Java Coding Standard

Version:	*4.0.6*		
Date:	*2014-09-10*		
Status:	*released*		
Downloads:	*PDF versions / SonarQube profiles*		
Authors:	*FG-722*	*Christos Dovas*	*christos.dovas@bmw.de*
		Markus Radspieler	*markus.radspieler@bmw.de*
		Volker Weber	*volker.weber@bmw.de*
Review partners:	*FG-722*	*Hamed Behrouzi*	*hamed.behrouzi@bmw.de*
		Markus Dieterle	*markus.dieterle@bmw.de*
		Thomas Hanel	*thomas.hanel@bmw.de*

Document history:

Version	Date	Author(s)	Change(s)
1.0	2003-06-13	D. Menges	Created.
2.0	2004-01-29	J. Weigend	Alignment with FZ-32 coding guidelines; language shift from German to English.
3.0	2005-03-15	R. Mertingk	Explanations and multiple enhancements added.
4.0	2013-05-13	C. Dovas / M. Radspieler / V. Weber	Alignment with SonarQube quality assurance profiles; migration from MS Word to web/wiki structure.

PDF versions:

Version	Date	Document
4.0.6	2014-09-10	BMW Java Coding Standard 4.0.6
4.0.5	2014-05-23	BMW Java Coding Standard 4.0.5
4.0.0	2014-03-17	BMW Java Coding Standard 4.0.0

See Comments and suggestions on how you can contribute to the BMW Java Coding Standard.

Table of contents

with their severity, parametrization and relevance to the Java Coding Standard.

Recently Updated

Java Coding Standard updated by Markus Radspieler (view change)	about 2 hours ago
SonarQube profile updated by Markus Radspieler (view change)	about 2 hours ago
BMW_Java_Coding_Standard_4.0.6_java.xml attached by Markus Radspieler	about 2 hours ago
General coding style updated by Markus Radspieler (view change)	about 6 hours ago
A updated by Volker Weber (view change)	Sep 08, 2014
General formatting issues updated by Volker Weber (view change)	Sep 08, 2014
Java Coding Standard updated by Volker Weber (view change)	Aug 20, 2014
JUnit test implementation updated by Volker Weber (view change)	Aug 18, 2014
Constructor and initializer design updated by Volker Weber (view change)	Aug 13, 2014
Constructors and initialization updated by Markus Radspieler (view change)	Jul 16, 2014
Line formatting and whitespaces updated by Markus Radspieler (view change)	Jul 16, 2014
Overload wrapper with primitive sample updated by Markus Radspieler (view change)	Jul 16, 2014
String conversions sample updated by Markus Radspieler (view change)	Jul 16, 2014
Introduction updated by Volker Weber (view change)	Jul 15, 2014
Constructors and initialization updated by Volker Weber (view change)	Jul 08, 2014

Navigate space

Introduction

This chapter gives an overview of the entire document and how it is to be used.

Overview

This document defines fundamental standards for the implementation of Java projects at BMW. It provides a detailed list of coding rules as well as basic design principles, encouraging developers and architects to think about the design of the application prior to the actual implementation.

The main focus of the Java Coding Standard is to improve software quality. Thus, the standard provides a detailed mapping of the coding rules to the SonarQube rule sets and rule configurations. SonarQube is the core quality assurance component of the BMW Continuous Integration build and quality assurance solution for Java, .NET (C#) and the Android mobile platform.

In addition to the Java Coding Standard, BMW Master Solutions and Solution Building Blocks define solution-specific conventions, which are not covered here. See the BMW Java Community Site for details.

Contents of this document

This document consists of Java coding rules, categorized by coding aspects, and a mapping of these rules to a corresponding SonarQube rule set for the BMW Continuous Integration build and quality assurance solution:

Who should read this document?

This document is intended for participants in the implementation of Java-based applications:

- **Developers** who implement and maintain Java code.
- **Build Managers (BM)** holding responsibility for the quality of Java-based application code.
- **Technical test analysts** who are responsible that Java-based application code fulfills the required non-functional requirements for a roll-out into production.
- **Technical Build Managers (TBM)** who construct and test project artifacts like binaries, installation packages and documentation for entire applications or pre-defined fractions like modules or packages.

Scope and applicability

The standard defined in this document is binding for all Java custom code owned by BMW.

Depending on the type of Java application, some rules defined by this standard might not be appropriate for individual applications or individual application components, or rules require application-specific configurations. In this case, it is the project's responsibility to identify required deviations, agree deviations with the BMW application approval and communicate agreed deviations to the parties involved in the project. Under no circumstances may external partners decide on such exceptions from this standard on their own, without explicit approval by BMW.

Guidelines and rules defined by BMW master solutions are treated with a higher priority than the rules defined by this document and may overrule these. In this case, all rules of the Java Coding Standard which are not affected by solution-specific rules remain valid.

Notes on using this document

This document consists of two main sections: the rules section and the SonarQube profile. A Samples section which provides an overview of all samples used in the rules section, and an alphabetical Rules index complete the document.

Rules section

The rules section of the Java Coding Standard, comprising chapters Code design to Testing, defines the coding rules in a uniform structure as shown below for some type naming rules:

Java Coding Standard naming rules sample

Type name

Type (class, interface, enumeration or annotation) names are camel case, consisting of letters or digits, and must begin with an upper case letter.

Reasoning
Complies to standard SUN Java coding conventions.

SonarQube rule
Type Name

Shadowing type name

Type names must not shadow names of supertypes or implemented interfaces.

Reasoning
Avoids confusion on which class is being used and reduces risks from using syntactically ambiguous type names.

SonarQube rules
Bad practice - Class names shouldn't shadow simple name of implemented interface

Bad practice - Class names shouldn't shadow simple name of superclass

Each Java Coding Standard rule consists of:

- **Rule name** (bold face, italics)
  The rule's name.
- **Rule**
  A normative text expressing the rule's semantics: What does the rule want the programmer to do?
- **Reasoning**
  An explanation of the rationale behind the rule: Why does it make sense to apply that rule? Every rule comes with a Reasoning section.

- **Sample(s)**
  Program code samples which illustrate the rule's intention. Only some rules come with a Sample(s) section.
- **SonarQube rule(s)**
  For rules which are checked by SonarQube, a hyperlink to the corresponding rules in the SonarQube profile. For rules which don't have a SonarQube counterpart, the SonarQube rule(s) section has been omitted.

The Reasoning, Sample(s) and SonarQube rule(s) sections are expandable, being collapsed by default in order to present a higher number of rules on the screen. A mouse click on the triangle icon left of the section's title expands and collapses the section's content.

SonarQube profile

The SonarQube profile lists all SonarQube rules for Java in alphabetical order as a table with each rule as a separate line:

Severity	Rule name	Parametrization	Remarks	Coding rules
. . .	. . .	. . .	. . .	. . .
Major	Bad practice - Class names shouldn't shadow simple name of implemented interface **findbugs:NM_SAME_SIMPLE_NAME_AS_INTERFACE**			**Naming** – *Shadowing type name*
Major	Bad practice - Class names shouldn't shadow simple name of superclass **findbugs:NM_SAME_SIMPLE_NAME_AS_SUPERCLASS**			**Naming** – *Shadowing type name*
. . .	. . .	. . .	. . .	. . .

Major	Type Name checkstyle:com.puppycrawl.tools.checkstyle.checks.naming.TypeNameCheck	*format:* ^[A-Z][a-zA-Z0-9]+$ *applyToPublic:* true *applyToProtected:* true *applyToPackage:* true *applyToPrivate:* true *tokens:* CLASS_DEF, INTERFACE_DEF		**Naming** – *Type name*
...	...	...	...	...

Each SonarQube rule consists of five entries:

- **Severity**
  Indicates the impact of a rule violation, as defined in Severities. For rules which are not incorporated by the Java Coding Standard, the rule's severity entry is **inactive**.
- **Rule name**
  The rule's name and technical key in SonarQube. The name's hyperlink refers to the rule's documentation in the internet.
- **Parametrization**
  Display the exact configuration of the rule in SonarQube. The entry is empty for rules which can only be turned on or off, but cannot be configured individually.
- **Remarks**
  Additional comments. This entry is mainly used to document why inactive SonarQube rules have not been incorporated into the Java Coding Standard.
- **Coding rules**
  The rule's category and the rule entry in the Java Coding Standard. If the rule is marked as inactive, the rule's link refers to the section of the Java Coding Standard which discusses the rule's context.

Comments and suggestions

This document is intended to be a living document, which is up-to-date with new versions of the Java programming language, best practices and continuous improvements of the Java code analysis tools.

In order to leverage the knowledge of the entire Java community at BMW and its development partners, readers of this document are highly encouraged to actively provide suggestions for improvements. These improvements will be collected, discussed and, if positively approved as useful for this type of document, incorporated into a new version of the BMW Java Coding Standard. In this case, the new version will have a new minor or major version number.

Suggestions which just correct typos or enhance the understandability of particular sections of this document are directly incorporated into the standard, as they don't affect the standard's semantics.

To provide feedback, either

- use the "Write a comment..." section at the bottom of each page; in this case, **please provide your name and email at the beginning of your comment**.

or

- directly send an email to Christos Dovas (christos.dovas@bmw.de) or Markus Radspieler (markus.radspieler@bmw.de), who coordinate the development of the BMW Java Coding Standard.

Code design

This chapter discusses technical design rules for applications written in Java.

Design metrics

Metrics on Java code design.

General design rules

General patterns and rules on how to design your software.

Class and interface design

Rules on how to design classes, interfaces and type hierarchies in Java.

Constructor and initializer design

Rules on how to design constructors and initializers.

Enumeration design

Rules on how to design enumeration types in Java.

Field design

Rules on how to design static and member fields in Java.

Method design

Rules on how to design methods and method signatures in Java.

Serialization design

Rules on how to design code for serialization.

Design metrics

This section defines metrics on Java code design.

File level metrics

Outer types per file

A Java source file must contain exactly one outer type. The only exception to this rule is the `package-info.java` file, which contains a package's Javadoc and contains no type at all.

Reasoning
It is a common practice to implement each outer type in a separate file.

SonarQube rule
Outer Type Number

Statements per file

The number of non-commenting source statements (NCSS) must not exceed 2,000 statements per file.

Reasoning
Too long files are hard to maintain. Consider refactoring if the threshold is exceeded, e.g. move inner classes to separate files.

Note that this measure only counts non-commenting lines and ignores empty lines and extra lines for statements that are formatted over more than one line.

SonarQube rule
JavaNCSS

Type level metrics

Statements per type

The number of non-commenting source statements (NCSS) must not exceed 1,500 per type.

Reasoning
Too long types are hard to maintain. Consider refactoring the type into multiple types if the threshold is exceeded.

Note that this measure only counts non-commenting lines and ignores empty lines and extra lines for statements that are formatted over more than one line.

SonarQube rule
JavaNCSS

Method level metrics

Statements per method

The number of non-commenting source statements (NCSS) must not exceed 50 per method.

Reasoning
Too long methods are hard to maintain. Consider refactoring the method into multiple methods if the threshold is exceeded.

Note that this measure only counts non-commenting lines and ignores empty lines and extra lines for statements that are formatted over more than one line.

SonarQube rule
JavaNCSS

Cyclomatic complexity

The Cyclomatic complexity (McCabe metrics) should be 15 or less.

Reasoning
High cyclomatic complexity may indicate over complex methods that in return are difficult to maintain and understand. The following table lists a commonly accepted interpretation of cyclomatic complexities at method level:

CC value	complexity	interpretation
1-4	low	good: nothing to be done
5-7	moderate	still OK: nothing to be done
8-10	high	complex: consider refactoring
> 10	very high	too complex: refactor now
Mathematical surveys conducted by Thomas J. McCabe, the developer of this measure, turned out that under certain circumstances it may be appropriate to relax the restriction and permit a cyclomatic complexity of 15.		

For details, see Wikipedia and A Complexity Measure by Thomas J. MacCabe.

SonarQube rule
Cyclomatic Complexity

Constructor parameter number

Constructors must not have more than 9 parameters.

Reasoning

Classes should only implement a minimal, well defined functionality. If a class requires more than 9 parameters, it is very likely that it implements multiple aspects that can be distributed over multiple classes. Consider refactoring the class.

SonarQube rule

Excessive Parameter List

Method parameter number

Methods must not have more than 7 parameters.

Reasoning

Since methods should only perform a single and preferably simple task, only few parameters should be given. If a method takes many parameters, then it is a sign for the method trying to do to much.

SonarQube rule

Parameter Number

Block level metrics

Nested for depth

Loops should not be nested too deep, as this might make the implementation hard too read and maintain.

Reasoning

If the depth of the nesting exceeds 4, it should be checked if the implemented algorithm can be partitioned in a way, that some of the inner loops may be refactored in a reasonable way. E.g. maybe the two most inner loops could be extracted into their own method in a meaningful way.

SonarQube rule

Nested For Depth

Nested if depth

Don't nest `if` statements more than 4 levels deep.

Reasoning

Complex boolean expressions are hard to understand and are a frequent cause for bugs, not only during initial coding, but especially with respect to code maintenance. Scattering such expressions across multiple, cascading `if` statements increases this issue. As a rule of thumb, nesting of `if` statements more than 3 levels deep should be avoided.

SonarQube rule

Nested If Depth

Nested try depth

Don't nest `try` statements more than 3 levels deep.

Reasoning

Deeper nested `try` statements are hard to understand, especially with respect to the various `catch` and `finally` blocks.

SonarQube rule

Nested Try Depth

General design rules

This section discusses general patterns and rules on how to design your software.

Design principles

Leverage architecture and design standards

Use BMW architecture and design standards whenever possible, or an industry standard when no BMW standard is available.

Reasoning

BMW standards are proven solutions or solution building blocks, so there is no need to reinvent another solution which serves the same purpose. Furthermore, code which adheres to such standards is easier portable to new platforms or new versions of the standard, as the required migration steps only need to be defined and verified once and can then be applied to a series of applications which are based on the same standard.

Separation of Concerns

Separate your application into distinct features that overlap as little as possible. Implement each feature as a separate type or set of types rather than providing multiple, disjoint features as a single type.

Reasoning

Separation of Concerns is a core design principle design principle of modular software. Splitting the software into small pieces with a distinct responsibility facilitates implementation, testing and extending software and makes reuse of existing software pieces much easier.

Use design patterns

Use design patterns where applicable. If you vary a design pattern, document the variations.

Reasoning

Design patterns are well known among programmers, thus giving a quick rough understanding of the applied components, their interaction, limitations and runtime behavior. This enhances code readability and makes code maintenance easier. Furthermore, applying design patterns avoids unnecessarily re-inventing solutions for problems which have already been solved.

Avoid redundant code

Rather than repeating the same program logic multiple times, refactor such code into a separate method.

Reasoning

Code duplications have a significant impact on maintenance efforts and code stability: If the program logic needs to modified, it is hard to impossible to reliably detect and fix all duplicates concisely. This easily leads to sporadically unexpected behavior of the application and to errors that can be hard to find.

SonarQube rule

Strict Duplicate Code

Don't oversize classes

Don't write code which is too complex to be processed by static code analyzers.

Reasoning

Automated source code analysis plays an important role in quality assurance of custom code at BMW. Code which cannot be fully analyzed imposes high risks, as potentially critical or blocker violations checked by code analysis tools would remain undetected.

SonarQube rule

Dodgy - Class too big for analysis

Avoid finalizers

Finalizers are unpredictable, often dangerous, and generally unnecessary.

Reasoning

Don’t use finalizers except as a safety net or to terminate noncritical native resources. In those rare instances where you do use a finalizer, remember to invoke super.finalize. If you use a finalizer as a safety net, remember to log the invalid usage from the finalizer. Lastly, if you need to associate a finalizer with a public, nonfinal class, consider using a finalizer guardian, so finalization can take place even if a subclass finalizer fails to invoke super.finalize.

SonarQube rule

Bad practice - Finalizer does nothing but call superclass finalizer

Finalize Overloaded

Bad practice - Explicit invocation of finalizer
This rule checks for invocations of finalizers that have been erroneously declared as `public`.

Malicious code vulnerability - Finalizer should be protected, not public

Bad practice - Empty finalizer should be deleted

Bad practice - Finalizer does not call superclass finalizer

Bad practice - Finalizer nulls fields

Accessibility

Minimize accessibility

Minimize accessibility to Java artifacts.

Reasoning

This minimizes the Java artifact's external interface to those things which are required and intended for external use. It prevents artifacts designed for internal use from external misuse, and it minimizes the amount of code which needs to be maintained in order to keep the artifact's interface contract stable.

Avoid accessibility alteration

When overriding a superclass method, avoid changing the method's accessibility. If you need a method with the same semantics but a higher accessibility, add a new method with a different name which serves as a delegate to the method whose accessibility is too restricted.

Reasoning

Altering the accessibility would change the method's interface, as defined in the superclass.

SonarQube rule

Avoid Accessibility Alteration

Coupling between Java artifacts

Avoid cyclic dependencies

Avoid cyclic dependencies between Java types or packages. The only exception to this rule are cyclic dependencies injected by the programming model of specific frameworks, e.g. the parent/child or predecessor/successor relationships between controller dialogues (Android Activities, WEB UI views).

Reasoning

Cyclic dependencies significantly reduce structural clearness and maintainability of your code and are a clear indication of a poor application design. Changes to one element involved in a cycle require re-testing all other elements in the cycle, which can significantly increase the effort required to maintain an application and to embrace business change.

Consider implementing delegates to resolve such cycles, as this reduces dependencies and enables reuse of the affected elements in another application context.

SonarQube rules

Avoid cycle between java packages

Avoid unnecessary specific types

For field declarations, method/constructor parameters, return typees and local variables, use the most general type that doesn't require typecasts throughout your code. Use abstract types rather than instantiable types and interface types rather than implementation types.

Note that this rule does not apply to exception types in `catch` clause parameters.

For instances of type	use type
`java.util.GregorianCalendar`	`java.util.Calendar`

`java.util.ArrayList`
`java.util.LinkedList`
`java.util.Vector`
	`java.util.List`

`java.util.HashMap`
`java.util.Hashtable`
`java.util.LinkedHashMap`
`java.util.TreeMap`
	`java.util.Map`

`java.util.HashSet`
`java.util.LinkedHashSet`
`java.util.TreeSet`
	`java.util.Set`

Reasoning

you should favor the use of interfaces rather than classes to refer to objects. If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types. The only time you really need to refer to an object’s class is when you’re creating it with a constructor. (effective java)

Types of catch parameters and return values are exempted from this rule, as they should be as specific as possible.

SonarQube rule

Illegal Type

Coupling to runtime environment

Avoid hard-coded runtime environment parameters

Don't use hard-coded runtime configuration properties like references to absolute path names, server names or IP addresses. Instead, load such properties from an external configuration at runtime.

Reasoning

Runtime configuration parameters are likely to differ under different execution environments and should be adjustable. Defining them as constants in the code hardens the customization of an application and is likely to require code adoptions which only depend on the chosen runtime environment.

If such runtime properties are loaded from an external configuration at runtime, it is sufficient to change the external configuration, instead of changing code and rebuilding and redeploying the entire application. An easy way to decouple the code from the runtime environment is using `java.util.Properties` to retrieve runtime configurations from an external configuration file.

SonarQube rules

Avoid Using Hard Coded IP

Dodgy - Code contains a hard coded reference to an absolute pathname

Avoid hard-coded SD card references (Android)

If an application is capable of accessing the SD card, don't use hard-coded device constants. For instance, the storage directory of an Android device should be retrieved by calling `Environment.getExternalStorageDirectory()` instead of hard-coding `"/sdcard"`.

Reasoning

The device-specific property might change over time, e.g. because a device gets replaced or, in case of the Android storage directory, because a future version of the Android operating system introduces a new convention for logical device locations.

SonarQube rule

Android - Do Not Hard Code SD Card

Don't rely on default encoding

Provide a charset name or a `java.nio.charset.Charset` object to conversions between bytes and Strings.

