

- Android: These rules deal with the Android SDK, mostly related to best practices.To get better results, make sure that the auxclasspath is defined for type resolution to work.
- Basic: The Basic ruleset contains a collection of good practices which should be followed.
- Braces: The Braces ruleset contains rules regarding the use and placement of braces.
- Clone Implementation: The Clone Implementation ruleset contains a collection of rules that find questionable usages of the clone() method.
- Code Size: The Code Size ruleset contains rules that find problems related to code size or complexity.
- Comments: Rules intended to catch errors related to code comments
- Controversial: The Controversial ruleset contains rules that, for whatever reason, are considered controversial.They are held here to allow people to include them as they see fit within their custom rulesets.
- Coupling: Rules which find instances of high or inappropriate coupling between objects and packages.
- Design: The Design ruleset contains rules that flag suboptimal code implementations. Alternate approachesare suggested.
- Empty Code: The Empty Code ruleset contains rules that find empty statements of any kind (empty method,empty block statement, empty try or catch block,...).
- Finalizer: These rules deal with different problems that can occur with finalizers.
- Import Statements: These rules deal with different problems that can occur with import statements.
- J2EE: Rules specific to the use of J2EE implementations.

- Jakarta Commons Logging: The Jakarta Commons Logging ruleset contains a collection of rules that find questionable usages of that framework.
- JavaBeans: The JavaBeans Ruleset catches instances of bean rules not being followed.
- Java Logging: The Java Logging ruleset contains a collection of rules that find questionable usages of the logger.
- JUnit: These rules deal with different problems that can occur with JUnit tests.
- Migration: Contains rules about migrating from one JDK version to another. Don't use these rules directly,rather, use a wrapper ruleset such as migrating_to_13.xml.
- Naming: The Naming Ruleset contains rules regarding preferred usage of names and identifiers.
- Optimization: These rules deal with different optimizations that generally apply to best practices.
- Security Code Guidelines: These rules check the security guidelines from Sun, published at http://java.sun.com/security/seccodeguide.html#gcg
- Strict Exceptions: These rules provide some strict guidelines about throwing and catching exceptions.
- String and StringBuffer: These rules deal with different issues that can arise with manipulation of the String, StringBuffer, or StringBuilder instances.
- Type Resolution: These are rules which resolve java Class files for comparison, as opposed to a String
- Unnecessary: The Unnecessary Ruleset contains a collection of rules for unnecessary code.
- Unused Code: The Unused Code ruleset contains rules that find unused or ineffective code.

Android (java)

- CallSuperFirst: Super should be called at the start of the method
- CallSuperLast: Super should be called at the end of the method
- DoNotHardCodeSDCard: Use Environment.getExternalStorageDirectory() instead of "/sdcard"

Basic (java)

- JumbledIncrementer: Avoid jumbled loop incrementers - its usually a mistake, and is confusing even if intentional.
- ForLoopShouldBeWhileLoop: Some for loops can be simplified to while loops, this makes them more concise.
- 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 thereference is intended to point to.For more details refer to: http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-double.html

- ReturnFromFinallyBlock: Avoid returning from a finally block, this can discard exceptions.
- UnconditionalIfStatement: Do not use "if" statements whose conditionals are always true or always false.
- BooleanInstantiation: Avoid instantiating Boolean objects; you can reference Boolean.TRUE, Boolean.FALSE, or call Boolean.valueOf() instead.
- CollapsibleIfStatements: Sometimes two consecutive 'if' statements can be consolidated by separating their conditions with a boolean short-circuit operator.
- ClassCastExceptionWithToArray: When deriving an array of a specific class from your Collection, one should provide an array ofthe same class as the parameter of the toArray() method. Doing otherwise you will will resultin a ClassCastException.
- AvoidDecimalLiteralsInBigDecimalConstructor: One might assume that the result of "new BigDecimal(0.1)" is exactly equal to 0.1, but it is actuallyequal to .1000000000000000055511151231257827021181583404541015625.This is because 0.1 cannot be represented exactly as a double (or as a binary fraction of any finitelength). Thus, the long value that is being passed in to the constructor is not exactly equal to 0.1,appearances notwithstanding.The (String) constructor, on the other hand, is perfectly predictable: 'new BigDecimal("0.1")' isexactly equal to 0.1, as one would expect. Therefore, it is generally recommended that the(String) constructor be used in preference to this one.
- MisplacedNullCheck: The null check here is misplaced. If the variable is null a NullPointerException will be thrown.Either the check is useless (the variable will never be "null") or it is incorrect.
- AvoidThreadGroup: Avoid using java.lang.ThreadGroup; although it is intended to be used in a threaded environmentit 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) andfor Java 1.5 onwards, BigInteger.TEN and BigDecimal (BigDecimal.ZERO, BigDecimal.ONE, BigDecimal.TEN)
- AvoidUsingOctalValues: Integer literals should not start with zero since this denotes that the rest of literal will beinterpreted as an octal value.
- AvoidUsingHardCodedIP: Application with hard-coded IP addresses can become impossible to deploy in