Reasoning

Otherwise the code will assume that the platform default encoding is suitable, which would make the application behavior vary between platforms.

SonarQube rule

Reliance on default encoding

Use target JDK for development

For the developer's IDE, at least ensure that the JDK version is the same as on the target platform. For build, test and integration environments, ensure that the JDK used for development is the same with respect to vendor, version and operating system as the target platform's JDK.

Reasoning

Avoids using deprecated APIs that don't exist any more in the target runtime environment, or APIs that are not yet supported by the target runtime environment if the target JDK has a lower version number than the one used for development.

Furthermore, this is a prerequisite for the same application behavior during development and in production.

Compatibility

Maintain backward compatibility

Try to maintain backward compatibility when changing interfaces or concrete classes. Main reasons for breaking backward compatibility are

- Removal of elements from the interface
- Modifications of the accessibility of interface elements to a more restrictive accessibility
- Changes of the types of interface elements to an incompatible or more restrictive type (a subtype of the previous type)
- Changing threadsafe code into a non-threadsafe implementation
- Changing runtime preconditions to execute the code, like required privileges or access to resources

▾ Reasoning
Loosing backward compatibility requires refactoring of code which uses the artifact affected by the broken compatibility.

If backward compatibility is given for the artifact's interface, but not for its runtime behavior, code which uses the affected artifact can behave unexpectedly. Such errors are hard to detect. At least a clear documentation of the changes of the runtime behavior should be provided to facilitate avoiding or finding the cause for the unexpected behavior.

Package declaration

Declare package for all types

All Java types must provide a package declaration.

▾ Reasoning
Packages are an elementary element to structure Java code. Classes that live in the null package cannot be imported. Many novice developers are not aware of this.

▾ SonarQube rule
Package Declaration

Third-party libraries

Don't use types from internal JDK packages

Don't use internal JDK packages which are not included in the Java Language Specification, unless required to meet functional or non-functional criteria which cannot be met by the Java standard types.

▾ Reasoning

Types from package `com.sun.*` are not externally released as part of the Java API. They are subject to change or removal.

SonarQube rule
Illegal Import

Class and interface design

This section discusses how to design classes, interfaces and type hierarchies in Java.

Abstract classes

Abstract class must have at least one method

Abstract classes should have at least one method.

Reasoning
An abstract class without any methods could be also considered as an interface. If you intend to use the class as a container for nothing but constants, rather implement a concrete class with just a private default constructor and without factory methods that create instances of the class.

SonarQube rule
Abstract class without any methods

Abstract class must have abstract method

Abstract classes must have at least one abstract method, unless it doesn't implement at least one method of an interface which is implemented by the abstract class.

Reasoning

An abstract class with an implementation of all methods does not adhere to its concept as being an interface with an incomplete implementation.

If you intend to use the class as a non-instantiable interface which provides a default implementation for all of its methods, rather implement a concrete class with at least one protected constructor and without public or package private constructors or factory methods that create instances of the class.

SonarQube rule

Abstract Class Without Abstract Method

Classes without subclasses

Class with only private constructors must be declared as final

Classes that only have private constructors must be declared as `final`.

Reasoning

A class with only private constructors cannot be subclassed. Declaring the class final makes this obvious.

SonarQube rule

Final Class

Exception classes

Don't extend java.lang.Error

Don't subclass `java.lang.Error`.

Reasoning

While the Java Language Specification does not require it, there is a strong convention that errors are reserved for use by the JVM to indicate resource deficiencies, invariant failures, or other conditions that make it impossible to continue execution. Given the almost universal acceptance of this convention, it’s best not to implement any new Error subclasses.

SonarQube rule

Do Not Extend Java Lang Error

Exceptions must be immutable

Exceptions must not allow state modifications of fields that have already been initialized by the constructor.

Reasoning

The state of an exception should be set when it is thrown, as the circumstances would not change later. Make exceptions immutable by not providing a method to change the exceptions's state.

SonarQube rule
Mutable Exception

Implemented interfaces

Class must not unnecessarily implement same interface as superclass

Interfaces may only implemented once in an inheritance hierarchy, unless the interface that is implemented multiple times is a super-interface of another interface that is implemented by a class in the hierarchy.

Reasoning
Implementing the same interface multiple times on different levels in the inheritance tree of a class is usually due to incomplete refactoring and may lead to confusion when trying to debug or refactor the class.

However, an interface can be a super-interface of multiple interfaces implemented in the inheritance hierarchy, e.g. the `java.io.Serializable` interface. Thus, the purpose of this rule is to prevent that classes unnecessarily declare to be implementing interfaces which have already been declared by a superclass.

SonarQube rule
Dodgy - Class implements same interface as superclass

Comparator must implement Serializable

Classes which implement `java.util.Comparator` must also implement `java.io.Serializable`.

Reasoning
If a comparator is passed to the constructor of an ordered collection, the collection can only be serialized if the comparator is serializable. Since most collections are serializable and likely to be passed around, this rule is more than just good practice of defensive programming.

The effort to implement a serializable comparator is low, as comparators should not have any state: It is sufficient to declare the comparator implementing the `java.io.Serializable` interface.

SonarQube rule
Bad practice - Comparator doesn't implement Serializable

Nested classes

Prefer static member classes over nonstatic

If possible, nested classes should declared as `static` nested classes.

Reasoning
If an nested class does not use its embedded reference to the object which created it except during construction of the inner object, it should be declared as a static nested class. This reference makes the instances of the class larger, and may keep the reference to the creator object alive longer than necessary.

SonarQube rules
Performance - Could be refactored into a static inner class

Performance - Should be a static inner class

Declare inner class referenced from outer class ThreadLocal as static

Inner classes must be declared as `static` if the outer class contains a `java.lang.ThreadLocal` field that holds a reference to an instance of the inner class.

Reasoning
If the inner class wasn't declared as `static`, the inner and outer instance will both be reachable and not eligible for garbage collection, as the inner class instance holds an implicit reference to the outer class instance.

SonarQube rule
Correctness - Deadly embrace of non-static inner class and thread local

Interfaces

Interface must define type

Use interfaces as abstract type definitions, or in rare cases as marker interfaces without any methods like the `java.io.Serializable` interface. Don't misuse interfaces as containers for global constants.

Reasoning
An interface should only be used to characterize the external behavior of an implementing class: Using an interface as a container for constants is a poor usage pattern; rather use a non-instantiable class.

SonarQube rule
Interface Is Type

Don't use access modifiers for interface elements

Don't use access modifiers for constants and methods defined in interfaces.

Reasoning
All interface elements are `public` by definition. Access modifiers for elements defined by the interface are nothing but unused code. The only access modifier of an interface is the keyword `public` immediately before the `interface` keyword in the interface declaration.

Non-instantiable classes

Non-instantiable class must have static method or field

Non-instantiable classes without `protected` constructors must implement static methods or declare static fields.

Reasoning
Otherwise the class couldn't be used, as access to non-static methods or fields requires that a class is instantiable – either directly in case of a concrete class or indirectly by instantiating a subclass.

SonarQube rule
Missing Static Method In Non Instantiatable Class

Usage of modifiers

Don't use superfluous modifiers for nested type declarations

Don't declare classes or interfaces that are nested in interfaces or interfaces nested in a class as `public` or `static`. Only classes that are nested in classes may use modifiers.

Reasoning
Classes nested in an interface are automatically `public` and `static`. Interfaces nested in a class or interface are automatically `public`.

SonarQube rule
Redundant Modifier

Constructor and initializer design

This section discusses how to design constructors and initializers. See *Initialization of enum types* on how to initialize enum types.

Constructors

Named class requires constructor

Every named class or enumeration type requires at least one constructor, rather than relying on an implicitly generated `public` default constructor.

Reasoning
This makes obvious if and how instances of the class are to be created and prevents undesired object instantiations. If objects of the class are not to be instantiated from outside the class, e.g. in situations where

instantiation is handled by factory methods, the constructor must be declared as `private`. If a `public` constructor without arguments and without further initialization logic is sufficient, implement a non-argument constructor that just calls a superclass's constructor, e.g. `super();` in a constructor of a direct subclass of `java.lang.Object`.

SonarQube rule

[Missing Constructor]

Use delegating constructors

Avoid redundant constructor code by using delegating constructors. Aim to implement a single constructor that contains all initialization code, and have other constructors directly or indirectly delegate object initialization to this constructor.

Reasoning

Delegating constructors is a powerful concept to minimize initialization code and to ensure that objects are fully initialized. Especially when member fields are added to a class, delegating constructors aid in ensuring that all constructors also correctly initialize the new fields.

Sample

See *Don't use non-static initializers* for a sample on how to implement a class with a very similar functionality in a far less elegant fashion by using a non-static initializer for object initialization.

Delegating constructor

```
 1  public class Iso3166Alpha2CountryCode {
 2
 3      // The initial version number.
 4      public static final int INITIAL_VERSION = 1;
 5
 6      // Objects of this class shall be immutable. Thus, all fields are declared as final.
 7      private final String alpha2Code;  // Mandatory, consisting of 2 upper case letters.
 8      private final int    codeVersion;
 9
10      // 2-argument constructor which initializes all member fields with parameter values.
11      public Iso3166Alpha2CountryCode(final String alpha2Code,
12                                      final int    codeVersion) {
13          super();
14          if (alpha2Code == null) {
15              throw new IllegalArgumentException("Missing ISO 3166-1 alpha-2 country code."); //$NON-NLS-1$
16          }
17          if ((alpha2Code.length() != 2) || (! alpha2Code.equals(alpha2Code.toUpperCase()))) {
18              throw new IllegalArgumentException("Invalid ISO 3166-1 alpha-2 country code."); //$NON-NLS-1$
19          }
20          if (codeVersion < INITIAL_VERSION) {
21              throw new IllegalArgumentException("Invalid code version."); //$NON-NLS-1$
22          }
23          this.alpha2Code  = alpha2Code;
24          this.codeVersion = codeVersion;
25      }
26
27      // Constructor from a parameter list that doesn't contain values for all member fields.
28      public Iso3166Alpha2CountryCode(final String alpha2Code) {
29          // Instead of repeating initialization code from the constructor above as in ...
30          super();
31          if(alpha2Code == null) {
32              throw new IllegalArgumentException("Missing ISO 3166-1 alpha-2 country code."); //$NON-NLS-1$
33          }
34          if ((alpha2Code.length() != 2) || (! alpha2Code.equals(alpha2Code.toUpperCase()))) {
35              throw new IllegalArgumentException("Invalid ISO 3166-1 alpha-2 country code."); //$NON-NLS-1$
36          }
37          this.alpha2Code  = alpha2Code;
38          this.codeVersion = INITIAL_VERSION;
39
40          // ... delegate initialization to the other constructor and replace lines 30-38 by just:
41          this(alpha2Code, INITIAL_VERSION); // The delegating call of the other constructor.
42      }
43
44      // Only getters, no setters, since this is an immutable class.
45      public final String getAlpha2Code() {
46          return this.alpha2Code;
47      }
48
49      public final int getCodeVersion() {
50          return this.codeVersion;
51      }
52  }
53
```

Static initializers

Implement static initializer only if required

Only implement a static initializer if you need complex initialization during class loading.

Reasoning
The primary purpose of static initializer blocks is, to execute code that cannot be included in an initialization expression that is part of a static variable declaration.

Avoid initialization circularity

Static initializers must be circle free.

Reasoning
Using circles in static initializers may result in an infinite loop (leading to a stack overflow) or some strange behaviour due to partially initialized objects being used. Furthermore, such dependencies violate the fundamental design principle Avoid cyclic dependencies on a type level.

SonarQube rule
Dodgy - Initialization circularity

Make class with only static members uninstantiable

Make classes that only contain static fields and methods uninstantiable by

- declaring the class as `abstract` and providing a `protected` no-argument constructor if the class should be subclassed;
- declaring the class as `final` and providing a `private` no-argument constructor if the class is not to be subclassed.

Most often this is the case for utility classes that only contain static methods and fields.

Reasoning
Prevents developers from erroneously creating instances of classes where instantiation is not needed and thus reduces the memory footprint of the application.

SonarQube rule
Hide Utility Class Constructor

Initialization of member fields

Don't use non-static initializers

Never use non-static initializers (except for anonymous inner classes), but only constructors for object initialization.

Reasoning

There is no need for non-static initializers, as we have constructors at hand when the class is not an anonymous inner class. Thus, using non-static initializers only makes code unnecessarily complicated and confusing: To understand an object's state after creation, the programmer would have to read the non-static initializer's code in addition to the constructor's code, instead of having it all in the constructor.

If the same initialization code shall be shared by multiple constructors, use delegating constructors (best) or a `private final void init()` method that contains the shared code and is called by multiple constructors rather than implementing a non-static initializer.

Sample

See *Use delegating constructors* for a sample on how to better implement object initialization and avoid using a non-static initializer.

Non-static initializer

```java
public class Iso3166Alpha2CountryCode {

    // The initial version number.
    public static final int INITIAL_VERSION = 1;

    // Objects of this class shall be immutable. Thus, all fields should be declared as final.
    private final String mAlpha2Code;  // Mandatory, consisting of 2 upper case letters.
    private int codeVersion; // Cannot be final, since it is assigned a value twice.

    // The non-static initializer, setting the default value for codeVersion.
    {
        this.codeVersion = INITIAL_VERSION; // First assignment to codeVersion.
    }

    // 2-argument constructor which initializes all member fields with parameter values.
    public Iso3166Alpha2CountryCode(final String alpha2Code,
                                    final int    codeVersion) {
        this(alpha2Code);
        if (codeVersion < INITIAL_VERSION) {
            throw new IllegalArgumentException("Invalid code version."); //$NON-NLS-1$
        }
        this.codeVersion = codeVersion; // Second assignment to mCodeVersion.
    }

    // Constructor from a parameter list that doesn't contain values for all member fields.
    // codeVersion is initialized by the non-static initializer.
    public Iso3166Alpha2CountryCode(final String alpha2Code) {
        if (alpha2Code == null) {
            throw new IllegalArgumentException("Missing alpha-2 ISO country code."); //$NON-NLS-1$
        }
        if ((alpha2Code.length() != 2) || (! alpha2Code.equals(alpha2Code.toUpperCase()))) {
            throw new IllegalArgumentException("Invalid alpha-2 ISO country code."); //$NON-NLS-1$
        }
        this.alpha2Code = alpha2Code;
    }

    // Only getters, no setters, since this is an immutable class.
    public final String getAlpha2Code() {
        return this.alpha2Code;
    }

    public final int getCodeVersion() {
        return this.codeVersion;
    }
}

```

SonarQube rule

Non Static Initializer

Use private init() method to avoid too long constructors

If constructor code is complex and long, consider extracting initialization code into a separate method that is called by the constructor(s).

Name such a method `init()` to make its purpose obvious.

The method must be declared as `private`.

Reasoning

An external method for object initialization can be useful in case of complex initialization processing or if otherwise multiple constructors of the same class would need to implement redundant initialization code. The latter case can also be well addressed by delegating constructors.

The `init()` method must be declared as `private` to prevent other classes like subclasses from calling the method, and it assures that the method is `final`, which is essential for code that is executed during object creation. Note that the `init()` method is also executed when an instance of a subclass is constructed, as it is called by the constructor of the superclass. Thus, subclasses don't require access to the method for the purpose of ensuring correct initialization of the superclass's fields.

Use init() method to initialize anonymous inner class

To initialize an anonymous inner class, have the anonymous class implement an `init()` method that acts like a constructor. The `init()` method must return `this`.

Reasoning

In contrast to other classes, anonymous classes don't have the concept of constructors. However, in some situations it is useful to initialize the anonymous class's instance before it is being used. The implementation of the `init()` method substitutes the missing constructor. The method must return `this` because anonymous classes are declared as part of an assignment to a variable of the surrounding class.

Sample

Anonymous inner class initialization

```
import java.util.Collection;
import java.util.Iterator;

public class MyCollectionType<E>
implements   Collection<E> {

    ...

    // Returns an instance of an anonymous inner class:
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private int index;

            // The emulated constructor:
            Iterator<E> init(final int index) {
                this.index = index;
                return this; // An emulated constructor must always return this.
            }

            public E next() {
                ...
            }
            ...
        }.init(0); // The invocation of the emulated constructor.
    }
}

```

Initialization of applets

Initialize applets in the init() method

Initializate applets in the `init()` method, not in the constructor.

Reasoning

Since the `AppletStub` is only initialized after the `init()` method has been called, code that is invoked in the constructor may refer to uninitialized parts of the applet. To avoid this situation, an applet's initialization must be done in the `init()` method.

Note

Due to massive problems with Java browser plugins and the surge of alternate technologies, Applets should not be used anymore.

SonarQube rule
Experimental - Bad Applet Constructor relies on uninitialized AppletStub

Initialization of XML

Use generic means to instantiate XML types

Don't allocate vendor-specific implementations of XML interfaces. Rather use the provided factory classes to create these objects, such as

- `javax.xml.parsers.DocumentBuilderFactory`
- `javax.xml.parsers.SAXParserFactory`
- `javax.xml.transform.TransformerFactory`
- and the `createXXX(...)` methods of class `org.w3c.dom.Document`.

Reasoning
Using generic factories and object creation methods allows for changing the XML implementation at runtime.

SonarQube rule
Dodgy - Method directly allocates a specific implementation of xml interfaces

Enumeration design

This section discusses how to design enumeration types in Java.

Enum type

Implement enumerations as types

Implement codomains with a fixed set of discrete values as enumeration types rather than plain Java constants. Especially, implement numerical or physical units as enumeration types.

Reasoning
Java 5 introduced `enum` as another type keyword besides `interface` and `class` in order to provide an easy way to implement enumerations as types, which enhances type safety and correctness, as it limits assignments

of enumerated values to fields of the `enum`'s type and prevents accidental assignments of other values to such fields.

When applied to units, this is an important prerequisite for the implementation of type-safe and stable measure types.

Make enums immutable

Assure that enum values are immutable, i.e. an `enum` must not implement setters for its member fields.

Reasoning
Enumerations are are more convenient and type-safe kind of invariants or constants, with each value being instantiated exactly once. Thus, allowing programmatic changes of the state of an `enum` value may compromise the application's integrity, as such modifications affect all instances which hold or will in the future have a reference to the modified value. This fundamentally contradicts the concept of an enumerated value.

Initialization of enum types

Declare enum constructors as private

Constructors of `enum` types should be declared as `private`.

Reasoning
Constructors of `enum` types are only invoked by the `enum` type itself. As `enum` classes can't be subclassed, there is no need to declare constructors `protected` to be accessible from subclasses.

Field design

This section discusses how to design static and member fields in Java.

Field declarations

Don't use modifiers in interface field declarations

In interfaces, don't declare fields as `public` and/or `final`.

Reasoning

Fields declared in an interface are automatically `public final` constants.

SonarQube rule

Redundant Modifier

Declare externally accessible fields as final

Only fields that are declared as `static final` (constants) may be `public`. All other field may only be accessed via getter and setter methods. The only exemption from this rule are fields of `private` non-static inner classes and anonymous inner classes: For such classes, fields may be declared as package private (without access modifier), but never `public` or `protected`.

Reasoning

Prevents modifications of fields beyond control of their owning class.

Fields of `private` non-static inner classes and anonymous inner classes are only accessible from their surrounding class. Thus, modifications of such fields can only me made from code within the same class file. Declaring such fields as `public` or `protected` is useless and just confusing due to missing accessibility from outside the surrounding class.

SonarQube rule

Visibility Modifier

Declare immutable fields as final

Declare fields that are only written during initialization or by a constructor `final`.

Reasoning

This documents the immutable state of a class, prevents from accidentally updating such fields and aids in converting classes to immutable classes.

SonarQube rule

Immutable Field

Declare fields of immutable classes as final

Fields of immutable classes must be declared as `final`.

Reasoning

Declaring the field as `final` protects it from unintended modifications, which would otherwise make instances of the class mutable.

SonarQube rule

Bad practice - Fields of immutable classes should be final

This rule checks all classes annotated with `net.jcip.annotations.Immutable`.

Declare static EJB fields as final

Static fields in Enterprise Java Beans must be declared as `final`. Don't use such fields to store state that is specific to a session or transaction.

▾ Reasoning
Required by the JEE specification:

An enterprise bean must not use read/write static fields. Using read-only static fields is allowed. Therefore, it is recommended that all static fields in the enterprise bean class be declared as final. This rule is required to ensure consistent runtime semantics because while some EJB containers may use a single JVM to execute all enterprise bean’s instances, others may distribute the instances across multiple JVMs.

▾ SonarQube rule
Static EJB Field Should Be Final

Constants

Don't redefine standard Java constants

Don't redefine constants that are already defined by the Java standard.

▾ Reasoning
Useless code and a potential cause for errors in comparisons of such redefined constants with the equivalent Java standard constants.

Define constants for literals

Define constants for all literals that

- are accessible from outside the Java type and could be used to steer the control flow or be mapped to other objects, e.g. literals assigned to fields if the values are externally visible, or literals that are mapped to GUI elements like keys of values that are displayed in a list box;
- have a special meaning, e.g. literals which represent an identifier, a status or a scientific constant;
- control the global runtime behavior of the code, e.g. timeouts for waits or lock acquisitions; declare such constants in a global class for constants;
- are frequently used in your code.

Use these constants in your code rather than repeating the literals.

▾ Reasoning
Declaring literals which are accessible from outside as constants enables decoupling of external code from the constant's value: Both the internal code and external code reference the constant instead of using copies of the literal and thus don't need to be systematically refactored if the constant's value changes in a later version of the code, as otherwise the application's correctness and stability could be affected in a negative way.

Declaring literals with a special meaning as constants documents the constant's semantics, which should be expressed by the constant name and explained in the constant's Javadoc. The code becomes better readable, as the constant's name should be self-explaining, whereas its value isn't in most cases.

If literals that control the runtime behavior are declared as global constants, they can be maintained at a central place.

Declaring frequently used literals as constants eliminates redundancies and maintenance thereof, and it reduces the size of Java class files.

No constants are required if there is no positive effect on the correctness/stability of the application or the readability/maintainability of the code.

Declare fields for constants as static

Fields that are initialized to a compile-time static value and that are not modified at runtime should be declared as `static final`.

Reasoning

According to the principle of least surprise, constants should be declared as `static final` fields. There is no reason to declare such fields instance member fields.

SonarQube rule

Performance - Unread field: should this field be static? (checks only for access modifier `final`)

Illegal fields

Don't declare fields in stateless classes

Stateless classes must not contain member fields.

Reasoning

Declaring a member field in a stateless class contradicts the class's property of being stateless. Furthermore, stateless objects can be shared by multiple threads or sessions with independent contexts. If these manipulate the content of the member field, isolation of the context might get lost. For instance, this is the case for Servlets and Struts Actions, which are only instantiated once.

SonarQube rules

Dodgy - Class extends Servlet class and uses instance variables

Dodgy - Class extends Struts Action class and uses instance variables

Fields must not be of a subclass type

Static or member fields of a class must not have a type which is a subtype of that class.

Reasoning

Using subtypes within a supertype causes circular dependencies.

Don't store method local state in fields

Don't use static or member fields to store method local state. Rather use local variables.

Reasoning

State that is consistently set at the beginning of methods only to be retrieved later by the same methods or state that is only accessed by a single method can be stored in local variables. This doesn't spoil the class's state and avoids unexpected side-effects in case of concurrent or re-entrant access to the instances of the class.

SonarQube rules

Singular Field

Stateless class must not have fields other than constants

Don't declare fields others than constants in classes that are designed to be stateless. Only use method local variables instead.

Reasoning

Static and member fields store the class's resp. instance's state. Thus, a class with fields isn't stateless.

Instances of stateless classes are intended to be created only once and be reused over and over again. If an instance holds state, that state is still there when the instance is reused, although the previously assigned state doesn't make sense in the context of the reuse. If the state is specific to a session or transaction, this would violate the session's or transaction's integrity.

Furthermore, objects referenced by the instance cannot be garbage collected as long as the instance hasn't reached the end of its lifecycle. This may be never in case of stateless SessionBeans, servlets or Struts Actions.

SonarQube rules

Dodgy - Class extends Servlet class and uses instance variables

Dodgy - Class extends Struts Action class and uses instance variables

Don't mask superclass fields

Classes may not mask externally accessible superclass fields.

Reasoning

Prevents confusion, as it becomes unclear if code in the class that masks the superclass's field or in one of its subclasses accesses the masking field by intention or if the superclass field was meant.

SonarQube rule

Correctness - Class defines field that masks a superclass field

Superfluous fields

Remove unused fields

Don't leave non `static final` fields in the code that are never used. This includes fields that are declared and modified by the code, but never read, and fields that are always set to `null`.

Reasoning

Such fields are just dead code. Removing them enhances code readability, maintenance and performance without impact on the application.

SonarQube rules

Correctness - Field only ever set to null

Performance - Unread field

Correctness - Unwritten field

Method design

This section discusses how to design methods and method signatures in Java.

Method declarations

Don't use modifiers in interface method declarations

In interfaces, don't declare methods as `public`, `abstract` and/or `final`.

Reasoning

Methods declared in an interface are automatically `public abstract`, a declaration as `final` is just ignored.

SonarQube rule

Redundant Modifier

Don't declare final class methods as final

Don't declare methods of final classes as `final`.

Reasoning

Methods of classes declared as `final` are automatically final.

SonarQube rule

Redundant Modifier

Methods of anonymous classes

Not externally callable methods of anonymous classes must be private

Declare all methods of an anonymous class as `private` that are not defined by a superclass or an interface that is implemented by the anonymous class and that are not directly invoked by the declaration of the variable or field of the outer class that defines the instance of the anonymous class.

Reasoning

On instances of an anonymous class, it is only possible to externally invoke

- Methods that are defined by a superclass or an interface that is implemented by the anonymous class;
- A method that is directly called by the declaration of the instance of the anonymous class, e.g. an `init()` method that substitutes a constructor as shown in Use init() method to initialize anonymous inner class.

Declaring all other methods `private` visualizes that those methods cannot be called, except from the anonymous class itself.

Anonymous classes must not contain uncallable methods

Don't implement methods in an anonymous class that are not

- called by the anonymous class, nor
- defined by a superclass or an interface that is implemented by the anonymous class, nor
- directly called by the declaration of the instance of the anonymous class, e.g. an `init()` method that substitutes a constructor as shown in Use init() method to initialize anonymous inner class.

Reasoning

Such methods are just dead code, since they cannot be invoked from outside an anonymous class and are not used by the class itself.

SonarQube rule

Correctness - Uncallable method defined in anonymous class

clone() method

Avoid clone()

It is recommended to avoid implementing the `clone()` method. Instead, implement a copy constructor or a static factory method which accept the object to be clones as an argument if you need to create a duplicate of an object.

Reasoning

The `clone()` method relies on rules which prevent a reasonable implementation in some situations, e.g. when dealing with inheritance.

- The class must return the same object as returned from `super.clone()`. If this is correctly implemented for a class, it will fail for all subclasses, as the returned object will no longer be an instance of the subclass.
- The `clone()` method does not take an argument. Thus, a subclass that attempts to return a duplicate of itself cannot safely set the fields of the superclass if the superclass does not provide an initialization method from another instance of the superclass.

The copy constructor circumvents these issues, and the copy constuctor from the subclass can just call the copy constructor of the superclass as its first statement.

SonarQube rule

No Clone

clone() requires Cloneable

A class that implements the `clone()` method must also implement the `java.lang.Cloneable` interface.

Reasoning

If a class overrides `java.lang.Object.clone()`, it must maintain the method's semantics, expressed by declaring the class implementing the `java.lang.Cloneable` interface.

SonarQube rule

Bad practice - Class defines clone() but doesn't implement Cloneable

Cloneable requires clone()

A class that implements the `java.lang.Cloneable` interface must also implement the `clone()` method.

Reasoning

The `java.lang.Cloneable` Interface is a marker interface to declare that the given class implements an own `clone()` method instead of inheriting the implementation from `java.lang.Object`. Implementing this interface without defining the method would contradict itself.

SonarQube rule

Bad practice - Class implements Cloneable but does not define or use clone method

Don't provide variants of clone()

Don't overload the `clone()` method or provide methods with similar names.

Reasoning

Method variants with parameter lists or just a similar name like `cloneTo()` are just confusing and don't behave as one might expect.

clone() must call super.clone()

Implementations of the `clone()` method must call `super.clone()`.

Reasoning

Ensures that the returned cloned object instance has been correctly constructed across the entire inheritance hierarchy.

SonarQube rule

Bad practice - clone method does not call super.clone()

clone() must not modify source object

The source object must not be modified in a `clone()` implementation. All changes are to be made on the cloned object.

Reasoning

Calls to `clone()` are considered as read only operations and should be implemented as such. If the source object is changed, this is an indication for an invocation of the modifying operation on the wrong object.

clone() must not return null

A `clone()` implementation must not return `null`.

Reasoning

A `clone()` method that can return `null` violates the clone contract.

SonarQube rule

Bad practice - Clone method may return null

Non-duplicating clone() must throw CloneNotSupportedException

A `clone()` method that does not return a copy (the precise meaning of 'copy' may depend on the class of the object) of the object that it has been invoked on must throw a `java.lang.CloneNotSupportedException`.

Reasoning
Required by the interface contract of the `clone()` method.

SonarQube rule
Clone Throws CloneNotSupportedException

compareTo() method

compareTo() requires Comparable

Don't provide a `compareTo()` method unless the class implements `java.lang.Comparable`. Otherwise, choose a different method name.

Reasoning
The `compareTo()` method is a clear indicator that the class implements Comparable. It is confusing if a class implements this method, but not the `java.lang.Comparable` interface.

compareTo() requires equals() and HashCode()

A class that implements a `compareTo(Object)` method must also override `equals(Object)` and `hashCode ()`.

Reasoning
The `compareTo(Object)` is supposed to return `0` if and only if the `equals(Object)` return `true`.compared objects equal. Thus, if the class defines the behavior of `compareTo(Object)`, it must also provide a consistent implementation of `equals(Object)`. In case of inconsitent behavior of the two methods, unpredictable failures may occur, e.g. when using collection types.

The hashCode() method must be overridden when equals(Object) is overridden. See *equals() requires hashCode()* for details.

SonarQube rule
Bad practice - Class defines compareTo(...) and uses Object.equals()

Don't provide variants of compareTo()

Don't overload the `compareTo(Object)` method or provide methods with similar names.

Reasoning
A `compareTo()` method with a type other than `java.lang.Object` will not be called when the specific type is cast to `Object` before being given to the `compareTo()` method. Therefore `java.lang.Object` must be used as type for the parameter.

Method variants with different parameter lists or just a similar name like `compareto()` are just confusing and don't behave as one might expect.

SonarQube rules

Bad practice - Abstract class defines covariant compareTo() method

Bad practice - Covariant compareTo() method defined

equals() method

equals() requires hashCode()

A class that overrides the `equals(Object)` method must also override the `hashCode()` method. A class must override both methods if it inherits an implementation of `equals(Object)` from a possibly abstract superclass that doesn't also provide an implementation of `hashCode()`.

Reasoning

It's a Java invariant that two objects which equal must also return the same hash code. If only `equals(Object)` is overridden but not `hashCode()`, two different instances may equal, but still have different hash codes.

> The hash code returned from the `hashCode()` implementation in `java.lang.Object` returns the object identity's hash code for most JVMs. Thus, different objects also have different hash codes.

SonarQube rules

Bad practice - Class inherits equals() and uses Object.hashCode()

Correctness - Signature declares use of unhashable class in hashed construct

Correctness - Use of class without a hashCode() method in a hashed data structure

Equals Hash Code

Class which adds fields must override inherited equals()

A class which defines member fields and inherits an overridden implementation of `Object.equals(Object)` must provide an own implementation of `equals(Object)`.

Reasoning

In comparisons for equality, the inherited implementation of `equals(Object)` does not include the member fields added by the class, which leads to a wrong comparison result if the values of the added fields don't equal for the added fields. If the added fields are irrelevant to compute the object's equality, consider aggregating an instance of the superclass rather than inheriting from the superclass. In this case, invoke equals on the aggregated superclass instance instead of an instance of this class.

SonarQube rule

Dodgy - Class doesn't override equals in superclass

Don't provide variants of equals()

Don't overload the `equals(Object)` method or provide methods with similar names.

Reasoning

An `equals()` method with a type other than `java.lang.Object` will not be called when the specific type is cast to `Object` before being given to the `equals()` method. Therefore `java.lang.Object` must be used as type for the parameter.

Method variants with different parameter lists or just a similar name like `equal()` are just confusing and don't behave as one might expect.

SonarQube rules

Bad practice - Abstract class defines covariant equals() method

Bad practice - Covariant equals() method defined

Correctness - Covariant equals() method defined for enum

Correctness - Covariant equals() method defined, Object.equals(Object) inherited

Correctness - equals() method defined that doesn't override equals(Object)

Correctness - equals() method defined that doesn't override Object.equals(Object)

Naming - Suspicious equals method name

hashCode() method

hashCode() requires equals()

A class that overrides the `hashCode()` method must also override the `equals()` method.

Reasoning

It's a Java invariant that two objects which equal must also return the same hash code. If only `hashCode()` is overridden but not `equals(Object)`, an inherited implementation of `equals(Object)` may return `true` for two objects with different hash codes returned from the overridden `hashCode()` method.

SonarQube rules

Bad practice - Class defines hashCode() and uses Object.equals()

Bad practice - Class defines hashCode() but not equals()

Don't provide variants of hashCode()

Don't overload the `hashCode()` method or provide methods with similar names.

Reasoning

Method variants with parameter lists or just a similar name like `hashcode()` are just confusing and don't behave as one might expect.

SonarQube rule

Naming - Suspicious Hashcode method name

Method parameters

Avoid unused method parameters

Don't declare method parameters that are not used by the method's implementation, unless

- the parameter has been declared by a method that is overridden by this class, or
- the method's surrounding class and the method are non-final and the parameter is expected to be required by an overriding implementation in a subclass.

Document unused parameters in the method's Javadoc.

Reasoning

Unused parameters are just confusing and might hinder programmers from calling the method because they cannot provide a reasonable value for the parameter. In case of overridden methods, not all implementations of the method must use the full parameter set.

Declare parameters of method implementations as final

Declare parameter of constructors and non-abstract methods as `final`. Note that method declarations in interfaces are abstract by nature.

Reasoning

Declaring parameters as `final` reduces the risk of introducing coding errors by accidently assigning another reference to the parameter producing unwanted and potentially harmful side effects.

SonarQube rules

Correctness - A parameter is dead upon entry to a method but overwritten

Final Parameters

Don't declare parameters of abstract methods as final

Don't declare method parameters in interfaces or abstract methods as `final`.

Reasoning

The `final` modifier in a parameter declaration of an abstract method is just ignored. The Java specification doesn't hinder implementations of the method from using the parameter in a non-final way, although it has been declared as `final` in the abstract method declaration.

Methods declared in interfaces are abstract by definition.

SonarQube rule

Redundant Modifier

Don't declare unnecessary specific parameter type

Declare the parameter type as general as possible.

Reasoning

Defining the parameter type more concrete than required by the method's code can unnecessarily increase the impact of future changes. Consider a more abstract parameter type or overriding a method in a subclass.

If the parameter type is more abstract than required by the method implementation, the method needs to check the type of an argument that has been passed for the parameter at runtime and cast it to the more concrete type. This reduces the code's stability and at the same time pollutes the method's signature, since the method promises to accept arguments of a larger range of types than it really does.

Don't overload wrapper parameters with primitives of a higher value range

Don't overload a method with a parameter of a wrapper type (`Byte`, `Character`, `Short`, `Integer`, `Long`, `Float` or `Double`) with a method with the same signature, except that the wrapper type is replaced by a primitive (byte (8 bit), char (16 bit), short (16 bit), int (32 bit), long (64 bit), float (32 bit) or double (64 bit)) with a higher value range.

Reasoning

As auto-boxing is present in Java, a call that passes an argument of the wrapper's corresponding primitive type will not call the method with the wrapper type, but the method with a primitive if that primitive has the same or a higher value range. The reason for this is that the primitive argument is auto-boxed to the primitive with the higher value range.

As an example, if the methods are declared one with a `Character` parameter and the other with an `int` parameter, a call to the method that passes a `char` argument will invoke the method with the `int` parameter. In case of a lower value range of the primitive parameter type, e.g. a byte in the overriding method of the previous sample, auto-boxing will create an intermediate `Character` instance and invoke the method with the `Character` parameter.

Sample

Overload wrapper with primitive

```java
public class Autoboxing {

    public static void a(Character x) {
        System.out.println("Character " + x);
    }

    public static void a(int x) {
        System.out.println("int " + x);
    }

    public static void main(String[] args) {
        a('z'); // Output: int 122
    }
}
```

Don't overload methods only by variants of the package of parameter types

Don't design a method with a parameter list that is almost identical with the parameter list of another method of the same class or a superclass, except that the package of the parameters' types is different for one or more parameters.

Reasoning

Such a method appears to be overriding a superclass method, although it in fact adds a new method with a slightly different signature. The difference is the method signatures is not obvious, which makes such constructions very confusing for programmers and provokes programming errors and unintended runtime behavior of the code.

Try to consolidate the two methods and declare one of them deprecated.

SonarQube rules

Bad practice - Method doesn't override method in superclass due to wrong package for parameter

Correctness - Method doesn't override method in superclass due to wrong package for parameter

Method return type

Declare a specific method return type

Use specific types in declarations of the return type of a method. If the externally relevant interface of the returned object type is defined by an interface that is implemented by the object's class, return the interface type rather than its concrete implementation type, e.g. for collections.

Reasoning

Specifying a specific type makes the method's behavior clearer and can free code that invokes the method from the hassle of downcasting from a too general type the object's more specific type: In most situations, returning Object is neither meaningful, nor helpful for the caller of the method. Furthermore, such typecasts are critical since they might no longer work if your method is changed in a way to also return objects of a more general type.

> An overriding method may well specify a more specific return type than the overridden method.

Throws clause

Don't declare generic exceptions in throws clause

Be specific with the exceptions declared in the method's throws clause. Don't declare `throws Exception` or even more unspecific `throws Throwable`.

Reasoning

Such unspecific exception declarations contradict the purpose of exceptions and proper exception handling. They just serve to enforce that the compiler accepts the code, but make it hard to handle the thrown exception and corner cases correctly if you call a method with such a poor exception declaration.

SonarQube rules

Signature Declare Throws Exception

Don't declare RuntimeExceptions in throws clause

Don't declare subtypes of `java.lang.RuntimeException` in the method's `throws` clause. Use the method's Javadoc to document such exceptions.

Reasoning

Runtime exceptions don't need to be declared and are ignored by the compiler. So they are just superfluous code.

Don't declare errors in throws clause

Don't declare subtypes of `java.lang.Error` in the method's `throws` clause.

Reasoning

Errors should never be declared in method signatures or be explicitly thrown in method implementations.

▼ SonarQube rule
Redundant Throws

Don't declare subtypes of declared exceptions in throws clause

If a method can throw both an instance of an exception and instance of a subtype of the exception, only declare the supertype exception in the method's `throws` clause. Use the method's Javadoc to document the subtype.

▼ Reasoning
The throws clause is relevant to the compiler, whereas the Javadoc is to the programmer. For the compiler, declaring the supertype exception is sufficient.

▼ SonarQube rule
Redundant Throws

Superfluous methods

Remove uncalled private methods

Remove private methods that are never called.