some cases.Externalizing IP adresses is preferable.
- CheckResultSet: Always check the return values of navigation methods (next, previous, first, last) of a ResultSet.If the value return is 'false', it should be handled properly.
- AvoidMultipleUnaryOperators: The use of multiple unary operators may be problematic, and/or confusing.Ensure that the intended usage is not a bug, or consider simplifying the expression.
- ExtendsObject: No need to explicitly extend Object.
- CheckSkipResult: The skip() method may skip a smaller number of bytes than requested. Check the returned value to find out if it was the case or not.
- AvoidBranchingStatementAsLastInLoop: Using a branching statement as the last part of a loop may be a bug, and/or is confusing.Ensure that the usage is not a bug, or consider using another approach.
- DontCallThreadRun: Explicitly calling Thread.run() method will execute in the caller's thread of control. Instead, call Thread.start() for the intended behavior.
- DontUseFloatTypeForLoopIndices: Don't use floating point for loop indices. If you must use floating point, use doubleunless you're certain that float provides enough precision and you have a compellingperformance need (space or time).

Braces (java)

- IfStmtsMustUseBraces: Avoid using if statements without using braces to surround the code block. If the code formatting or indentation is lost then it becomes difficult to separate the code beingcontrolled from the rest.
- WhileLoopsMustUseBraces: Avoid using 'while' statements without using braces to surround the code block. If the code formatting or indentation is lost then it becomes difficult to separate the code beingcontrolled from the rest.
- IfElseStmtsMustUseBraces: Avoid using if..else statements without using surrounding braces. If the code formatting or indentation is lost then it becomes difficult to separate the code being controlled from the rest.
- ForLoopsMustUseBraces: Avoid using 'for' statements without using curly braces. If the code formatting or indentation is lost then it becomes difficult to separate the code being controlled from the rest.

Clone Implementation (java)

- 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 (java)

- 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 and increase readability.
- ExcessiveMethodLength: When methods are excessively long this usually indicates that the method is doing more than itsname/signature might suggest. They also become challenging for others to digest since excessive scrolling causes readers to lose focus.Try to reduce the method length by creating helper methods and removing any copy/pasted code.
- ExcessiveParameterList: Methods with numerous parameters are a challenge to maintain, especially if most of them share thesame datatype. These situations usually denote the need for new objects to wrap the numerous parameters.
- ExcessiveClassLength: Excessive class file lengths are usually indications that the class may be burdened with excessive responsibilities that could be provided by external classes or functions. In breaking these methodsapart the code becomes more managable and ripe for reuse.
- CyclomaticComplexity: Complexity directly affects maintenance costs is determined by the number of decision points in a method plus one for the method entry. The decision points include 'if', 'while', 'for', and 'case labels' calls. Generally, numbers ranging from 1-4 denote low complexity, 5-7 denote moderate complexity, 8-10 denotehigh complexity, and 11+ is very high complexity.
- StdCyclomaticComplexity: Complexity directly affects maintenance costs is determined by the number of decision points in a method plus one for the method entry. The decision points include 'if', 'while', 'for', and 'case labels' calls. Generally, numbers ranging from 1-4 denote low complexity, 5-7 denote moderate complexity, 8-10 denotehigh complexity, and 11+ is very high complexity.
- ModifiedCyclomaticComplexity: Complexity directly affects maintenance costs is determined by the number of decision points in a method plus one for the method entry. The decision points include 'if', 'while', 'for', and 'case labels' calls. Generally, numbers ranging from 1-4 denote low complexity, 5-7 denote moderate complexity, 8-10 denotehigh complexity, and 11+ is very high complexity. Modified complexity treats switch statements as a singledecision point.
- ExcessivePublicCount: Classes with large numbers of public methods and attributes require disproportionate testing effortssince combinational side effects grow rapidly and increase risk. Refactoring

these classes intosmaller ones not only increases testability and reliability but also allows new variations to bedeveloped easily.