▼ Reasoning
Facilitates maintenance, as such methods are just dead code.

▼ SonarQube rule
Performance - Private method is never called

Remove overriding methods that only call the overridden method

Remove methods which override a superclass method by just calling the overridden superclass method.

▼ Reasoning
Such code is useless, creates little computational overhead and unnecessarily increases the amount of code to be maintained.

▼ SonarQube rule
Experimental - Method superfluously delegates to parent class method

Remove overriding methods that only copy the overridden method's code

Remove methods which override a superclass method by an exact copy of the overridden method's code.

▼ Reasoning
Such code is useless, creates little computational overhead and unnecessarily increases the amount of code to be maintained.

Don't redefine methods from implemented interfaces

Don't declare abstract methods which are already defined in an implemented interface.

Reasoning

The abstract method declaration provides no additional value and is just confusing.

SonarQube rule

Experimental - Abstract Method is already defined in implemented interface

Serialization design

This section discusses how to design code for serialization.

Types

Java beans must be serializable

Java beans must implement the `java.io.Serializable` interface.

Reasoning

Java beans are intended to be serialized and de-serialized. Thus, they must be serializable as well as each non-primitive member of an instance field that is not declared as `transient`.

Outer class of serializable non-static inner class must be serializable

Non-static inner classes may only be serializable if their surrounding class is also serializable.

If serializing the outer class instance is not required, make the inner class a `static` inner class or another outer class to resolve the problem rather than making the outer class serializable. If the instance of such a static inner class requires a reference to an instance of the surrounding class, declare the field that holds the reference as `transient`.

Reasoning

Instances of non-static inner classes implicitly hold a reference to the instance of their outer class. Thus, if the instance of the outer class will be serialized when the inner class instance is serialized. This leads to a `NotSerializableException` if the outer class is not serializable.

SonarQube rules

Bad practice - Non-serializable class has a serializable inner class

Fields

Serializable class must have serialVersionUID

A class that implements the java.io.Serializable interface must provide a `private static final long serialVersionUID` that contains a unique id for the artifact and its version.

Reasoning

The Java Runtime uses the serial version uid to determine if the class of the serialized object has the same version as the class of the object to be created during deserialization. If the `serialVersionUID` exists, it gets streamed into the object's output stream during serialization. When deserialized, the `serialVersionUID` in the stream is compared against the `serialVersionUID` of the target class. In case of a mismatch a `java.io.InvalidClassException` is thrown.

SonarQube rules

Bad practice - serialVersionUID isn't final

Bad practice - serialVersionUID isn't long

Bad practice - serialVersionUID isn't static

Missing Serial Version UID

Only declare fields of serializable classes as transient

If a field is declared as transient, the class must implement the `java.io.Serializable` interface.

Reasoning

A field marked as `transient` indicates that the class is exempted from serialization. Thus, this modifier does nothing and is only confusing or misleading if it is used for a field of a non-serializable class.

SonarQube rule

Dodgy - Transient field of class that isn't Serializable

Types of serialized fields must be serializable

For serializable classes, the types of all non-primitive fields that are not declared as `transient` or `volatile` must be serializable, unless the class implements

- the `readObject()`/`writeObject()` methods, or
- the `java.io.Externalizable` interface.

Reasoning

All fields that are not declared as `transient` or `volatile` are subject to default serialization. Thus, default deserialization will fail if such a field refers to a non-serializable object.

In case of a custom implementation for serialization/deserialization, the custom code must take care of correct serialization/deserialization and may also support fields that reference non-serializable objects.

SonarQube rule

Non-transient non-serializable instance field in serializable class

Don't store non serializable object into HTTP session

Objects of non-serializable types must not be stored into HTTP sessions.

Reasoning

Storing non-seriable objects into HTTP sessions leads to errors when the session is passivated or migrated.

SonarQube rule

Bad practice - Store of non serializable object into HttpSession

Constructors

Superclass of class that implements Serializable must provide no-argument constructor

Assure that the non-serializable superclass of a class that implements the `java.io.Serializable` interface has a no-argument constructor and that this constructor is accessible from the serializable class. This is not required if the class implements its own serialization and deserialization by implementing the `java.io.Externalizable` interface.

Reasoning

During deserialization, the state of a non-serializable superclass is restored by invoking the no-argument constructor of the superclass.

See the Java Object Serialization FAQ for details.

In case of a custom implementation for serialization/deserialization, the serializable class implements the `java.io.Externalizable` interface, which is a sub-interface of `java.io.Serializable`. The Java Runtime calls the no-argument constructor of the externalizable class to construct an empty object, and is depends on the implementation of this constructor which constructor of the non-serializable superclass is being invoked.

SonarQube rule

Serializable but its superclass doesn't define a void constructor

Class which implements Externalizable must provide no-argument constructor

A class which implements the `java.io.Externalizable` interface must also provide a `public` no-argument constructor.

Reasoning

By implementing the `java.io.Externalizable` interface, Java's default serialization is replaced by a custom implementation. In this case, the Java Runtime calls the class's no-argument constructor to construct an empty object before the custom implementation of the `readExternal()` method is invoked on the object to restore the object's state form the data stream.

▾ SonarQube rule
[Bad practice - Class is Externalizable but doesn't define a void constructor]

Methods

Don't write non-serializable object to ObjectOutput

In case of custom serialization, don't write non-serializable objects to `java.io.ObjectOutput`. Instead, replace such objects by a custom representation of serializable objects during serialization and deserialization.

▾ Reasoning
Writing non-serializable objects to ObjectOutput leads to a `java.io.IOException`.

▾ SonarQube rule
[Dodgy - Non serializable object written to ObjectOutput]

readResolve() must be protected

The `readResolve()` method must be declared as `protected`.

▾ Reasoning
Deserialization ignores the access modifier of the `readResolve()` method. However, if the method is declared as `private` or package private (without access modifier), it won't be inherited by subclasses.

Declaring the method as `public` would open unnecessary accessibility outside deserialization.

▾ SonarQube rule
[Dodgy - private readResolve method not inherited by subclasses]

readResolve() must not be static

The `readResolve()` method must not be declared as `static`.

▾ Reasoning
In order for the `readResolve()` method to be recognized by the Java default serialization mechanism, it must not be declared as `static`.

▾ SonarQube rule
[Correctness - The readResolve method must not be declared as a static method]

readResolve() must return Object

Don't replace the return type of the `readResolve()` method by a subtype of `java.lang.Object`.

▾ Reasoning

In order for the `readResolve()` method to be recognized by the Java default serialization mechanism, it must be declared to have a return type of `java.lang.Object`.

▾ SonarQube rule
Bad practice - The readResolve method must be declared with a return type of Object

readObject() and writeObject() must be private

The `readObject()` and `writeObject()` methods must be declared as `private`.

▾ Reasoning
In order for the `readObject()` and `writeObject()` methods to be recognized by the Java default serialization mechanism, they must be declared as `private`.

▾ SonarQube rule
Correctness - Method must be private in order for serialization to work

Coding style

This chapter discusses rules and best practices on how to implement Java code in a way which best supports stability, readability and maintainability.

General coding style

Rules and best practices related to annotations, casts, imports, internationalization and general programming style.

Arrays and collections

Rules and best practices related to using arrays and collections as well as to using and implementing iterators.

Assignments

Rules and best practices related to assignments and operator usage.

Comparisons

Rules and best practices related to comparisons and comparators.

Conditionals

Rules and best practices related to the the ? operator, if ... else and switch statements.

Constructors and initialization

Rules and best practices related to constructors, class and object initialization.

Exceptions

General rules and best practices on how to throw and handle exceptions.

GUI programming

General rules and best practices related to implementing native Java user interfaces with AWT and Swing.

Logging

Rules and best practices on how to setup and use logging.

Loops

Rules and best practices related to recursion, for and while loops.

Methods

Rules and best practices related to method usage, returns and reflection.

Null

Rules and best practices on how to safely deal with object references that can be null.

Superfluous code

Rules on code which is either unused or has no effect.

General coding style

This section discusses rules and best practices which are related to annotations, casts, imports, internationalization and general programming style.

- *Express operator precedence by braces in arithmetic expressions*

Annotations

Annotate deprecated elements as @Deprecated

Mark fields, methods and types that are no longer intended to be used with a `@Deprecated` annotation. If an element is annotated as `@Deprecated`, its Javadoc must contain a `@deprecated` tag with an explanation, and vice versa.

Reasoning
Enables users of the code to prepare for avoiding or removing code that is no longer intended to be used. This reduces refactoring efforts when a deprecated element disappears in a future software version.

SonarQube rule
Missing Deprecated

Annotate overridden methods as @Override

Overridden methods must be marked with an override annotation. This also applies to methods declared in an interface.

Reasoning
Makes it is easier to detect errors when the superclass method is renamed or removed from the class.

SonarQube rule
Missing Override

Use compact annotation style

Use compact style for annotations with just a single element named `value`.

Reasoning
The compact style syntax as in `@MyAnnotation ("MyValue")` are shorter and easier to read than the equivalent array style notation as in `@MyAnnotation (value = "MyValue")`.

SonarQube rule
Annotation Use Style

Annotate implementation of annotations with @Retention(RetentionPolicy.RUNTIME)

Annotate the implementation of annotations with `@Retention(RetentionPolicy.RUNTIME)`only if needed at runtime.

Reasoning
Only annotations which have themselves been annotated with `@Retention(RetentionPolicy.RUNTIME)` can be evaluated at runtime using reflection. Otherwise, the `isAnnotationPresent(...)` method of `java.lang.reflect.AnnotatedElement` will return `false` for the annotation.

SonarQube rule
Correctness - Can't use reflection to check for presence of annotation without runtime retention

Don't use modifiers in annotation field declarations

Don't use modifiers such as `public` or `final` in declarations of annotation fields.

Reasoning
Fields of annotations are automatically `public` and `final`.

SonarQube rule
Redundant Modifier

Blocks

Don't nest blocks

Don't nest code blocks.

Reasoning
Nested blocks decrease code readability and are often leftovers from debugging. Furthermore, local variables declared in a nested block may obscure local variables with the same name that have been declared in outer blocks, which is highly confusing.

SonarQube rule
Avoid Nested Blocks

Declare local variables that never get their values changed as final

Declare a local variable as `final` if it is only assigned a value once. This is the case if the value doesn't get replaced later in the code.

Reasoning
Enhances code readability and facilitates maintenance, as one can be sure that an assignment of a value to the variable cannot be followed by another assignment later in the code.

SonarQube rule
Final Local Variable

Casts

Check type before cast

Guard typecasts by an `instanceof` check for type compatibility.

Reasoning
Enhances type safety of the code and makes handling of type incompatibilities easier.

SonarQube rules

Correctness - Impossible cast

Correctness - Impossible downcast

Dodgy - Unchecked/unconfirmed cast

Unchecked/unconfirmed cast of return value from method

Avoid excessive usage of cascading instanceof checks

Avoid using multiple `instanceof` checks to differentiate code based on the specific type of an argument.

Reasoning

The usage of inheritance or design patterns instead of multiple `instanceof` checks improves readability. OOP provides possibilities to overcome 'manual' type inspection, in particular polymorphy and method overloading.

Avoid casting collection type

Don't cast collection types, e.g. from `Collection` to an abstract collection type such as `List`, `Map` or `Set`, or from an abstract type to a concrete subtype such as `ArrayList` or `HashSet`. If a more specific type is needed, the collection should be specified as such.

Reasoning

Downcasts might fail at runtime and become instable if the type of the used object depends on external code. Furthermore, this breaks the exchangeability of the Collection implementation.

SonarQube rules

Dodgy - Questionable cast to abstract collection

Dodgy - Questionable cast to concrete collection

Avoid raw types

When using generics avoid raw types where possible.

Reasoning

If used with `java.lang.Object`, objects returned from the generic type need to be type casted to their concrete type before a method that is not provided by java.lang.Object can be invoked.

SonarQube rule

Bad practice - Unchecked type in generic call

Imports

Avoid unnecessary fully qualified class names

Use imports instead of fully qualified class names, unless unavoidable and for types which are only required by the Javadoc, but not by the Java code.

Using fully qualified class names cannot be avoided

- in the context of reflection;
- in case of invocations of methods which expect a fully qualified class name argument, e.g. `Class.forName(String)`;
- in situations where you have to use multiple classes with the same name from different packages.

In all other situations, import the type and just use its name.

Reasoning
Increased code readability, since dependencies to other types can be directly derived from the `import` statements.

See *Use fully qualified type names in Javadoc links* on how to reference types in Javadoc.

Avoid static imports

Don't use `import static` statements.

Reasoning
Static imports enable the importing code to omit type prefixes for static fields and methods of the imported type. Such code appears to access fields or methods of the importing type, but doesn't. The benefit of shorter statements is overcompensated by the loss of understandability of the code.

SonarQube rule
Avoid Static Import

Avoid star imports

Instead of using wildcards for imports as in `import com.bmw.myapp.*;`, use fully qualified type names in imports. (Note: most IDEs can generate those imports when yet unknown types are used.).

Reasoning
Star imports import all types of a package. Thus, it is no longer obvious which types are really used in the importing type's code, whereas dedicated imports per type clearly document type dependencies. For types used in the code, the containing package becomes intransparent, which is an unnecessary for code maintenance.

SonarQube rule
Avoid Star Import

Don't import unused types

Remove imports for types which are not used in the Java code. (Note: most IDEs can automatically remove unused imports.)

Reasoning
Keeps the code clean from unused code and facilitates the detection of dependencies to other code. Since documentation often contains cyclic references between documentation of related types, such cycles are not injected into the Java code.

Furthermore, Javadoc generation remains independent of the code. Especially, if a type is removed from the imports because it is no longer required by the Java code, Javadoc may still reference the removed type and will still be correctly generated.

See *Use fully qualified type names in Javadoc links* on how to reference types in Javadoc.

SonarQube rule
Unused imports (Checkstyle)

Avoid unnecessary imports

Avoid redundant imports. Don't import

- the same type multiple times;
- types from the `java.lang package`;
- types from the same package.

Reasoning
Such import statements are useless and only unnecessarily blow up code.

SonarQube rule
Redundant import

Programming style

Avoid break and continue to labels

Never use labels, e.g. `LABEL:`, and jumps to labels such as `break LABEL;` or `continue LABEL;` in Java code. Rather invest in clearly structured control blocks. Especially, avoid `switch` statements that contain control flow labels from an outer scope element like a loop.

Reasoning
Labels encourage a goto-ish programming style, which significantly decreases code understandability. If used in combination with case labels, programmers easily mix up case and non-case labels.

SonarQube rule
Non Case Label In Switch Statement

Static access requires class name

Qualify each access to a static field or static method from another class with the class name, rather than using an instance of that class. Also use the class name qualifier when the static element is inherited from a superclass. Omit the class name qualifier when a static element is accessed from its owning class.

Reasoning
Static fields and methods do not require an instance of their enclosing class to be present, but can be accessed directly. Using an instance reference is only confusing, as it obscures that the accessed element does not belong to an instance. Furthermore, if an instance is created only to access the static element, an unnecessary object must be allocated and garbage collected.

SonarQube rules
Bad practice - Needless instantiation of class that only supplies static methods

Correctness - Unneeded use of currentThread() call, to call interrupted()

Instance-level access requires this

Prefix each access to instance fields and methods with `this`. When accessing instance fields and methods from a non-static inner class, the prefix is `<OuterClass>.this`, where `<OuterClass>` is a placeholder for the name of the outer class. Note that this rule only applies to calls from named or anonymous *non-static* inner classes, but not to calls from *static* inner classes.

Reasoning

Reliably prevents unintended access to static methods, parameters and local variables.

An instance of a *non-static* inner class holds an implicit reference to the instance of its outer class. It has access to all methods of the outer and of the inner class. Prefixing calls to the outer instance makes obvious on which object the method is invoked, and it prevents unintended invocations of the wrong method if the outer and the inner class implement methods with the same signature. In such a situation, `<OuterClass>.this.method()` invokes the method on the outer class instance, whereas `this.method()` invokes the method on the inner class instance. The distinction between such calls is even more helpful if the outer class provides a method that is also inherited from a superclass: Without the `this` prefix, It is almost impossible to read such a method call as a call of the inherited method, and not of the outer class's method as it appears.

In case of a *static* inner class using the outer class name as a prefix is not an issue as the inner class's instance doesn't hold an implicit reference to an instance of the outer class. References to an outer class instance can only be explicit, so the outer class's method can be invoked on the explicit reference.

SonarQube rules

Dodgy - Ambiguous invocation of either an inherited or outer method

Require This

Express operator precedence by braces in arithmetic expressions

Intensively use braces in arithmetic expressions that would otherwise depend on operator precedence.

Reasoning

Enhances code readability and prevent unintended side effects and logical errors.

SonarQube rule

Correctness - Integer multiply of result of integer remainder

Arrays and collections

This section discusses rules and best practices which are related to using arrays and collections as well as to using and implementing iterators.

Using arrays and collections

Use collection interface methods over concrete collection implementation

Access and manipulate collections using the methods defined in the `java.util.Collection` interface rather than using the concrete methods before Java.1.2.

Reasoning
The collection interface methods decouple the collection from its concrete implementation.

Use isEmpty() to check if collection has elements

Use `Collection.isEmpty()` to check if a collection contains elements, rather than comparing its size against zero.

Reasoning
The `isEmpty()` method is easier to understand than using `size() == 0`.

SonarQube rule
Use Collection Is Empty

Use Collection.toArray() with array argument to convert collections to arrays

To obtain an array of the elements contained in a collection for the declared generic type of the collection, use the collection's `toArray(<T>[])` method rather than the no argument `toArray()` method, where <*T*> is the collection's generic element type.

Reasoning
For most collections, `toArray()` returns an array of `Object` and not an array of <*T*>, which doesn't allow a later type cast to <*T*>`[]`, whereas `toArray(<T>[])` returns an array of <*T*>.

SonarQube rule
Correctness - Impossible downcast of toArray() result

Don't pass array of primitives for variable length arguments of type Object

Don't pass an array of primitives for a method parameter that accepts a variable number of `java.lang.Object` arguments.

Reasoning
In the Java programming language, arrays are objects. Thus, the method invocation creates an array of Object, which contains the array of primitives as its one and only member, rather than autoboxing each array entry..

SonarQube rule
Correctness - Primitive array passed to function expecting a variable number of object arguments

Reuse value returned from ConcurrentMap.putIfAbsent()

Use the object returned from `java.util.concurrent.ConcurrentMap.putIfAbsent()` for further processing, rather than the value passed to the method.

Reasoning
If the ConcurrentMap already contains an object for the given key, this value remains in the map and is returned. The value passed to the map is only added to the collection and returned by `putIfAbsent()`, if the map did not contain a value associated with the given key prior to the method invocation. Ignoring the return value poses the risk of using an object that isn't stored in the map.

SonarQube rule
Correctness - Return value of putIfAbsent ignored, value passed to putIfAbsent reused

Use addAll() rather than iteration

To add all elements of one collection to another collection, call `addAll()` on the target collection with the source collection passed as the argument rather than iteratively adding every single element of the source collection. If the source is an array, use `java.util.Arrays.asList()` to coerce the array into a collection.

Reasoning
On some collection types (like ArrayList) calling `addAll()` will additionally offer better performance than iteration. Also for thread safe target collections `addAll()` is synchronized by default.

SonarQube rule
Use Arrays As List

Use toArray() to create array from collection

To create an array from collection, use the collection's `toArray()` method.

Reasoning
It is shorter, easier to understand code to create an array from a collection using the `toArray()` method rather than iterating over the collection and filling the array manually.

Iterator implementation

Use iterators over indices to iterate over lists

Don't iterate over a list using an incrementing index only used for List.get(index). Use an Enhanced for statement instead.

Reasoning
For many `java.util.List` implementations, iterators and two-argument `for` loops are considerably faster than an incrementing index variable. The code becomes shorter and is easier to understand as the index must not be defined. Furthermore, the list can easily be exchanged by another collection type.

Consider using Iterator instead of Enumeration

Use iterator objects to iterate over collections instead of the older enumeration objects.

Reasoning

Iterator takes the place of Enumeration in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics
- Method names have been improved

SonarQube rule

Replace Enumeration With Iterator

Iterator.next() must throw NoSuchElementException when depleted

An implementation of the `java.util.Iterator.next()` method must throw a `java.util.NoSuchElementException` after the last element has been reached, rather than return `null`.

Reasoning

This the expected behavior as defined by the `java.util.Iterator` interface.

SonarQube rule

Bad practice - Iterator next() method can't throw NoSuchElementException

Don't modify iterator in hasNext()

Don't modify iterator in the `hasNext()` method of an iterator. Especially, don't call `next()`.

Reasoning

The `next()` method modifies the iterator, whereas `hasNext()` is expected to not modify the iterator.

SonarQube rule

Correctness - hasNext method invokes next

Don't reuse Map.Entry objects in iterators

Iterator implementations may not retain the key/element mappings returned from a map's `entrySet()` method for later reuse.

Reasoning

`Map.Entry` objects are only guaranteed to contain valid data during iteration. If `Map.Entry` objects are kept, e.g. in a separate `Set`, then their contents is not defined – they may be empty, or all of them may contain the values of the last entry objects, as they may be reused during iteration.

This is a cause for unexpected behavior if a map `m` returns an iterator for its entry set and `c.addAll(m.entrySet())` is called on a collection `c`.

SonarQube rule

Don't reuse entry objects in iterators

Assignments

This section discusses rules and best practices which are related to assignments and operator usage.

Assignments to static fields

Static fields must be immutable

Once being initialized, a static field's value should not be re-assigned. Declare static fields as `final` to avoid such assignments.

Reasoning

Using static fields as a global variable may lead to unexpected behaviour, in particular e.g. for multi-threaded applications.

SonarQube rule

Assignment To Non Final Static

Don't write to static field from instance method

Don't write to static field from instance method. If required at all, rather invoke a static method which manipulates the field. Consider concurrency issues discussed in Thread-safety when the code might be used in a multi-threaded environment.

Reasoning

Instance methods are only intended read static fields and to manipulate object members, but not to manipulate static fields.

In a multi-threaded context, manipulating static fields can cause unexpected and wrong results if applied without proper synchronization, since an invocation of a method which manipulates a static field can modify the field's value between a write and a subsequent read access to the field in another thread.

SonarQube rule

Dodgy - Write to static field from instance method

Boolean assignments

Assign result of boolean expression directly

Directly assign the result of boolean expressions to boolean variables, instead of using `if...else` conditionals or the ? operator.

Reasoning

A direct boolean assignment reduces computational overhead and expresses a simple thing in a simple way.

Sample

Boolean assignment

```java
public void foo(final Collection<ElementType> collection,
                final ElementType             element) {
    ...
    boolean elementFound;

    // Replace a boolean assignment like ...
    if (collection.contains(element)) {
        elementFound = true;
    } else {
        elementFound = false;
    }

    // ... or like ...
    elementFound = collection.contains(element)
                   ? true
                   : false;

    // ... by a less expensive and clearer direct assignment:
    elementFound = collection.contains(element);

    ...
}
```

Obscured assignments

Avoid inner assignments

Use inner assignments sparsely and only if their semantics can easily be captured.

Reasoning

Inner assignments often makes code hard to read and understand. Still, on some occasions it may even increase readability (see sample).

Sample

This is a sample of an inner assignment which enhances code readability:

Inner assignment

```java
import java.io.BufferedReader;
...
private void foo(final BufferedReader reader) {
    String line;
    while((line = reader.readLine()) != null) {
       ...
    }
}
...
```

Without the inner assignment, there would have to be a statement `l_line = reader.readLine();` right before the loop and a second time right before the end of the loop.

SonarQube rule

Correctness - Useless assignment in return statement

Don't assign booleans in conditions

Don't assign a value to a boolean field inside a boolean condition.

Reasoning

Boolean inner assignments in conditions are an easily overlooked cause for logical errors, especially if used with conditionals.

Sample

Inner boolean assignment error

```java
// This inner boolean assignment causes an infinite loop ...
while (booleanValue = true) {
    ...
}
```

```java
// ... whereas this is a straight forward boolean condition:
while (booleanValue == true) {
    ...
}
```

SonarQube rule

Correctness - Method assigns boolean literal in boolean expression

Operator usage in assignments

Don't use increment or decrement operators in self assignments

Don't use unary increment or decrement operators in self assignments, like `i = i++;`. Rather only use only the unary operator `i++;` or its long form `i = i + 1;`.

Reasoning

Mixing unary increment/decrement operators leads to unexpected results when using post-increment or post-decrement operators. This is highly confusing and makes code hard to understand, as it suggests an increment/decrement, but has no such effect.

Sample

Self-assignment error

```java
// This self assignment is vacuous ...
i = i++; // i remains unmodified, as it is assigned before its value is incremented.
```

```java
// ... whereas these assignments will increment i by 1:
i++;
i = i + 1;
```

SonarQube rule

Correctness - Overwritten increment

Avoid multiple unary operators

Avoid using multiple unary operators in the same expression. Rather simplify the expression to only use a single or no unary operator per expression.

Reasoning

Multiple unary operators for the same operand are highly confusing.

Sample

Confusing multiple unary operators

```
// The subsequent combinations of multiple unary operators ...
int i = +-+1;         // Unnecessary signs.
boolean b = ! ! true; // Double negation.
int j = -~j;          // Negative unary complement of j.
```

```
// ... can be simplified to better readable:
int i = -1;
boolean b = true;
int j = j+1;
```

SonarQube rule

Avoid Multiple Unary Operators

Comparisons

This section discusses rules and best practices which are related to comparisons and comparators.

General comparison rules

Call methods on guaranted non-null objects

When calling methods using non `null` constants or non `null` literals, place the constant or literal on the left side of the method call, rather than passing it as an argument to the other value involved in the comparison.

Reasoning

Calling on a known non `null` value will guarantee that a `NullPointerException` cannot be thrown, whereas calling a method on an unknown value requires a separate `null` check for safe invocation. Examples are `equals()`, `equalsIgnoreCase()` or `Comparable.compareTo()`

SonarQube rule

Equals Avoid Null

Boolean comparisons

Don't use boolean literals in boolean comparisons

Use boolean expressions directly rather than comparing them first with boolean literals.

Replace	with
`(<boolean expression> == true)`	`<boolean expression>`
`(<boolean expression> == false)`	`(! <boolean expression>)`

Reasoning

Comparing boolean values with boolean literals is unnecessary and only decreases code readability and increases computational overhead.

SonarQube rule

Avoid unnecessary comparisons in boolean expressions

Use short-circuit boolean operators

Only use conditional-AND/OR operators (&& or ||) rather than strict operators (& and |) in boolean expressions.

In cases where the execution of the expression right of the operator is required independently of the evaluation of the expression left of the operator, extract the right statement as a separate statement before the boolean expression and make the boolean expression evaluate the extracted statement's boolean result, using a conditional-AND/OR operator.

Reasoning

In contrast to conditional-AND/OR operators (&& or ||), operators & and | cause both sides of the expression to be always evaluated, even when the result can be inferred from the left-hand evaluation result. This is less efficient and can lead to runtime errors in many situations.

Samples

Non short-circuit boolean operator failure

```java
// The following code will throw a NullPointerException for a being null ...
if ((a == null) | a.isEmpty()) {
    ...
}
```

```java
// ... whereas this works as expected:
if ((a == null) || a.isEmpty()) {
    ...
}
```

Elimination of non short-circuit boolean operator

```java
public void foo(final boolean b,
                final MyClass anObj) {

    // The following non-short-circuit sample ...
    if (b & anObj.booleanMethod()) { // booleanMethod is also called if b == false.
        ...
    }

    // ... is equivalent to:
    boolean retVal = anObj.booleanMethod(); // Assures that booleanMethod is called.
    if (b && retVal) {
        ...
    }
}
```

SonarQube rules

Dodgy - Potentially dangerous use of non-short-circuit logic

Dodgy - Questionable use of non-short-circuit logic

compare() / compareTo() methods

Don't check for specific values returned by compare()/compareTo()

Only compare the result of `java.lang.Comparable.compareTo()` and `java.util.Comparator.compare()` through its sign, but not by the specific returned value.

Reasoning

Only the sign of the result of a `compare()` / `compareTo()` method call is relevant and should be considered for comparisons. Explicitly checking for expected return values of `-1`, `0` and `1` may fail as some implementations return other values like `Integer.MIN_VALUE` / `Integer.MAX_VALUE` instead of `-1` / `1`.

SonarQube rule

Code checks for specific values returned by compareTo

Don't negate result of compare()/compareTo()

Don't invert the sign of the result of `java.lang.Comparable.compareTo()` and `java.util.Comparator.compare()`. Reverse the order of the operands instead.

Reasoning

Negating the result may yield incorrect values when the method returns `Integer.MIN_VALUE` instead of `-1`, as this value cannot be negated correctly.

SonarQube rule

Negating the result of compareTo()/compare()

compare()/compareTo() must return -1, 0, 1

Methods `java.lang.Comparable.compareTo()` and `java.util.Comparator.compare()` must return -1, 0, 1 instead of arbitrary numbers. Every implementation must be able to return all three of the above values.

Reasoning

Replacing `-1` or `1` by other values can yield comparison errors in code that invokes the methods.

Implementations that do not return all of the three values fundamentally break the semantics of `compare()` / `compareTo()`:

- When `a.compareTo(b)` returns `1`, the `b.compareTo(a)` must return `-1` and vice versa.

SonarQube rules

compareTo()/compare() returns Integer.MIN_VALUE

Conditionals

This section discusses rules and best practices which are related to the the ? operator, `if ... else` and `switch` statements.

General rules for conditionals

Don't ignore evaluation result of conditionals

Don't use an `if` or `switch` clause if the evaluation result is ignored and all branches contain semantically identical code.

Reasoning

Program code that ignores the result of a conditions indicates either a superfluous conditions (dead code) or a logical program error.

SonarQube rules

Correctness - Useless control flow to next line

Dodgy - Method uses the same code for two branches

Dodgy - Method uses the same code for two switch clauses

Avoid unconditional conditionals

Don't use boolean expressions that always yield `true` or `false` in the condition of an `if` statement or in inline conditionals (? operator).

Reasoning

When such an expression is used in an `if` statement or an inline conditional, the `if` clause / inline conditional is useless and indicates a programming error.

SonarQube rule

Unconditional If Statement

Conditional operator

Use inline conditionals for assignments which depend on conditions

The ternary ? operator, aka 'inline conditional', may be used in situations where a boolean condition determines which value is assigned to a field or variable.

Reasoning

The ternary operator makes codes shorter and more precise, as independently of the condition result, the value is assigned to the same field or variable. An `if...then...else...` statement would allow an assignment to another field in the `if` clause than in the `else` clause. Thus, the ternary operator is more precise. Besides, the code is shorter.

Sample

Ternary operator

```
// The following code ...
if (myParam == null) {
    myVariable = new MyClass();
} else {
    myVariable = myParam;
}
```

```
// ... can be expressed shorter as:
myVariable = (myParam == null)
                ? new MyClass()
                : myParam;
```

Don't use ? operator with operands of different boxed types

Don't use the question mark operator (?) if the operands have different boxed types.

Reasoning

If a wrapped primitive value is unboxed and converted to another primitive type as part of the evaluation of a conditional ternary operator (the `b ? e1 : e2` operator), the semantics of Java mandate that if `e1` and `e2` are wrapped numeric values, the values are unboxed and converted/coerced to their common type (e.g, if `e1` is of type `Integer` and `e2` is of type `Float`, then `e1` is unboxed, converted to a floating point value, and boxed. See JLS Section 15.25.

SonarQube rule

Correctness - Primitive value is unboxed and coerced for ternary operator

switch statements

Remove empty switch statements

Empty `switch` statements must be removed.

Reasoning

Empty `switch` statements are just dead code that needs to be maintained.

SonarQube rule

[Empty Switch Statements]

Mark fall-through in switch statements

`case` groups in `switch` statements that contain Java code, but lack a `break`, `return`, `throw` or `continue` statement must document that this fall-through behavior is intended by inserting a `Fallthru` or `Fallthrough` comment.

Reasoning

Unintended fall-through behavior in `switch` statements is a main cause for program errors. The rule enforces to document that the fall-through is intended and no error, which facilitates detecting coding errors and enhances understandability of the code.

Sample

Intended fall-through

```java
switch (color) {
    case BLACK:
    case BLUE:
        doSomething();
        // Fallthru
    case GREEN:
        doSomethingElse();
        break;
    default:
        doSomeDefaultHandling();
}
```

SonarQube rule

[Fall Through]

Watch out for unintentional fall-through

In case of a fall-through, don't assign a value to the same field or variable in multiple case blocks that are involved in the fall-through without using the field/variable between the assignments.

Reasoning

The assignment is either useless or, which is more likely, a `break` or `return` statement has been forgotten at the end of the previous `case` branch.

Sample

Unintentional fall-through

```java
String myValue = "zero";

switch (color) {
    case BLACK:
        myValue = "black";
        // Unintended fall-through!!! In this case, myValue will be set to "blue" in the next label.
    case BLUE:
        myValue = "blue";
        break;
    case GREEN:
        myValue = "green";
        break;
    default:
        myValue = "white";
}
```

SonarQube rules

Dead store due to switch statement fall through

Dead store due to switch statement fall through to throw

Default must be last label in switch block

A `default:` branch in a switch statement must be the last `case` branch.

If a value is found that is not covered by the `case` branches and that cannot be processed correctly, the `default:` branch must take care of the correct processing.

Reasoning

Detects values that are not handled by the `case` branches and enables implementing a controlled processing of such situations. This is especially important for `switch` statements on an `enum` value: If a new value is added to the `enum`, the value at least gets detected and handled at runtime rather than being ignored, which could lead to an application error that might be hard to debug.

Placing the `default:` branch after all `case` branches enhances code readability and discourages from constructing hard to follow "fall through" statements.

SonarQube rule

Default Comes Last

Missing Switch Default

Constructors and initialization

This section discusses rules and best practices which are related to constructors, class and object initialization.

- Class initialization
  - *Don't call subclass during initialization of superclass*
  - *Ensure initialization of Class objects*
- Constructor implementation
  - *Call superclass constructor in non-delegating constructors*
  - *Don't call non-final methods in constructors*
  - *Don't use side-effect constructors*
  - *Don't use uninitialized fields or variables*
  - *Don't initialize non-null members with null*
  - *Don't create instance before initialization of all static final fields*

Class initialization

Don't call subclass during initialization of superclass

Never use a subclass during the initialization of a superclass.

Reasoning

The subclass will not yet be initialized, since the superclass is always initialized before the subclass. Operations like assigning an instance of a subclass to a static field of the superclass will fail. Furthermore, such code creates a circular dependency between the superclass and its subclass, which is never acceptable.

SonarQube rule

Bad practice - Superclass uses subclass during initialization

Ensure initialization of Class objects

Invoke method on class object to ensure that the class's static initializer has been executed.

Reasoning

Up to Java 1.4, such a reference would force the static initializer of the referenced class to be executed. As of Java 5, this is no longer the case: The static initializer is only executed on the first method invocation on the Class object.

Sample

Class initialization

```java
import com.bmw.MyClass;

// Up to Java 1.4, this would initialize the class ...
Class clazz = MyClass.class;

// ... whereas as of Java 5, this is required:
Class<MyClass> clazz = MyClass.class;
Class.forName(clazz.getName()); // Invoke a method on the Class object.
```

> Note that any invocation of a static method or constructor ensures that the called class has been initialized. Thus, if the intention was only to initialize the class, the assignment can be removed.

SonarQube rule

Correctness - Dead store of class literal

Constructor implementation

Call superclass constructor in non-delegating constructors

The first statement of a constructor must be a call to another constructor of the same class or to a superclass constructor.

Reasoning

Explicitly calling another constructor makes the construction process obvious and, in case of a call to the superclass's default constructor, documents that this behavior is intended, rather than relying on the Java compiler to implicitly create the same call when no constructor call is present.

SonarQube rule
Call Super In Constructor

Don't call non-final methods in constructors

Constructors must not call non-final methods.

Reasoning
Since non-final methods can be overridden by subclasses, a subclass implementation would be executed against an object that has not been initialized at subclass level.

SonarQube rule
Constructor Calls Overridable Method

Don't use side-effect constructors

A constructor may only modify values inside its enclosing object and enclosing class. Use separate methods rather than constructors for modifications of other objects or the application's overall state.

Reasoning
A constructor is only expected to return an initialized object. If the constructor performs additional logic, the constructor is a side-effect constructor. Programmers are tempted to use such a constructor just for the side-effect code to be executed, without holding a reference to the created object. Such code is hard to understand and, as it looks like dead code, it is likely to be removed during maintenance, which leads to errors if the side-effect code is still required.

Don't use uninitialized fields or variables

Don't use fields or variables before they have been explicitly initialized. Don't rely on default initializations by the VM.

Reasoning
This is a typical error resulting from a wrong initialization order. This aspect can be tricky if a field is read by a method that is used by a constructor. Relying on the VM's default initialization can lead to wrong program code if a used field has mistakenly been forgotten to be initialized or if is being used prior to initialization, e.g. for a field that has been added at a later point of time, or if the initialization order is wrong.

SonarQube rules
Correctness - Uninitialized read of field in constructor

Correctness - Uninitialized read of field method called from constructor of superclass

Don't initialize non-null members with null

Assure that fields which are defined or annotated to be non-null are initialized with a non-null value.

Reasoning
For such fields, code which accesses the fields' values must rely on the fact that the value can be de-referenced without risk. Such code typically doesn't implement precautions like null checks, which can lead to `NullPointerExceptions` being unexpectedly thrown if the field's contract is violated.

▾ SonarQube rule
Nonnull field is not initialized

Don't create instance before initialization of all static final fields

In a static initializer, don't create an instance of a class before all `static final` fields that don't hold an instance of the class have been fully initialized.

▾ Reasoning
A constructor of the class or a method called by a constructor may rely on a constant defined in the class, either now or as a result of a later code modification. If the constant has not been initialized before the instance is created, it might not be correctly initialized. To avoid such risks, create the static instances of the class at the end of the static initializer.

▾ SonarQube rule
Bad practice - Static initializer creates instance before all static final fields assigned

Exceptions

This section discusses general rules and best practices on how to throw and handle exceptions.

General exception policies

Don't use exceptions as flow control

Don't abuse exceptions as another means of control flow. Either add the necessary validation or use an alternate control structure.

▾ Reasoning
Exceptions denote special and exceptional circumstances in a program that must be handled to avoid program

failure, or can't be handled and need to be logged to document system failure. Using them as a form of flow control obscures real issues in the program.

▾ SonarQube rule
Exception As Flow Control

Remove empty try blocks

Remove `try` blocks that contain no statements.

▾ Reasoning
Superfluous code that just reduces readability. Typically, such code is a leftover from code refactoring.

▾ SonarQube rule
Empty Block

Throwing exceptions

Created exceptions must be thrown

Don't create an exception (or error) object without throwing it.

▾ Reasoning
A created exception that is not be thrown has no impact. For example, something like

```
if (x < 0)
   new IllegalArgumentException("x must be nonnegative");
```

It was probably the intent to throw the created exception:

```
if (x < 0)
   throw new IllegalArgumentException("x must be nonnegative");
```

▾ SonarQube rule
Correctness - Exception created and dropped rather than thrown

Only throw subtypes of Exception and RuntimeException

Thrown exceptions must be subtypes of `java.lang.Exception` and `java.lang.RuntimeException`. Don't throw direct instances of these exceptions or a `java.lang.Error` or `java.lang.Throwable`.