- TooManyFields: Classes that have too many fields can become unwieldy and could be redesigned to have fewer fields,possibly through grouping related fields in new objects. For example, a class with individual city/state/zip fields could park them within a single Address field.
- NcssMethodCount: This rule uses the NCSS (Non-Commenting Source Statements) algorithm to determine the number of linesof 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 linesof 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 linesof 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 tohave more fine grained objects.

Comments (java)

- CommentRequired: Denotes whether comments are required (or unwanted) for specific language elements.
- CommentSize: Determines whether the dimensions of non-header comments found are within the specified limits.
- CommentContent: A rule for the politically correct... we don't want to offend anyone.

Controversial (java)

- UnnecessaryConstructor: This rule detects when a constructor is not necessary; i.e., when there is only one constructor,its public, has an empty body, and takes no arguments.
- NullAssignment: Assigning a "null" to a variable (outside of its declaration) is usually bad form. Sometimes, this typeof assignment is an indication that the programmer doesn't completely understand what is going on in the code.NOTE: This sort of assignment may used in some cases to dereference objects and encourage garbage collection.
- OnlyOneReturn: A method should have only one exit point, and that should be the last statement in the

method.

- 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 octalescape sequence inside a literal String shall consist of a backslashfollowed by: OctalDigit | OctalDigit OctalDigit | ZeroToThree OctalDigit OctalDigitAny octal escape sequence followed by non-octal digits can be confusing,e.g. "\038" is interpreted as the octal escape sequence "\03" followed bythe literal character "8".
- CallSuperInConstructor: It is a good practice to call super() in a constructor. If super() is not called butanother 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 function calls.
- 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 anyarithmetic capabilities for the short type: the JVM must convert the short into an int, do the proper calculationand convert the int back to a short. Thus any storage gains found through use of the 'short' type may be offset byadverse impacts on performance.
- AvoidUsingVolatile: Use of the keyword 'volatile' is generally used to fine tune a Java application, and therefore, requiresa 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: Unnecessary reliance on Java Native Interface (JNI) calls directly reduces application portabilityand increases the maintenance burden.

- AvoidAccessibilityAlteration: Methods such as getDeclaredConstructors(), getDeclaredConstructor(Class[]) and setAccessible(),as the interface PrivilegedAction, allows for the runtime alteration of variable, class, ormethod visibility, even if they are private. This violates the principle of encapsulation.
- DoNotCallGarbageCollectionExplicitly: Calls to System.gc(), Runtime.getRuntime().gc(), and System.runFinalization() are not advised. Code should have thesame 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 memoryleaks develop within an application, it should be dealt with JVM options rather than within the code itself.
- OneDeclarationPerLine: Java allows the use of several variables declaration of the same type on one line. However, itcan lead to quite messy code. This rule looks for several declarations on the same line.
- AvoidPrefixingMethodParameters: Prefixing parameters by 'in' or 'out' pollutes the name of the parameters and reduces code readability.To indicate whether or not a parameter will be modify in a method, its better to document methodbehavior with Javadoc.
- AvoidLiteralsInIfCondition: Avoid using hard-coded literals in conditional statements. By declaring them as static variablesor private members with descriptive names maintainability is enhanced. By default, the literals "-1" and "0" are ignored.More exceptions can be defined with the property "ignoreMagicNumbers".
- UseObjectForClearerAPI: When you write a public method, you should be thinking in terms of an API. If your method is public, it means other classwill use it, therefore, you want (or need) to offer a comprehensive and evolutive API. If you pass a lot of informationas a simple series of Strings, you may think of using an Object to represent all those information. You'll get a simplierAPI (such as doWork(Workload workload), rather than a tedious series of Strings) and more importantly, if you need at somepoint to pass extra data, you'll be able to do so by simply modifying or extending Workload without any modification toyour API.
- UseConcurrentHashMap: Since Java5 brought a new implementation of the Map designed for multi-threaded access, you canperform efficient map reads without blocking other threads.