▾ Reasoning
Throwing direct instances of `java.lang.Exception` and `java.lang.RuntimeException` would force exception handlers to catch these exception types, which prevents any specific exception handling. This is likely to result in processing and discarding exceptions that are not intended to be handled by the exception handler, e.g. exceptions thrown deep in the runtime.

▾ SonarQube rule
Illegal Throws

Don't throw NullPointerException

Don't throw `java.lang.NullPointerException`.

Reasoning
Throwing a NPE from within application code is considered bad style. It is better to check for null and then throw a meaningful exception, may it be technical or application specific.

SonarQube rule
Illegal Throws

Pass a meaningful message or error id to exception constructors

Pass an error message or a unique error identifier when constructing an exception.

Reasoning
Aids in identifying why the exception was thrown in exception handlers and facilitates the generation of meaningful error messages, e.g. for error logs or displayed to users.

Holding error messages or ids in a central class that holds global constants make it easier to ensure that the messages or ids passed to the constructor are the same that are expected by exception handlers or other code that processes the exception.

catch clause

Avoid empty catch blocks

Avoid empty `catch` blocks. If an exception needs to be caught, but doesn't require exception handling, indicate this by an explanatory comment.

Reasoning
Swallowing exceptions and continue processing puts applications into an unstable state. However, there are rare conditions where catching an exception without handling can be legitimate. This is the case for exception handlers within a `try` statement that is nested in a `finally` block, as `finally` blocks may not throw exceptions, of if the thrown exception has been expected and does not indicate an error, e.g. when testing if a file exists. Such exceptions often don't require special processing, but still need to be caught.

The comment indicates that the exception wasn't just swallowed to silence a problem, but by intention and for a good reason.

SonarQube rule
Empty Catch Block

Only catch exceptions that can be handled

Only catch exceptions that can be handled, e.g. to do some internal processing, to recover from the exception or to throw a more meaningful exception. If nothing of this is applicable, don't catch the exception.

Just catching an exception to do nothing more than rethrowing it or throwing an exception of the same type doesn't justify an exception handler.

Reasoning

It's useless to catch all exceptions that are possibly thrown by an invoked method if you can't do anything useful with the exception. Just swallowing or rethrowing the exception is not a useful reaction on the exception to be thrown.

SonarQube rules

Avoid Rethrowing Exception

Strict Exception - Avoid throwing new instance of same exception

Avoid Instanceof Checks In Catch Clause

Don't catch exceptions that cannot be handled

Don't catch exceptions that cannot be handled. The minimum handling is throwing an exception of another, more precise type. In this case, pass the caught exception to the `cause` parameter of the constructor of the thrown exception.

Reasoning

Swallowing exceptions complicates dubbing and thus has an impact on maintainability and operations.

Pass the caught exception to the thrown exception in order to preserve information on the nature and location of the error that caused the exception to be thrown.

SonarQube rule

Avoid Losing Exception Information

Don't catch generic exceptions

In normal program code, don't catch generic exceptions such as `java.lang.Error`, `java.lang.Exception`, `java.lang.RuntimeException` or `java.lang.Throwable`. Catching such exceptions is only acceptable in an application's overall exception handler, or in test code that is not deployed into production.

Reasoning

Catching generic `Throwable` types would also swallow all errors that are thrown by the VM to signal an inconsistent or critical system state.

SonarQube rule

Illegal Catch

Not fully recovering exception handler must throw exception

Error handlers that cannot recover from the caught exception must throw an exception. Pass the caught exception to the `cause` parameter of the constructor for the exception to be thrown.

Try to throw a more meaningful exception that is documented in the method's API. If the API doesn't support such an exception to be thrown and cannot be extended to another exception type, throw a subtype of `java.lang.RuntimeException` that supports the semantics of the exception. If the exception to be thrown is of the same type as the caught exception, don't rethrow the caught exception, but create a new instance, still passing the original exception to the new exception's constructor.

Reasoning

Throwing an exception in case of incomplete recovery from another exception is essential to signal an error to calling methods.

Passing the original exception to the thrown exception preserves the stack trace and provides additional data on the exception's cause.

Throwing a more meaningful exception aids calling methods in correctly interpreting and handling the exception.

Wrapping an exception into a subtype of `java.lang.RuntimeException` can be the only means to ensure that an exception is thrown without violating the method's API. However, this should only be done if the method's API cannot be extended to a more specific and feasible exception type.

Using a direct instance of `java.lang.RuntimeException` is not a feasible alternative!

Re-throwing an exception would pass a wrong location of where the exception was thrown in its stack trace.

SonarQube rule

Preserve Stack Trace

finally clause

Remove empty finally blocks

Remove `finally` blocks that contain no statements.

Reasoning

Superfluous code that just reduces readability. Typically, such code is a leftover from code refactoring.

SonarQube rule

Empty Block

Use finally block to reliably release resources

Use the `finally` block to release resources obtained in the `try` block in a reliable way for Java versions < 7. Note that a `try` block doesn't require a `catch` block if a `finally` block is present.

In Java 7 the try-with-resources (ARM) construct was introduced to cope with that problem. See link for a detailed description to that topic.

Reasoning

The `finally` block is always executed before returning from the `try` statement, even if the code in the `try` block threw an exception. Thus, it is the ideal place to release resources like a lock that has been obtained in the `try` block, rather than having the `try` statement release the resource in case that its code does not run into an exception, and implementing one or more exception handlers to release the resource in exceptional cases. Furthermore, releasing resources in the `finally` block preserves the exception thrown in the `try` block or another exception handler.

Don't return from finally blocks

A finally block must not contain a `return` statements.

Reasoning

A return statement in a `finally` block would abnormally short-circuit normal method termination. Furthermore, exceptions thrown in the `try` of `catch` block would be discarded.

SonarQube rule

Return From Finally Block

GUI programming

This section discusses general rules and best practices which are related to implementing native Java user interfaces with AWT and Swing.

AWT/Swing programming

Assign name to GUI elements

When using a GUI element which inherits from `java.awt.Component`, assign an element name by calling the elements `setName(String)` method.

Reasoning

Assigning a name to GUI elements enables screen readers and GUI automation tools to correctly interpret the GUI elements.

Link GUI labels with the labeled fields

When using a `javax.swing.JLabel`, call `setLabelFor()` and pass the Component that specifies the labeled field as a parameter.

Reasoning

Linking the label to the field enables screen readers and GUI automation tools to correctly interpret the GUI elements.

Localize labels that are used in GUI components

When passing a label to a Swing component, assure that the label is localized through the use of a resource bundle rather than passing a non-localized string.

Reasoning

As the string will be shown to users, passing a localized string makes the GUI portable to other countries and languages.

Retrieve foreground and background colors from the operating system

Rather retrieve the colors for foregrounds and backgrounds from the operating system than using explicit values defined in the application.

▾ Reasoning
Using operating system color settings improves the application's native look and feel and can prevent difficulties for people with vision problems.

Don't use absolute coordinates for layout of GUI elements

Always pass a `java.awt.LayoutManager` for the `mgr` parameter of the `setLayout()` method of a `java.awt.Container`.

▾ Reasoning
If no `LayoutManager` is present, components will be laid out using absolute coordinates. This makes changes of font sizes etc. difficult since items will not reposition.

Use pack() to size a java.awt.Window

Rather use the `pack()` than the `setSize()` method to set the size of a `java.awt.Window`.

▾ Reasoning
The `pack()` method calculates the window size dynamically and encounters font size changes.

Custom adapters must correctly override event handling methods

When implementing a `java.awt.event.Adapter` that implements a listener defined in the `java.awt.event` or `javax.swing.event` package, ensure that the implementation overrides the Adapter's methods that handle events.

▾ Reasoning
Event handling methods which don't override Adapter methods will not be called at runtime.

▾ SonarQube rule
Correctness - Class overrides a method implemented in super class Adapter wrongly

JComponent should implement Accessible

A class that extends `javax.swing.JComponent` should also implement the `javax.accessibility.Accessible` interface.

▾ Reasoning
Otherwise the GUI will not be accessible and cannot be processed, e.g. by Braille lines or screen readers.

Logging

This section discusses rules and best practices on how to setup and use logging.

Logger setup

Don't log to hard-coded log target

Use dedicated logging through a logger. Don't use hard-coded log targets like `System.err` or `System.out` for log messages.

Reasoning

Loggers allow fine-grained filtering of log output, e.g. by severity or component, as well as a flexible configuration of the log target. Furthermore, loggers decouple the creation of potential log output from the logging configuration, whereas logging to `System.err`/`.out` hard codes logging in the application and doesn't allow an independent, fine grained log configuration.

Especially in application server environments, it is not guaranteed that the output of System.out or System.err is persisted.

SonarQube rule

System Println

Use proper logging framework

Use a framework that is approved by the BMW IT Blueprint. As of Dec. 2012, these are either Apache Log4J or Apache Commons Logging.

Reasoning

Powerful, easy to use and widely spread logging frameworks.

Define logger properly

Define the logger either

- as a `private` constant named `LOG` and with the surrounding class as parameter in each class file that is subject to logging, or
- as a `private final volatile` member field that is initialized with a logger passed to the constructor.

Reasoning

The logger instance can be declared as a static final field as it is immutable and thread-safe.

Using the class of the surrounding class as context for the logger enables fine tuning of the log behavior by configuration means. Holding a reference to the logger supports the scenario with a central logger instance being passed around. In this case, the scenario must be final and a passed into the logged class's constructors.

Samples

Apache Log4J static logger

```java
import org.apache.log4j.Logger;

public class MyClass {
    ...
    private static final Logger LOG = Logger.getLogger(MyClass.class);
    ...
    public void doSomething(final Object obj) {
        LOG.trace("Entering doSomething(Object)");
        ...
    }
}
```

Apache Commons static logger

```java
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class MyClass {
    ...
    private static final Log LOG = LogFactory.getLog(MyClass.class);
    ...
    public void doSomething(final Object obj) {
        LOG.trace("Entering doSomething(Object)");
        ...
    }
}
```

SonarQube rules

Proper Logger

Static logger name

Name the `static` logger constant `LOG` when implementing a single logger per class.

Reasoning

A common logger name enhances understandability of the code. See Logging on how to define and use a logger.

SonarQube rule

Proper Logger

Don't loose Java Logger configuration

Don't loose Java Logger configuration after garbage collection due to weak reference in OpenJDK.

Reasoning

The behavior of `java.util.logging.Logger` differs between OpenJDK and Oracle's SDK. Instead of using strong references, the OpenJDK Logger uses weak references internally. Thus, a Logger instance with a configuration attached may be garbage collected, and when being looked up after garbage collection, a new Logger instance with a default configuration will be created, which might well differ from the Logger instance's configuration as before the garbage collection.

Samples

Lost logger configuration

```java
import java.util.logging.FileHandler;
import java.util.logging.Logger;
...
private static void initLogger() {
    Logger logger = Logger.getLogger("com.bmw.myapp");
    logger.addHandler(new FileHandler());
    logger.setUseParentHandlers(false);
}

// In OpenJDK, the logger is lost when garbage collection
// is called immediately after initialization:
public static void main(final String[] args) {
    initLogger(); // Adds a file handler to the logger.
    System.gc(); // Logger configuration gets lost.
    // This won't be logged to the file as expected:
    Logger.getLogger("com.bmw.myapp").info("My message");
}
```

Robust logger configuration

```java
import java.util.logging.FileHandler;
import java.util.logging.Logger;
...
private static final Logger LOG;

static {
    LOG = Logger.getLogger("com.bmw.myapp");
    LOG.addHandler(new FileHandler());
    LOG.setUseParentHandlers(false);
}

// This code also works with OpenJDK:
public static void main(final String[] args) {
    System.gc(); // Logger configuration isn't lost.
    // This will be logged to the file as expected:
    LOG.info("My message");
}
```

SonarQube rule
[Experimental - Potential lost logger changes due to weak reference in OpenJDK]

Logging implementation

Guard expensive log statements

Guard expensive statements for logging purposes by log level checks, e.g. when the construction of the log message requires concatenating Strings or when log message parameters need to be retrieved.

Reasoning
Creating expensive log messages should not be performed when the appropriate log level is not set. To ensure that the log messages only get computed when needed, the call should be wrapped by a check for the log level.

SonarQube rule
Guard Debug Logging

Pass exceptions to log method

Call the two-argument log method to log exceptions, passing a String that describes the context and the exception. Pass the exception as a whole rather than just pieces of its state.

Reasoning
Using the exception as an argument for the log call ensures that the complete stack trace of the exception is logged.

SonarQube rules

Use Correct Exception Logging

Don't use printStackTrace()

When handling exceptions, don't use `Throwable.printStackTrace()`. Rather pass the exception to the logger.

Reasoning

The `printStackTrace()` methods direct their output to System.err (default) or to a `java.io.PrintStream` or `java.io.PrintWriter` passed to the method, thus circumventing the logger and hard coding the output target. This violates the *Don't log to hard-coded log target* rule.

SonarQube rule

Avoid Print Stack Trace

Don't guard Android logging statements by LOGD or LOGV

In Android applications, don't guard calls of `Log.d()` or `Log.v()` by checks of `android.util.Config.LOGD` or `android.util.Config.LOGV` respectively.

Reasoning

Guarding calls of `Log.d()` or `Log.v()` by evaluations of `android.util.Config.LOGD` /`android.util.Config.LOGV` before sending debug log messages is useless as `Config.LOGD` is always `true`. and `Config.LOGV` is always `false`. Furthermore, the `android.util.Config.LOGD`/`android.util.Config.LOGV` constants have been deprecated in API level 4, and the entire type `android.util.Config` has been deprecated in API level 14.

Loops

This section discusses rules and best practices which are related to recursion, `for` and `while` loops.

Iteration

Use loops rather than iterators

Use loops rather than iterators to iterate over Collections.

Reasoning

The code is easier to read.

Sample

Iteration over collection via loop

```java
public void foo(final Collection<ElementType> collection) {
    ...

    // The subsequent iterator ...
    Iterator<ElementType> iter = collection.iterator();
    ElementType element;
    while (iter.hasNext()) {
        element = iter.next();
        ...
    }

    // ... is equivalent to the subsequent loop:
    for (ElementType element : collection) {
        ...
    }

    ...
}
```

General coding style for loops

Body of loop requires braces

The bodies of `for`, `while` and `do-while` statements must be wrapped in curly braces.

Reasoning

Prevents logical errors when more statements are added to the block and increases the readability of source code.

SonarQube rule

Need Braces

Declare variables which are only used within a loop inside the loop

Declare variables which are used in loops inside the loop if they are not relevant outside the loop, rather than declaring them before the loop.

Reasoning

The scope of the variable is limited to the loop. The Java compiler is smart enough to only allocate the variable once, even if the loop is iterated multiple times.

Use while loop when termination criteria is determined in body of loop

Use `while` loop rather than `for` loop when the termination criteria is not an integer loop index or reaching the end of an iteration, or when the termination criteria is not calculated in the loop's declaration, but in its body.

Reasoning

Enhanced code readability and better reflects the character of the loop: `for` loops should always

- iterate over a collection or array, or
- iterate over an incremented or decremented loop index that serves as the termination criteria.

SonarQube rule

For Loop Should Be While Loop

Don't modify loop indices of for loops

The loop index (not valid for an iterator) of a `for` loop is to be thought as constant during each iteration and may not be directly modified inside the loop. If modifying the loop index inside the loop is required, use a `while` or a `do ... while` loop rather than a `for` loop.

Reasoning

Modifying the index variable of a `for` loop is surprising to a reader and hardens maintenance. Loops should be simple and easy to understand. Modifications of an outer `for` loop's index in an inner loop's declaration is a typical copy/paste error.

Sample

Outer loop index modification in inner loop

```
for (int i = 0; i < 10; i++) {
    // Illegal modification of i in inner loop:
    for (int j = 0; j < 20; i++) { // Should probably be j++.
        ...
    }
}
```

SonarQube rules

Dodgy - Complicated, subtle or wrong increment in for-loop

Modified Control Variable

Infinite repetition

Avoid unintentional infinite loops

Be careful with loops that doesn't seem to have a way to terminate (other than by perhaps throwing an exception).

Reasoning

An unintentional infinite loop (also known as an endless loop or unproductive loop) normally caused the entire system to become unresponsive.

SonarQube rule

Correctness - An apparent infinite loop

Avoid infinite recursion

Don't have methods unconditionally invoke themselves.

Reasoning

Unconditional self invocations indicate an infinite loop and result in stack overflow.

SonarQube rule

Correctness - An apparent infinite recursive loop

Methods

This section discusses rules and best practices which are related to method usage, returns and reflection.

Method usage

Don't simulate call-by-reference

Don't wrap single object arguments by one-element arrays or collections to simulate call-by-reference.

Reasoning

Such patterns lead to less obvious logic and overhead for wrapping. Rather change the OO design of your classes to reflect the necessity of changing references, e.g. by adding a return type to a void method or by designing fitting Objects to be passed.

Don't ignore return values

Check return value of methods.

Reasoning

The return value is the same type as the type the method is invoked on, and from our analysis it looks like the return value might be important (e.g., like ignoring the return value of `String.toLowerCase()`). We are guessing that ignoring the return value might be a bad idea just from a simple analysis of the body of the method. You can use a `@CheckReturnValue` annotation to instruct FindBugs as to whether ignoring the return value of this method is important or acceptable. Please investigate this closely to decide whether it is OK to ignore the return value.

Sample

Ignored return value error

```java
String aString = " Some chars ";

// The following code doesn't modify aString's value...
aString.trim();

// ... whereas this statement does:
aString = aString.trim();
```

SonarQube rules

Correctness - Method ignores return value [2]

Method ignores return value, is this OK?

Bad practice - Method ignores exceptional return value

Ensure correct argument order

The arguments to this method call seem to be in the wrong order.

Reasoning

The arguments to this method call seem to be in the wrong order. For example, a call `Preconditions.checkNotNull("message", message)` has reserved arguments: the value to be checked is the first argument.

SonarQube rule

Reversed method arguments

Prohibited or restricted methods

Don't use deprecated APIs

Don't call deprecated methods or reference deprecated fields and constants.

Reasoning

Usage of deprecated APIs increases the difficulty of future migrations or updates, as the deprecated members might be removed in the future. If the functionality of a deprecated member is still needed, check the member's documentation on how to access the functionality in a non-deprecated way.

SonarQube rule

Avoid use of deprecated method

Don't call unsupported methods

Don't call methods in a way that provokes an `UnsupportedOperationException` to be thrown.

Reasoning

The `java.lang.UnsupportedOperationException` is intended to indicate that a called method is not supported, which is typically the case if the operation has not yet been fully implemented for the code segment that throws the exception. Calling such a method with a parametrization that raises the exception will not behave as expected and make the application unstable.

SonarQube rule

Dodgy - Call to unsupported method

Avoid terminating the JVM via <methods>

Methods `System.exit(int)`, `Runtime.exit(int)` and `Runtime.halt(int)` may not be used in code that might be executed on an application server.

Reasoning

Calling those methods will usually terminate not only the application at hand but the whole JVM. This is unwanted especially if an application does not run stand-alone but in an extended context, like an application server or a servlet container or even when used as a library. For further information and controls please refer to the following documentation:

- http://docs.oracle.com/javase/7/docs/api/java/lang/SecurityManager.html
- the doc. of your application server, servlet container, etc.

SonarQube rules

Bad practice - Method invokes System.exit(...)

Avoid finalizers

Finalizers are unpredictable, often dangerous, and generally unnecessary.

Reasoning

Finalizers are unpredictable, often dangerous, and generally unnecessary. Their use can cause erratic behavior, poor performance, and portability problems.

SonarQube rule

Bad practice - Method invokes dangerous method runFinalizersOnExit

Adhere to the Android API for hook methods

When overriding method hooks which are implemented by a superclass classes, call the superclass's method implementation. Depending on the type of callback, the superclass's implementation must be either called as the first or as the last statement of the overriding code. See the superclass method's documentation for details on the correct location of the superclass method call.