Coupling (java)

- CouplingBetweenObjects: This rule counts the number of unique attributes, local variables, and return types within an object. A number higher than the 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. This rule counts the number of unique imports and reports a violation if the count is above the user-specified threshold.

- LooseCoupling: The use of implementation types as object references limits your ability to use alternateimplementations in the future as requirements change. Whenever available, referencing objects by their interface types provides much more flexibility.
- LoosePackageCoupling: Avoid using classes from the configured package hierarchy outside of the package hierarchy, except when using one of the configured allowed classes.
- LawOfDemeter: The Law of Demeter is a simple rule, that says "only talk to friends". It helps to reduce coupling between classes or objects. See also the references:Andrew Hunt, David Thomas, and Ward Cunningham. The Pragmatic Programmer. From Journeyman to Master. Addison-Wesley Longman, Amsterdam, October 1999.;K.J. Lieberherr and I.M. Holland. Assuring good style for object-oriented programs. Software, IEEE, 6(5):38???48, 1989.;http://www.ccs.neu.edu/home/lieber/LoD.html;http://en.wikipedia.org/wiki/Law_of_Demeter

Design (java)

- UseUtilityClass: For classes that only have static methods, consider making them utility classes.Note that this doesn't apply to abstract classes, since their subclasses maywell include non-static methods. Also, if you want this class to be a utility class,remember to add a private constructor to prevent instantiation.(Note, that this use was known before PMD 5.1.0 as UseSingleton).
- SimplifyBooleanReturns: Avoid unnecessary if-then-else statements when returning a boolean. The result ofthe conditional test can be returned instead.
- SimplifyBooleanExpressions: Avoid unnecessary comparisons in boolean expressions, they serve no purpose and impacts readability.
- SwitchStmtsShouldHaveDefault: All switch statements should include a default option to catch any unspecified values.
- AvoidDeeplyNestedIfStmts: Avoid creating deeply nested if-then statements since they are harder to read and error-prone to maintain.
- AvoidReassigningParameters: Reassigning values to incoming parameters is not recommended. Use temporary local variables instead.
- SwitchDensity: A high ratio of statements to labels in a switch statement implies that the switch statement is overloaded. 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 debug.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-privatization 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 non-static 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: Calls to a collection's toArray() method should specify target arrays sized to match the size of thecollection. Initial arrays that are too small are discarded in favour of new ones that have to be createdthat are the proper size.
- BadComparison: Avoid equality comparisons with Double.NaN. Due to the implicit lack of representationprecision when comparing floating point numbers these are likely to cause logic errors.
- EqualsNull: Tests for null should not use the equals() method. The '==' operator should be used instead.
- ConfusingTernary: Avoid negation within an "if" expression with an "else" clause. 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, suchas "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 have no effect.
- SimpleDateFormatNeedsLocale: Be sure to specify a Locale when creating SimpleDateFormat instances to

ensure that locale-appropriateformatting is used.

- ImmutableField: Identifies private fields whose values never change once they are initialized either in the declaration of the field or by a constructor. This helps in converting existing classes to becoming immutable ones.
- UseLocaleWithCaseConversions: When doing String.toLowerCase()/toUpperCase() conversions, use Locales to avoids problems with languages thathave unusual conventions, 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 cause problems when new code is added to the method. Block-level synchronization helps to ensure that only the code that needs synchronization gets it.
- MissingBreakInSwitch: Switch statements without break or return statements for each case optionmay indicate problematic behaviour. Empty cases are ignored as these indicate an intentional fall-through.
- UseNotifyAllInsteadOfNotify: Thread.notify() awakens a thread monitoring the object. If more than one thread is monitoring, then onlyone is chosen. The thread chosen is arbitrary; thus its 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 suggestsan incomplete implementation, which is to be completed by subclasses implementing theabstract methods. If the class is intended to be used as a base class only (not to be instantiateddirectly) 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 comparisons, if the second argument is null then NullPointerExceptions can be avoided, they will just return false.
- PositionLiteralsFirstInCaseInsensitiveComparisons: Position literals first in comparisons, if the second argument is null then NullPointerExceptions can be avoided, they will just return false.
- UnnecessaryLocalBeforeReturn: Avoid the creation of unnecessary local variables
- NonThreadSafeSingleton: Non-thread safe singletons can result in bad state changes. Eliminatestatic