Reasoning

The superclass implementation might contain code that is relevant for a correct behavior of the application. This is especially important in hooks that control an object's life cycle and if the superclass holds the object's state or part of it.

SonarQube rules

Android - call super first

Android - call super last

Returns

Avoid returning null for primitive wrapper types

Avoid returning `null` from a methods with return a wrapper object for a primitive type, e.g. a `java.lang.Boolean`.

Reasoning

When returning a wrapper type for a primitive, the method can be invoked as though it returned a primitive value, and the compiler will insert automatic unboxing of the wrapper value. If a `null` value is returned, this will result in a `java.lang.NullPointerException`.

Sample

Return null for primitive error

```java
public static Boolean nullForPrimitive() {
    return null; // Critical return of null for primitive wrapper.
}

public static void callToNullForPrimitive() {
    // This will compile, but throw a NullPointerException:
    boolean b = nullForPrimitive();
}
```

Avoid modifications of turned in parameters

Avoid modifications of parameters that have been passed to a method. In situations where a modification of a parameter which has been passed to the method is unavoidable, don't use the same parameter as a return value. rather consider to define the method's return type as `void`, or to return a modified copy of the turned in argument.

Reasoning

A turned in parameter is intended to be used in a read-only way, unless otherwise documented, e.g. in the parameter's Javadoc. Returning a modified parameter is confusing to callers of the method, since it is not apparent that the turned in original has been modified. If the result of a modification of a turned in argument is returned, one would expect a modified clone of the argument rather that the argument itself.

Simplify boolean returns

If a boolean expression serves to calculate a return value, directly return the evaluation result rather than storing it in a local variable or using `if ... else ...` statements for the return.

Reasoning

The boolean expression can be used as the argument of the return statement, which reduces program code and enhances code readability.

Sample

Simplified boolean return

```java
// Replace ...
if (<boolean expression>) {
    return true;
} else {
    return false;
}
```

```java
// ... with:
return <boolean expression>;
```

SonarQube rule

[Simplify Boolean Return]

Reflection

Prefer static resolution of class objects

Try to use the class literal (`<name>.class`) whenever possible. Only use Reflection to dynamically resolve class objects via `Class.forName()` when it is really necessary.

Reasoning

The class literal has two advantages over the dynamic resolution via Reflection:

1. The source code can be refactored more easily (e.g. automatically via IDE) while in the call of `Class.forName("Name")` the passed String constant would have to be changed manually.
2. The compiled class file can more easily be modified by byte code generation, as the above mentioned string constant would also be be hard compiled into the byte code.

Null

Rules and best practices on how to safely deal with object references that can be `null`.

This section discusses rules and best practices on how to safely deal with object references that can be `null`.

Dereferencing null

Null values must not be dereferenced

Ensure that variables or fields which contain `null` are never dereferenced at runtime. This rule applies to all execution paths of the code, including exceptional paths and `default` paths in `switch` statements.

Reasoning

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

SonarQube rules

Correctness - close() invoked on a value that is always null

Correctness - Null pointer dereference

Correctness - Null pointer dereference in method on exception path

Correctness - Null value is guaranteed to be dereferenced

Correctness - Possible null pointer dereference

Correctness - Possible null pointer dereference in method on exception path

Correctness - Read of unwritten field

Correctness - Value is null and guaranteed to be dereferenced on exception path

Dodgy - Load of known null value

Dodgy - Possible null pointer dereference due to return value of called method

Dodgy - Possible null pointer dereference on path that might be infeasible

Read of unwritten public or protected field

Don't pass null for mandatory method parameters

Don't pass `null` as an argument for a method parameter which expects a non-null value.

Reasoning
Passing null for a mandatory method parameter violates the method's interface contract. In the best case, this will yield an `IllegalArgumentException`, or it will cause a `NullpointerException` to be thrown or, as the worst case, result in unpredictable erroneous processing.

SonarQube rules
Correctness - Method call passes null for nonnull parameter

Correctness - Method call passes null for nonnull parameter (ALL_TARGETS_DANGEROUS)

Correctness - Method call passes null to a nonnull parameter

Correctness - Non-virtual method call passes null for nonnull parameter

Don't assign null to mandatory fields

Don't assign `null` to a field which is expected to be non-null.

Reasoning
When a field is documented or annotated to be non-null, code that dereferences the field cannot be expected to check if the field contains a `null` value. This can easily cause unexpected application behavior and significantly reduce stability.

SonarQube rule
Correctness - Store of null value into field annotated NonNull

Null checks

Check mandatory method parameters for null values

Check if `null` values have been passed to mandatory method parameters and throw an `IllegalArgumentEx ception` in case of a detected `null` value.

Reasoning

Such null checks significantly improve the code's stability and maintainability, as unintended or false method invocations are detected early. The stack trace of the thrown exception clearly points at the location of the coding error. Additional computational overhead for null checks can be ignored, as it is very low.

SonarQube rule

Correctness - Method does not check for null argument

Don't guard instanceof with null check

Don't guard instanceof checks with a `null` check.

Reasoning

The check for `null` is superfluous, as `instanceof` evaluates `false` for `null` values: A conditional such as

```
if ((a != null) && (a instance of MyClass))
```

can be expressed shorter as

```
if (a instance of MyClass).
```

SonarQube rule

Simplify Conditional

Avoid broken or misplaced null checks

Assure that a field which is guarded by a null check is never dereferenced when it contains `null`.

Reasoning

Broken or misplaced null checks are a frequent cause for errors, and typically result from a logic error or a typo.

Samples

Broken null checks

```
public void foo(final String name) {

    // These null checks will fail ...
    if ((name == null) && name.isEmpty()) ...
    if ((name != null) || (! name.isEmpty())) ...

    // ... whereas these will work:
    if ((name == null) || name.isEmpty()) ...
    if ((name != null) && (! name.isEmpty())) ...

}
```

Misplaced null check

```
public void foo(final String name) {

    // This null check will fail ...
    if (name.isEmpty() || (name == null)) ...

    // ... whereas this will work:
    if ((name == null) || (name.isEmpty())) ...

}
```

SonarQube rules

Broken Null Check

Misplaced Null Check

Avoid useless null checks

Don't code `null` checks which have no effect on the control flow, like null checks on variables or fields

- which have been previously dereferenced;
- if the field or/variable is reassigned independent of the `null` evaluation result.

Reasoning
Such checks are confusing and indicate a logic error or typo.

SonarQube rules
Correctness - Nullcheck of value previously dereferenced

Returning null

Methods must not return null unless explicitly documented

Methods must not return `null` under any conditions when the interface contract doesn't explicitly document that `null` values may be returned. Especially, returning `null` from methods which explicitly declare not to do so is unacceptable.

Reasoning
The nature of the return values can be documented via Javadoc. If no such documentation exists, the corresponding member should be expected to never return null. Otherwise, code which invokes the method would either be likely to run into unexpected and false behavior at runtime, or it would have to be unnecessarily complex only to check if the invoked method really fulfills its interface contract.

SonarQube rule
Correctness - Method may return null, but is declared @NonNull

Return empty array or collection rather than null

It is often a better design to return a length zero array rather than a null reference to indicate that there are no results

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

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

SonarQube rule
Dodgy - Consider returning a zero length array rather than null

Superfluous code

This section discusses code which is either unused or has no effect. Such code is a constant cause for pain, as it unnecessarily increases software complexity and maintenance. In many cases, obsolete code is an indication for logic errors.

Unused code

Remove empty Java files

Remove Java code files that are either empty or only contain comments.

Reasoning
Such files are just confusing, as they suggest the presence of a type that doesn't exist.

SonarQube rule
Empty file

Remove empty blocks

Remove empty blocks, e.g. in `if ... else` clauses or loops.

Reasoning
Superfluous code that just reduces readability. Typically, such code is a leftover from code refactoring.

SonarQube rule
Empty Block

Remove unreachable code

Remove code which can never be executed because a condition to reach the code always evaluates `false`.

Reasoning
Such code is confusing, as it is often hard to detect and suggests that its containing method was doing something else than what it really does.

Remove commented-out code

Remove commented-out lines of code.

▾ Reasoning
Commented-out lines of code smells, as its relevance is unclear and time consuming to find out, and as it distracts the flow when reading the code. Rather trust the SCM engine if you want to keep older versions of your code.

▾ SonarQube rule
Avoid commented-out lines of code

Useless code

Remove empty statements

Remove statements which just consist of the `';'` character.

▾ Reasoning
Superfluous code.

▾ SonarQube rule
Empty Statement

Remove pointless method invocations

Remove pointless method invocations, such as calling `new String();`.

▾ Reasoning
Such operations are useless and confusing. Check for a logic error when encountering such code.

▾ SonarQube rule
D'oh! A nonsensical method invocation

Remove unread local variables

Don't define local variables that are never read after initialization or assign a value to a local variable that is never read after the assignment.

▾ Reasoning
Such variables and assignments are just dead code that reduces readability and needs to be maintained. Typically, unread local variables are a leftover from implementation or debugging, or they result from mixing up a static or instance variable with a local one, or they are an indication for a forgotten assignment of the variable's value to a member field.

▾ SonarQube rules
Unused local variable

Don't assign unread value to local variable

Don't assign a non-null value to a local variable if the value is never read after the assignment. When using Java 6 or higher, also don't assign `null` to a local variable if the variable is not read after the assignment.

Reasoning

Assigning an unread non-null value to a local variable is useless and just dead code. If you encounter such an assignment, check if an assignment to a member field with a similar name could have been intended.

Until Java 5, assigning `null` to a local variable could assist the garbage collector. As of Java 6, this is no longer needed or useful.

SonarQube rules

Dead store to local variable that shadows field

Dodgy - Dead store of null to local variable

Remove self assignments

Remove assignments of a value to itself.

Reasoning

Self assignments such as in `x = x;` are idempotent and have no effect. Check for typos or logic errors, as e.g. the assignment was intended between a local variable and a member field or vice versa with similar names.

SonarQube rules

Correctness - Self assignment of field

Dodgy - Self assignment of local variable

Self assignment of local rather than assignment to field

Remove double assignments

Remove double assignments of a value to the same member or variable as in `x = x = 3;`. Multiple assignments in the same statement as in `x = y = 3;` are generally prohibited and need to be split into two separate statements.

Reasoning

Double assignments such as in `x = x = 3;` are useless, as `x = 3;` serves the same purpose. Multiple assignments in the same statement are confusing, splitting them up into multiple statements makes things clearer.

SonarQube rules

Correctness - Double assignment of field

Dodgy - Double assignment of local variable

Don't use vacuous getter/setter sequences

Don't overwrite values by themselves, e.g. by invoking a getter to retrieve a field's or variable's value, followed by a setter which sets the same value to the same field or variable.

▾ Reasoning
Such sequences of method calls have no effect and are an indication for a logic error.

Remove nonsensical self computations

Remove nonsensical computations on a member field or variable like `x & x` or `x - x`.

▾ Reasoning
Such computations obscure what the code is doing: `x & x` performs a useless bitwise operation that returns `x`, and `x - x` equals to `0`. Replacing the statement by its result makes things obvious and is easier to understand and maintain. Before resolving such a nonsensical computation, check if the statement was caused by a typo or logic error.

▾ SonarQube rules
Correctness - Nonsensical self computation involving a field (e.g., x & x)

Correctness - Nonsensical self computation involving a variable (e.g., x & x)

Remove or refactor unnecessary checks or assertions

Remove or refactor unnecessary checks that will never or always fail, e.g.

- `null` checks after creating an object with the `new` operator;
- checks if an auto-boxed wrapper for a primitive is `null`;
- redundant `null` checks of values known to be null or non-null;
- comparisons of any combination of two values known to be null or non-null;
- nonsensical comparisons like `((a & 0) == 0)`;
- assertions that will never or always fail.

▾ Reasoning
Such code is either superfluous or erroneous:

- The `new` operator is guaranteed to succeed or throw an exception.
- As primitives can never be `null`, their auto-boxed object instances aren't either.
- Redundant `null` checks are an indication for a logic error or typo.
- Comparisons of known values should be replaced by the comparison result.
- Nonsensical comparisons should be revised carefully before being replaced by their result, as they are likely to result from a logic error or typo.
- Assertions that will never fail have no value, assertions that always fail break the stability of the code.

▾ SonarQube rules
Correctness - Check to see if ((...) & 0) == 0

Dodgy - Redundant comparison of non-null value to null

Dodgy - Redundant comparison of two null values

Dodgy - Redundant nullcheck of value known to be non-null

Dodgy - Redundant nullcheck of value known to be null

Avoid unnecessary type checks

Don't use the `instanceof` operator when type compatibility can be statically determined for values other than `null`.

Reasoning

The result of the type check can be determined from the code and doesn't require to be evaluated at runtime. Thus, the type check and its surrounding conditional are superfluous.

Using `instanceof` in a way that evaluates `true` or `false` for all values other than `null` is confusing and should be replaced by a simple and much better readable null check.

SonarQube rules

Correctness - instanceof will always return false

Correctness - Unnecessary type check done using instanceof operator

Dodgy - instanceof will always return true

Don't use instanceof on null argument

Don't use the `instanceof` operator when the argument is always `null`.

Reasoning

The type comparison ill always evaluate `false`, which is a clear indication of a severe logic error, possibly caused by a typo.

SonarQube rule

Correctness - A known null value is checked to see if it is an instance of a type

Don't call String.substring(0)

Don't invoke `String.substring()` with just a zero begin index argument, but no end index.

Reasoning

The returned String is the original value, which makes the statement either superfluous or an indication for a logic error.

SonarQube rule

Dodgy - Invocation of substring(0), which returns the original value

Avoid vacuous calls to collections

For any collection `c`, don't make superfluous calls like `c.containsAll(c)` or `c.retainAll(c)`.

Reasoning

Such operations have no effect, but can be very time consuming: `c.containsAll(c)` is always `true`, and `c.retainAll(c)` leaves the collection untouched. Before removing such a statement, check for logic errors or typos, as it is very likely that the method has been invoked with the wrong argument.

SonarQube rule

Correctness - Vacuous call to collections

Don't call intern() on String constants

Don't call `String.intern()` on a String constant or literal.

Reasoning

All String constants and literals are interned by definition. A call to `intern()` is superfluous and decreases code understandability for less experienced programmers.

Concurrency

This chapter discusses rules and best practices on how to deal with concurrency in Java code.

Threads

Rules and best practices on ow to handle threads in Java applications.

Thread-safety

Rules and best practices on how to synchronize concurrent access to objects accessible from multiple threads.

Wait, sleep and notifications

Rules and best practices on how to synchronize threads by using waits, sleep and notifications.

Threads

This section discusses rules and best practices on how to handle threads in Java applications.

- ClassLoader usage
  - *Use Thread.currentThread().getContextClassLoader() to obtain ClassLoader*
- Thread usage
  - *Use Threads (in (JEE) applications) carefully*
  - *Don't call Thread.run()*
  - *Don't call Thread.start() in constructors*
  - *Don't create empty threads*
  - *Don't use thread priorities*
  - *Don’t depend on the thread scheduler*
  - *Prefer Runnable over Thread*
  - *Don't call Thread.interrupted() on Thread object*
- ThreadGroup usage
  - *Avoid using ThreadGroup*
- ScheduledThreadPoolExecutor usage
  - *Don't create ScheduledThreadPoolExecutor with corePoolSize of zero*
  - *Don't call ScheduledThreadPoolExecutor.setMaximumPoolSize()*
- GUI programming
  - *Call Swing UI visualization methods in Swing thread only*

ClassLoader usage

Use Thread.currentThread().getContextClassLoader() to obtain ClassLoader

Obtain the class loader via `Thread.currentThread().getContextClassLoader()` instead of `Class.getClassLoader()`.

Reasoning

In JEE, `Class.getClassLoader()` might not work as expected as application servers typically use multiple concurrent class loaders. A class loader obtained via `getClassLoader()` is likely to be different from the class loader which can be called from the current thread. Using `Thread.currentThread().getContextClassLoader()` is safe in every context.

Furthermore, code which has initially been written for a non-JEE application can easily be reused in a JEE context without the need to refactor class loader retrieval or otherwise the hazard of locating this hard to find bug.

SonarQube rule

Use Proper Class Loader

Thread usage

Use Threads (in (JEE) applications) carefully

Create, manipulate or terminate threads carefully.

Reasoning

Threads allow multiple activities to proceed concurrently. Concurrent programming is harder than single-threaded programming, because more things can go wrong, and failures can be hard to reproduce. But you can’t avoid concurrency. It is inherent in much of what we do, and a requirement if you are to obtain good performance from multicore processors, which are now commonplace. Threads are particularly complicated in a JEE environment therefore it exists a own JSR for that topic (JSR 236: Concurrency Utilities for JavaTM EE)

Don't call Thread.run()

Don't make explicit calls to the `run()` method of `Thread` or other classes that implement `Runnable`. Use `Thread.start()` if you intend to invoke `run()` in a new thread.

Reasoning

When calling `run()` explicitly, the method will be executed in the caller's thread of control. Probably that is not what is intended and – if intended – violates the principle of least surprise. `Thread.start()` spawns a new thread and invokes the `run()` method in that thread.

SonarQube rule

Multithreaded correctness - Invokes run on a thread (did you mean to start it instead?)

Don't call Thread.start() in constructors

Do not start threads in constructors.

Reasoning

If the class gets subclassed, the thread will be started before the constructor of the subclass is executed. This may lead to errors or at least will produce unexpected behavior.

SonarQube rule
Multithreaded correctness - Constructor invokes Thread.start()

Don't create empty threads

Don't create empty threads. A thread is empty if

- it is derived from `java.lang.Thread`, but doesn't overwrite the `run()` method, or
- it is an instance of `java.lang.Thread` that doesn't get a `Runnable` passed.

Reasoning
Most likely a programming error. Such threads do nothing but waste CPU cycles.

SonarQube rule
Multithreaded correctness - A thread was created using the default empty run method

Don't use thread priorities

Avoid using thread priorities.

Reasoning
Thread priorities are not portable and not guaranteed to have the same semantics or even work identical on different operating systems. Thus, using thread priorities may imply varying system behavior on different platforms which can lead to issues that are hard to debug. Furthermore, using thread priorities may mask underlying multithreading issues like race conditions for a certain operating system or JVM, which could reappear on others.

Don’t depend on the thread scheduler

Do not depend on the thread scheduler for the correctness of your program. The resulting program will be neither robust nor portable. As a corollary, do not rely on `Thread.yield()`.

Reasoning
When many threads are runnable, the thread scheduler determines which ones get to run, and for how long. Any reasonable operating system will try to make this determination fairly, but the policy can vary. Therefore, well-written programs shouldn’t depend on the details of this policy. Any program that relies on the thread scheduler for correctness or performance is likely to be nonportable.

Prefer Runnable over Thread

`Runnable` makes your methods and classes more flexible.

Reasoning
Although `Thread` implements `Runnable`, it carries a lot of overhead related to the underlying system resources. Instantiating `Thread`-instances without starting them may even lead to a memory leak with older JREs.

SonarQube rule
Dodgy - Thread passed where Runnable expected

Don't call Thread.interrupted() on Thread object

Only call the `Thread.interrupted()` in a static fashion by preceding the method name with the class name, rather than invoking the method on a `Thread` object.

▾ Reasoning

The `Thread.interrupted()` method is static and interrupts the current thread. If invoked on another `Thread` instance than the current `Thread`, the current `Thread` will be interrupted, but not the one that the method is invoked on. This is confusing. Calling the method in a static fashion is much clearer and helps avoiding unintended side effects.

▾ SonarQube rule

Correctness - Static Thread.interrupted() method invoked on thread instance

ThreadGroup usage

Avoid using ThreadGroup

Avoid using `java.lang.ThreadGroup`.