singletons if possible by instantiating the object directly. Staticsingletons are usually not needed as only a single instance exists anyway.Other possible fixes are to synchronize the entire method or to use aninitialize-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 containstatements, but there is no comment. By explicitly commenting empty methodsit is easier to distinguish between intentional (commented) and unintentionalempty methods.
- UncommentedEmptyConstructor: Uncommented Empty Constructor finds instances where a constructor does notcontain statements, but there is no comment. By explicitly commenting emptyconstructors it is easier to distinguish between intentional (commented)and unintentional empty constructors.
- AvoidConstantsInterface: An interface should be used only to characterize the external behaviour of animplementing class: using an interface as a container of constants is a poor usage pattern and not recommended.
- UnsynchronizedStaticDateFormatter: SimpleDateFormat instances are not synchronized. Sun recommends using separate format instancesfor 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 thenew exception will cause the original stack trace to be lost making it difficult to debug effectively.
- UseCollectionIsEmpty: The isEmpty() method on java.util.Collection is provided to determine if a collection has any elements.Comparing the value of size() to 0 does not convey intent as well as the isEmpty() method.
- ClassWithOnlyPrivateConstructorsShouldBeFinal: A class with only private constructors should be final, unless the private constructor is invoked by a inner class.
- EmptyMethodInAbstractClassShouldBeAbstract: Empty methods in an abstract class should be tagged as abstract. This helps to remove their inapproprate usage by developers who should be implementing their own versions in the concrete subclasses.
- SingularField: Fields whose scopes are limited to just single methods do not rely on the containingobject to provide them to other methods. They may be better implemented as local variableswithin those methods.
- ReturnEmptyArrayRatherThanNull: For any method that returns an array, it is a better to return an empty array rather than a null reference. This removes the need for null checking all results and avoids inadvertentNullPointerExceptions.
- AbstractClassWithoutAnyMethod: If an abstract class does not provides any methods, it may be acting as a simple data container that is not meant to be instantiated. In this case, it is probably better to use a private or protected constructor in order to prevent instantiation than make the class misleadingly abstract.
- TooFewBranchesForASwitchStatement: Switch statements are indended to be used to support complex

branching behaviour. Using a switch for only a few cases is ill-advised, since switches are not as easy to understand as if-then statements. In these cases use theif-then statement to increase code readability.

- LogicInversion: Use opposite operator instead of negating the whole expression with a logic complement operator.
- UseVarargs: Java 5 introduced the varargs parameter declaration for methods and constructors. This syntactic sugar provides flexibility for users of these methods and constructors, allowing them to avoid having to deal with the creation of an array.
- FieldDeclarationsShouldBeAtStartOfClass: Fields should be declared at the top of the class, before any method declarations, constructors, initializers or inner classes.
- GodClass: The God Class rule detects the God Class design flaw using metrics. God classes do too many things,are very big and overly complex. They should be split apart to be more object-oriented.The rule uses the detection strategy described in "Object-Oriented Metrics in Practice".The violations are reported against the entire class. See also the references:Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice:Using Software Metrics to Characterize, Evaluate, and Improve the Designof Object-Oriented Systems. Springer, Berlin, 1 edition, October 2006. Page 80.
- AvoidProtectedMethodInFinalClassNotExtending: Do not use protected methods in most final classes since they cannot be subclassed. This should only be allowed in final classes that extend other classes with protected methods (whosevisibility cannot be reduced). Clarify your intent by using private or package access modifiers instead.

Empty Code (java)

- 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 isa 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: Empty finally blocks serve no purpose and should be removed.
- EmptySwitchStatements: Empty switch statements serve no purpose and should be removed.
- EmptySynchronizedBlock: Empty synchronized blocks serve no purpose and should be removed.
- EmptyStatementNotInLoop: An empty statement (or a semicolon by itself) that is not used as the sole body of a 'for' or 'while' loop is probably a bug. It could also be a double semicolon, which has no purposeand

should be removed.
- EmptyInitializer: Empty initializers serve no purpose and should be removed.
- EmptyStatementBlock: Empty block statements serve no purpose and should be removed.
- EmptyStaticInitializer: An empty static initializer serve no purpose and should be removed.

Finalizer (java)

- EmptyFinalizer: Empty finalize methods serve no purpose and should be removed.
- 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 most likely an attempt tooverload Object.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: When overriding the finalize(), the new method should be set as protected. If made public, other classes may invoke it at inappropriate times.
- AvoidCallingFinalize: The method Object.finalize() is called by the garbage collector on an object when garbage collection determinesthat there are no more references to the object. It should not be invoked by application logic.

Import Statements (java)

- DuplicateImports: Duplicate or overlapping import statements should be avoided.
- DontImportJavaLang: Avoid importing anything from the package 'java.lang'. These classes are automatically imported (JLS 7.5.3).
- UnusedImports: Avoid the use of unused import statements to prevent unwanted dependencies.
- ImportFromSamePackage: There is 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).
- UnnecessaryFullyQualifiedName: Import statements allow the use of non-fully qualified names. The use of a fully qualified namewhich is covered by an import statement is redundant. Consider using the non-fully qualified name.

J2EE (java)

- UseProperClassLoader: In J2EE, the getClassLoader() method might not work as expected. Use Thread.currentThread().getContextClassLoader() instead.
- MDBAndSessionBeanNamingConvention: The EJB Specification states that any MessageDrivenBean or SessionBean should be suffixed by 'Bean'.
- RemoteSessionInterfaceNamingConvention: A Remote Home interface type 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 have a suffix.
- DoNotCallSystemExit: Web applications should not call System.exit(), since only the web container or theapplication server should stop the JVM. This rule also checks for the equivalent call Runtime.getRuntime().exit().
- StaticEJBFieldShouldBeFinal: According to the J2EE specification, an EJB should not have any static fieldswith write access. However, static read-only fields are allowed. This ensures properbehavior especially when instances are distributed by the container on several JREs.
- DoNotUseThreads: The J2EE specification explicitly forbids the use of threads.

JavaBeans (java)

- 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. for a variable named foo, getFoo() and setFoo() accessor methods should be provided.
- MissingSerialVersionUID: Serializable classes should provide a serialVersionUID field.

JUnit (java)

- 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 an informative 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 evaluate to the same thing.Consider using flow control (in case of assertTrue(false) or similar) or simply removingstatements like assertTrue(true) and assertFalse(false). If you just want a test to halt after findingan error, use the fail() method and provide an indication message of why it did.
- 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);
- JUnitTestContainsTooManyAsserts: JUnit tests should not contain too many asserts. Many asserts are indicative of a complex test, for which it is harder to verify correctness. Consider breaking the test scenario into multiple, shorter test scenarios. Customize the maximum number of assertions used by this Rule to suit your needs.
- UseAssertTrueInsteadOfAssertEquals: When asserting a value is the same as a boolean literal, use assertTrue/assertFalse, instead of assertEquals.

Jakarta Commons Logging (java)

- UseCorrectExceptionLogging: To make sure the full stacktrace is printed out, use the logging statement with two arguments: a String and a Throwable.
- ProperLogger: A logger should normally be defined private static final and be associated with 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.

- GuardDebugLogging: When log messages are composed by concatenating strings, the whole section should be guarded by a isDebugEnabled() check to avoid performance and memory issues.
- GuardLogStatement: Whenever using a log level, one should check if the loglevel is actually enabled, orotherwise skip the associate String creation and manipulation.

Java Logging (java)

- MoreThanOneLogger: Normally only one logger is used in each class.
- LoggerIsNotStaticFinal: In most cases, the Logger reference can be declared as static and final.
- SystemPrintln: References to System.(out|err).print are usually intended for debugging purposes and can remain inthe codebase even in production code. By using a logger one can enable/disable this behaviour atwill (and by priority) and avoid clogging the Standard out log.
- AvoidPrintStackTrace: Avoid printStackTrace(); use a logger call instead.
- GuardLogStatementJavaUtil: Whenever using a log level, one should check if the loglevel is actually enabled, orotherwise skip the associate String creation and manipulation.

Migration (java)

- ReplaceVectorWithList: Consider replacing Vector usages with the newer java.util.ArrayList if expensive thread-safe operations are not required.
- ReplaceHashtableWithMap: Consider replacing Hashtable usage with the newer java.util.Map if thread safety is not required.
- ReplaceEnumerationWithIterator: Consider replacing Enumeration usages with the newer java.util.Iterator
- AvoidEnumAsIdentifier: Use of the term 'enum' will conflict with newer versions of Java since it is a reserved word.
- AvoidAssertAsIdentifier: Use of the term 'assert' will conflict with newer versions of Java since it is a reserved word.
- IntegerInstantiation: Calling new Integer() causes memory allocation that can be avoided by the static Integer.valueOf().It makes use of an internal cache that recycles earlier instances making it more memory efficient.
- ByteInstantiation: Calling new Byte() causes memory allocation that can be avoided by the static Byte.valueOf().It makes use of an internal cache that recycles earlier instances making it more memory efficient.
- ShortInstantiation: Calling new Short() causes memory allocation that can be avoided by the static

Short.valueOf().It makes use of an internal cache that recycles earlier instances making it more memory efficient.

- LongInstantiation: Calling new Long() causes memory allocation that can be avoided by the static Long.valueOf().It makes use of an internal cache that recycles earlier instances making it more memory efficient.
- 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 indicatedthrough the @RunWith(Suite.class) annotation.
- JUnitUseExpected: In JUnit4, use the @Test(expected) annotation to denote tests that should throw exceptions.

Naming (java)

- ShortVariable: Fields, local variables, or parameter names that are very short are not helpful to the reader.
- LongVariable: Fields, formal arguments, or local variable names that are too long can make the code difficult to follow.
- ShortMethodName: Method names that are very short are not helpful to the reader.
- VariableNamingConventions: A variable naming conventions rule - customize this to your liking. Currently, itchecks for final variables that should be fully capitalized and non-final variablesthat 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 denote an intentionto override the hashCode() method.
- SuspiciousConstantFieldName: Field names using all uppercase characters - Sun's Java naming conventions indicating constants - shouldbe declared as final.
- SuspiciousEqualsMethodName: The method name and parameter number are suspiciously close to equals(Object), which can denote anintention 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 should be chosen more carefully.
- AvoidFieldNameMatchingMethodName: It can be confusing to have a field name with the same name as a method. While this is permitted, having information (field) and actions (method) is not clear naming. Developers versed in Smalltalk often prefer this approach as the methods denote accessor methods.
- NoPackage: Detects when a class or interface does not have a package definition.
- PackageCase: Detects when a package definition contains uppercase characters.
- MisleadingVariableName: Detects when a non-field has a name starting with 'm_'. This usually denotes a field and could be confusing.
- BooleanGetMethodName: Methods that return boolean results should be named as predicate statements to denote this.I.e, 'isReady()', 'hasValues()', 'canCommit()', 'willFail()', etc. Avoid the use of the 'get'prefix for these methods.
- ShortClassName: Short Classnames with fewer than e.g. five characters are not recommended.
- GenericsNaming: Names for references to generic values should be limited to a single uppercase letter.

Optimization (java)

- LocalVariableCouldBeFinal: A local variable assigned only once can be declared final.
- MethodArgumentCouldBeFinal: A method argument that is never re-assigned within the method can be declared final.
- AvoidInstantiatingObjectsInLoops: New objects created within loops should be checked to see if they can created outside them and reused.
- UseArrayListInsteadOfVector: ArrayList is a much better Collection implementation than Vector if thread-safe operation is not required.
- SimplifyStartsWith: Since it passes in a literal of length 1, calls to (string).startsWith can be rewritten using (string).charAt(0)at the expense of some readability.
- UseStringBufferForStringAppends: The use of the '+=' operator for appending strings causes the JVM to create and use an internal StringBuffer.If a non-trivial number of these concatenations are being used then

the explicit use of a StringBuilder or threadsafe StringBuffer is recommended to avoid this.

- UseArraysAsList: The java.util.Arrays class has a "asList" method that should be used when you want to create a new List froman array of objects. It is faster than executing a loop to copy all the elements of the array one by one.
- AvoidArrayLoops: Instead of manually copying data between two arrays, use the efficient System.arraycopy method instead.
- UnnecessaryWrapperObjectCreation: Most wrapper classes provide static conversion methods that avoid the need to create intermediate objectsjust to create the primitive forms. Using these avoids the cost of creating objects that also need to be garbage-collected later.
- AddEmptyString: The conversion of literals to strings by concatenating them with empty strings is inefficient.It is much better to use one of the type-specific toString() methods instead.
- RedundantFieldInitializer: Java will initialize fields with known default values so any explicit initialization of those same defaultsis redundant and results in a larger class file (approximately three additional bytecode instructions per field).
- PrematureDeclaration: Checks for variables that are defined before they might be used. A reference is deemed to be premature if it is created right before a block of code that doesn't use it that also has the ability to return or throw an exception.

Strict Exceptions (java)

- AvoidCatchingThrowable: Catching Throwable errors is not recommended since its scope is very broad. It includes runtime issues such as OutOfMemoryError that should be exposed and managed separately.
- SignatureDeclareThrowsException: Methods that declare the generic Exception as a possible throwable are not very helpful since theirfailure modes are unclear. Use a class derived from RuntimeException or a more specific checked exception.
- ExceptionAsFlowControl: Using Exceptions as form of flow control is not recommended as they obscure true exceptions when debugging.Either add the necessary validation or use an alternate control structure.
- AvoidCatchingNPE: Code should never throw NullPointerExceptions under normal circumstances. A catch block may hide the original error, causing other, more subtle problems later on.
- 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 NullPointerExceptions. These are confusing because most people will assume that thevirtual machine threw it. Consider using an IllegalArgumentException instead; this will beclearly 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 exceptions within a 'finally' block is confusing since they may mask other exceptions or code defects.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 tocode size and runtime complexity.
- AvoidCatchingGenericException: Avoid catching generic exceptions such as NullPointerException, RuntimeException, Exception in try-catch block
- AvoidLosingExceptionInformation: Statements in a catch block that invoke accessors on the exception without using the informationonly add to code size. Either remove the invocation, or use the return result.

String and StringBuffer (java)

- 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 since they are immutable and can be safely shared.
- StringToString: Avoid calling toString() on objects already known to be string instances; this is unnecessary.
- InefficientStringBuffering: Avoid concatenating non-literals in a StringBuffer constructor or append() since intermediate buffers willneed to be be created and destroyed by the JVM.
- 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/StringBuilder.append methods.
- ConsecutiveAppendsShouldReuse: Consecutively calls to StringBuffer/StringBuilder .append should reuse the target object. This can improve the performance.
- ConsecutiveLiteralAppends: Consecutively calling StringBuffer/StringBuilder.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 itcreates a new String object just to check its size. Consider creating a static function thatloops through a string, checking Character.isWhitespace() on each character and returningfalse if a non-whitespace character is found.
- InsufficientStringBufferDeclaration: Failing to pre-size a StringBuffer or StringBuilder properly could cause it to re-size many timesduring runtime. This rule attempts to determine the total number the characters that are actually passed into StringBuffer.append(), but represents a best guess "worst case" scenario. An emptyStringBuffer/StringBuilder constructor initializes the object to 16 characters. This defaultis 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: Individual character values provided as initialization arguments will be converted into integers.This can lead to internal buffer sizes that are larger than expected. Some examples:new StringBuffer() // 16new StringBuffer(6) // 6new StringBuffer("hello world") // 11 + 16 = 27new StringBuffer('A') // chr(A) = 65new StringBuffer("A") // 1 + 16 = 17 new StringBuilder() // 16new StringBuilder(6) // 6new StringBuilder("hello world") // 11 + 16 = 27new StringBuilder('C') // chr(C) = 67new StringBuilder("A") // 1 + 16 = 17
- UseEqualsToCompareStrings: Using '==' or '!=' to compare strings only works if intern version is used on both sides.Use the equals() method instead.
- AvoidStringBufferField: StringBuffers/StringBuilders can grow considerably, and so may become a source of memory leaksif held within objects with long lifetimes.

Security Code Guidelines (java)

- MethodReturnsInternalArray: Exposing internal arrays to the caller violates object encapsulation since elements can be removed or replaced outside of the object that owns it. 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 future changes from the user from affecting the original array.

Type Resolution (java)

- 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.

Unnecessary (java)

- UnnecessaryConversionTemporary: Avoid the use temporary objects when converting primitives to Strings. Use the static conversion methodson the wrapper classes instead.
- UnnecessaryReturn: Avoid the use of unnecessary return statements.
- UnnecessaryFinalModifier: When a class has the final modifier, all the methods are automatically final and do not need to betagged as such.
- UselessOverridingMethod: The overriding method merely calls the same method defined in a superclass.
- UselessOperationOnImmutable: An operation on an Immutable object (String, BigDecimal or BigInteger) won't change the object itselfsince the result of the operation is a new object. Therefore, ignoring the operation result is an error.
- 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.
- UselessParentheses: Useless parentheses should be removed.

Unused Code (java)

- 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 without actually referencing them in the method body.
- 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.