▾ Reasoning

Although `ThreadGroup` is intended to be used in a threaded environment it contains methods that are not thread-safe.

▾ SonarQube rule

Avoid Thread Group

ScheduledThreadPoolExecutor usage

Don't create ScheduledThreadPoolExecutor with corePoolSize of zero

Don't create `java.util.concurrent.ScheduledThreadPoolExecutor` with corePoolSize of zero.

▾ Reasoning

A `ScheduledThreadPoolExecutor` with `corePoolSize` zero will never execute anything for Java version < 7u4, thus the Object is useless. In later versions it is not possible to create a `corePoolSize` of zero.

▾ SonarQube rule

Correctness - Creation of ScheduledThreadPoolExecutor with zero core threads

Don't call ScheduledThreadPoolExecutor.setMaximumPoolSize()

When dealing with `ScheduledThreadPoolExecutor`, don't invoke the `setMaximumPoolSize(int)` metho d.

▾ Reasoning

While ScheduledThreadPoolExecutor inherits from ThreadPoolExecutor, a few of the inherited tuning methods are not useful for it. In particular, because it acts as a fixed-sized pool using corePoolSize threads and an unbounded queue, adjustments to maximumPoolSize have no useful effect.

▾ SonarQube rule
Correctness - Futile attempt to change max pool size of ScheduledThreadPoolExecutor

GUI programming

Call Swing UI visualization methods in Swing thread only

Invoke Swing methods `pack()`, `setVisible()` and `show()` in Swing thread only.

▾ Reasoning
"The Swing methods `show()`, `setVisible()`, and `pack()` will create the associated peer for the frame. With the creation of the peer, the system creates the event dispatch thread. This makes things problematic because the event dispatch thread could be notifying listeners while pack and validate are still processing. This situation could result in two threads going through the Swing component-based GUI – it's a serious flaw that could result in deadlocks or other related threading issues. A `pack()` call causes components to be realized. As they are being realized (that is, not necessarily visible), they could trigger listener notification on the event dispatch thread." (taken from the rule's documentation)

▾ SonarQube rule
Bad practice - Certain swing methods need to be invoked in Swing thread

Thread-safety

This section discusses rules and best practices on how to synchronize concurrent access to objects accessible from multiple threads.

Problematic types in a multi-threaded context

Don't call Locale.setDefault() in internationalized applications

In applications which must support multiple locale settings in parallel, e.g. internationalized applications running on a JEE server, do not use the `Locale.setDefault()` method to change the default locale.

Reasoning

A call to this method changes the JVM's default locale for all threads and makes your applications unsafe to threads. It does not affect the host locale. Since changing the JVM's default locale may affect many different areas of functionality, this method should only be used if the caller is prepared to reinitialize locale-sensitive code running within the same Java Virtual Machine, such as the user interface.

If your application has to support multiple `Locale` settings in parallel, provide `Locale` arguments to methods whose behavior depends on the `Locale` and which would otherwise use the default `Locale`. Call methods with a `Locale` argument over other overloading methods without a `Locale` argument.

Don't use Calendar or DateFormat without synchronization

Do not share instances of `java.util.Calendar` or `java.text.DateFormat` or subclasses thereof across thread boundaries without proper synchronization. This may be the case if you use instances obtained via a static field.

Reasoning

Even though the Javadoc does not contain a hint about it, Calendars and DateFormats are inherently unsafe for multi-threaded use. Sharing a single instance across thread boundaries without proper synchronization will result in erratic behavior of the application and may cause serialization problems. Under Java 1.4 problems surface less often than under Java 5 where you will probably see random `ArrayIndexOutOfBoundsExceptions` or `IndexOutOfBoundsExceptions` in `sun.util.calendar.BaseCalendar.getCalendarDateFromFixedDate()`.

For more information on this see Sun Bug #6231579 and Sun Bug #6178997.

SonarQube rules

Multithreaded correctness - Call to static Calendar

Multithreaded correctness - Call to static DateFormat

Multithreaded correctness - Static Calendar

Multithreaded correctness - Static DateFormat

Don't use mutable fields in servlets

Don't use mutable fields in Servlet-classes.

Reasoning

A web server generally only creates one instance of a servlet or jsp class (i.e., treats the class as a Singleton), and will have multiple threads invoke methods on that instance to service multiple simultaneous requests. Thus, having a mutable instance field generally creates race conditions.

Having the `Servlet` implement the `SingleThreadedModel` would solve this issue, but can significantly degrade the performance of the web application as the servlet can only process a single call at a time. Thus, this is not an adequate solution.

SonarQube rule

Multithreaded correctness - Mutable servlet field

Synchronization mechanism

Prefer atomic variables rather than volatile

Instead of declaring a variable as `volatile`, use the types provided by the `java.util.concurrent.atomic` package for lock-free thread-safe programming on single variables.

Reasoning

When a field is declared volatile, the compiler and runtime are put on notice that this variable is shared and that operations on it should not be reordered with other memory operations. Volatile variables are not cached in registers or in caches where they are hidden from other processors, so a read of a volatile variable always returns the most recent write by any thread. Accessing a volatile variable performs no locking and so cannot cause the executing thread to block. Therefore volatile variables can only guarantee visibility. If you need atomicity you must use locking or atomic variables.

SonarQube rule

Avoid Using Volatile

Synchronization granularity

Avoid synchronization at method level

Avoid synchronizing at method-level. Use synchronized blocks instead.

▼ Reasoning

When more code gets added to the method during further development or maintenance, this may lead to too coarse grained locking with all negative implications like performance bottlenecks and increasing risk of deadlocks.

▼ SonarQube rule

Avoid Synchronized At Method Level

Keep synchronized blocks short

Keep synchronized blocks as short as possible. Move code outside of synchronized blocks when possible.

▼ Reasoning

The more code a synchronized block contains, the longer the lock is held which increases the probability for lock contention. This may lead to performance bottlenecks.

Synchronization semaphores

Use locks as synchronization semaphores

Use privately owned instances of `java.util.concurrent.locks.Lock` which are not externally exposed as synchronization semaphores.

▼ Reasoning

The Java classes which implement the `java.util.concurrent.locks.Lock` interface effectively support a large variety of synchronization scenarios. By implementing such locks as privately owned members without external access, deadlocks from usage of the same semaphore from outside the class where the synchronization is performed can be reliably suppressed.

Don't synchronize on externally accessible objects

Never use object-instances to synchronize on, that are not owned by the class where the synchronization is performed or that are externally accessible. Use private members for synchronization instead. Remember that `t his` cannot be considered as being owned by the current instance. So avoid synchronizing on `this` as well.

▼ Reasoning

If an object uses itself or an externally exposed object for synchronization, then there is the possibility that another method in some class might use the same object for synchronization, which can lead to deadlocks which are very hard to detect as the cause of the bug.

▼ SonarQube rule

Dodgy - Class exposes synchronization and semaphores in its public interface

Don't synchronize on singletons

Do not use singletons as synchronization semaphores.

Reasoning

Singletons are accessible from outside their implementing type and can be used by other threads or other code running in the same thread for synchronization, which can easily result in deadlocks.

Don't synchronize on enumeration values

Do not use enumeration values as synchronization semaphores.

Reasoning

Each enumeration value is only instantiated once. If multiple threads or multiple variables in the same thread independently obtain the same enumeration value, they will share the same instance of that value, which can easily lead to unexpected deadlocks.

Don't synchronize on objects returned by getClass()

Don't synchronize on an instance of `Class` that is obtained via a call to `getClass()`.

Reasoning

When your class is subclassed, a call to `getClass()`will yield a different `Class` instance that you are now synchronizing on. This means that synchronization won't work as expected. Race conditions and other threading issues are likely to arise. Furthermore, instances of `java.lang.Class` returned by factory methods `java.lang.Object.getClass()` or `java.lang.Class.forName(String)` behave like enumeration values, which compromises synchronization if multiple threads independently synchronize on the same `Class` instance.

SonarQube rule

Multithreaded correctness - Synchronization on getClass rather than class literal

Don't synchronize on boxed primitives or strings

Do not synchronize on boxed primitive constants (e.g. `Boolean.FALSE` or a boxed `Integer` constant) or `String` constants (interned Strings).

Reasoning

Since Boolean, Integer and String objects can be cached and shared, this code could be synchronizing on the same object as other, unrelated code, leading to unresponsiveness and possible deadlocks.

SonarQube rules

Multithreaded correctness - Synchronization on Boolean could lead to deadlock

Multithreaded correctness - Synchronization on boxed primitive could lead to deadlock

Multithreaded correctness - Synchronization on boxed primitive values

Multithreaded correctness - Synchronization on interned String could lead to deadlock

Don't synchronize on an updated field

When synchronizing on an object make sure that you are not synchronizing on a mutable field.

Reasoning

If you synchronize on an object referenced from a mutable field, different threads may obtain different objects to synchronize on. This means that synchronization doesn't work or at least doesn't work as expected. Race conditions and other threading issues are likely to arise.

▾ SonarQube rule
Multithreaded correctness - Method synchronizes on an updated field

Don't synchronize on a nullable field

Ensure that the object used for synchronization is not null. Since the object used for synchronization must not be null, don't perform superfluous null checks on that object in the synchronized block.

▾ Reasoning
If the object used for synchronization was null, a `NullPointerException` would be thrown on entering the synchronized block.

▾ SonarQube rule
Multithreaded correctness - Synchronize and null check on the same field

Synchronization scope

Ensure consistent synchronization

Thread-safe classes must synchronize both read and write access to all static variables and members which require thread-safe access. This includes the `writeObject()` method when implementing custom object serialization.

▾ Reasoning
Unsynchronized access to the resource – even from only one method – will comprise the whole concept of synchronization and may lead to race conditions and other issues that are hard to detect and analyze.

▾ SonarQube rules
Multithreaded correctness - Class's writeObject() method is synchronized but nothing else is

Multithreaded correctness - Inconsistent synchronization (2)

Multithreaded correctness - Unsynchronized get method, synchronized set method

Don't declare readObject() as synchronized

When implementing custom object serialization, don't declare the method `readObject()`as `synchronized`.

▾ Reasoning
By definition, an object created by deserialization is only reachable by one thread, and thus there is no need for `readObject()` to be synchronized. If the `readObject()` method itself is causing the object to become accessible from another thread, that is an example of very dubious coding style.

▾ SonarQube rule
Multithreaded correctness - Class's readObject() method is synchronized

Don't synchronize access to fields that are only modified during construction

Do not use synchronization for class or instance members which are only modified in a static initializer or constructor.

Reasoning

Constructors and static initializers are guaranteed to be thread safe. Defining synchronization is unnecessary and a potential performance bottleneck.

Synchronize increments/decrements on volatile fields

Apply proper synchronization when incrementing/decrementing numeric fields. Declaring the field as `volatile` is not sufficient.

Reasoning

The semantics of volatile are not strong enough to make the increment/decrement operation atomic, unless you can guarantee that the variable is written only from a single thread.

SonarQube rule

An increment to a volatile field isn't atomic

Use atomic array classes to access elements of volatile arrays

If you need to have `volatile` array elements, use one of the atomic array classes in `java.util.concurrent`. Simply declaring the reference to the array as `volatile` won't work.

Reasoning

With a `volatile` reference to an array, reads and writes of the reference to the array are synchronized, but the array elements are still non-volatile.

SonarQube rule

Multithreaded correctness - A volatile reference to an array doesn't treat the array elements as volatile

Remove empty synchronized blocks

Remove synchronization which don't contain any code.

Reasoning

Empty synchronized blocks are useless, as they only obtain a lock that is immediately returned. Furthermore, empty synchronized blocks increase the risk of deadlocks if they cannot obtain the requested lock.

SonarQube rule

Multithreaded correctness - Empty synchronized block

Lazy initialization of fields

Synchronize lazy initialization of fields

When using lazy initialization for a field in a multi-threaded environment make sure that assigning the instance to the field and initialization of the instance is synchronized.

Reasoning

When assigning the instance to the field and initialization is not synchronized, the setting of the field is visible to other threads as soon as it is set. This may lead to other threads accessing the instance while it is not completely initialized. Even if the instance is initialized before it is assigned to the field, the instructions may be reordered by the compiler for optimization purposes and other threads would see an object that is not completely initialized.

For more information, see the Java Memory Model web site.

SonarQube rules

Multithreaded correctness - Incorrect lazy initialization and update of static field

Multithreaded correctness - Incorrect lazy initialization of static field

Avoid double-checked locking

Make sure that you don't use the dysfunctional double-checked locking idiom when you apply lazy initialization.

Reasoning

Although code like the following seems to correctly synchronize initialization of a field, due to the Java memory model and/or compiler optimizations this is not the case.

Double-checked locking

```java
package com.bmw.my.package;
...
public class MyClass {
    ...
    private static MyClass instance;
    ...
    public static MyClass instance() {
        // Beware of such code. It might not work as expected.
        if (instance == null) {
            synchronized (Class.forName("com.bmw.my.package.MyClass")) {
                if (instance == null) { // The erroneous double-check.
                    instance = new MyClass();
                }
            }
        }
        return instance;
    }
    ...
}

```

This issue is known as the double-checked locking idiom. For more information, see The "Double-Checked Locking is Broken" Declaration.

Sample

Double-checked locking

```java
package com.bmw.my.package;
...
public class MyClass {
    ...
    private static MyClass instance;
    ...
    public static MyClass instance() {
        // Beware of such code. It might not work as expected.
        if (instance == null) {
            synchronized (Class.forName("com.bmw.my.package.MyClass")) {
                if (instance == null) { // The erroneous double-check.
                    instance = new MyClass();
                }
            }
        }
        return instance;
    }
    ...
}

```

▾ SonarQube rule
Multithreaded correctness - Possible double check of field

Using thread-safe types

Synchronize sequence of calls to a thread-safe object

If you intend to manipulate a thread-safe abstraction atomically by applying subsequent calls to the object, sometimes it is necessary to manually synchronize the call sequence (e.g. `ConcurrentHashMap`).

▾ Reasoning
Thread-safety only assures that each single call to the object of the thread-safe class is thread-safe and executed atomically. Subsequent calls to a thread-safe object will not be synchronized.

▾ SonarQube rule
Sequence of calls to concurrent abstraction may not be atomic

Don't rely on thread-safe types

Don't rely on thread-safe types for static fields or instance members as a substitute for synchronization. Consider using non thread-safe types for objects which are not externally exposed. This especially applies to collections, as collections should not be externally exposed directly.

If you are using	consider using
`java.lang.StringBuffer`	`java.lang.StringBuilder`
`java.util.Hashtable`	`java.util.HashMap`
`java.util.Vector`	`java.util.ArrayList`

▾ Reasoning
Using thread-safe types for such objects instead of proper synchronization can compromise object integrity and degrade performance:

- If non-synchronized code modifies an externally exposed thread-safe static field or member, external modifications of the object between two calls to the object cannot be prevented, which can lead to erroneous operations on the object, e.g. if the first statement reads the object's state and a subsequent statement modifies the object based on the obtained state. In between, the obtained state could have been externally modified, resulting in invalid state of the object after the modification.
- Performance is degraded as every call to a thread-safe object also invokes the object's internal synchronization. Implementing synchronization outside such calls and using a non thread-safe type of object is usually faster, as synchronization only needs to be executed once in the caller, but not inside every method called on the object.

▾ SonarQube rules
Avoid StringBuffer field

Replace Hashtable With Map

Replace Vector With List

Synchronize iterator-based access to thread-safe collections

When an `Iterator` is used to iterate over a synchronized `Collection` (obtained via `Collections.synchronizedXXXX()`), manually synchronize access to the `Collection`.

Reasoning
An `Iterator` is by definition not thread-safe. So using it together with a synchronized `Collection` means a conflict in concept and won't work as expected.

Locks

Don't use Lock as argument for synchronized

When using `java.util.concurrent.locks.Lock`, use the Lock's `lock()`, `tryLock(long, TimeUnit)` and `unlock()` methods for synchronization. Don't use old style synchronization, now on an instance of `Lock` (e.g. `synchronized(myLock) { ...}`).

Reasoning
Using old fashioned Java synchronization via `synchronized(Object)` on an instance of `Lock` means severe misuse of the API and indicates a fundamental misunderstanding of Java's advanced synchronization concepts.

SonarQube rule
Multithreaded correctness - Synchronization performed on java.util.concurrent Lock

Don't use isLocked() or isHeldByCurrentThread() on ReentrantLock

Do not use the methods `ReentrantLock.isLocked()`or `ReentrantLock.isHeldByCurrentThread()` in productive code.

Reasoning
These methods are intended for debugging or monitoring purposes and should not be used for synchronization control. Usage of these methods in productive code indicates improper locking and thus may lead to race conditions and related multithreading issues.

Use Lock.tryLock() with timeout

When using `Lock` or `ReentrantLock`, use `tryLock(long, TimeUnit)` instead of `tryLock()` to acquire a free lock.

Reasoning
Calling `Lock.tryLock()` or `ReentrantLock.tryLock()`without a timeout does not honor the lock's fairness setting.

Samples
The following sample demonstrates how to use locks instead of `synchronized` blocks. However, the implementation is still optimistic that a free lock can be obtained at all and does not take precautions for deadlock situations:

Lock usage without timeout

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
...
public class MyClass {
    ...
    private final Lock lock = new ReentrantLock();
    ...
    public final void myThreadSafeMethod() {
        ... // Code that does not require synchronization.
        // Acquire lock, possibly waiting forever (see
        // "Lock usage with timeout sample" for a safer implementation):
        this.lock.lock();
        // Execute synchronized code in a try block to ensure proper lock release:
        try {
            ... // Code that requires synchronization on this instance.
        } finally {
            // Reliably release lock in the finally block:
            this.lock.unlock();
        }
        ... // Code that does not require synchronization.
    }
    ...
}

```

A better implementation uses timeouts when attempting to acquire a free lock:

Lock usage with timeout

```java
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import com.bmw.my.package.global.Constants; // The global constants used in my app.
...
public class MyClass {
    ...
    private final Lock lock = new ReentrantLock();
    ...
    public final void myThreadSafeMethod() {
        ... // Code that does not require synchronization.
        try {
            // Acquire lock before timeout:
            if (this.lock.tryLock(Constants.DEFAULT_LOCK_TIMEOUT, TimeUnit.MILLISECONDS)) {
                // Execute synchronized code in a try block to ensure proper lock release:
                try {
                    ... // Code that requires synchronization on this instance.
                } finally {
                    // Reliably release lock in the finally block:
                    this.lock.unlock();
                }
            } else {
                // Error handling in case that lock could not be acquired before timeout:
                ...
            }
        } catch(final InterruptedException e) {
            // Error handling in case that the current thread was interrupted:
            ...
        }
        ... // Code that does not require synchronization.
    }
    ...
}

```

Ensure proper lock release

Ensure that locks are released, also under exceptional conditions by using the `finally` block of `try` statement s to release locks set in the `try` block.

Reasoning

If it is not guaranteed that the lock gets released in any case, the lock may be held forever. This may result in blocking the whole application.

SonarQube rules

Multithreaded correctness - Method does not release lock on all exception paths

Multithreaded correctness - Method does not release lock on all paths

Wait, sleep and notifications

This section discusses rules and best practices on how to synchronize threads by using waits, sleep and notifications.

Waits

Don't read field in loop as a substitute for synchronization

Don't read a field in a loop to synchronize waiting on an event. Use higher-level language features provided by `java.util.concurrent` instead.

Reasoning

Besides the fact that there are far more elegant solutions for this problem than spinning in a loop, this may also produce a severe problem. The compiler may legally hoist the read out of the loop, turning the code into an infinite loop.

SonarQube rule

Multithreaded correctness - Method spins on field

Check wait condition before wait

Before you call `Object.wait()` make sure that the condition you are waiting for is not already satisfied.

Reasoning

If you call `Object.wait()` and the condition you are waiting for is already satisfied, the wait may last forever, block the current thread of execution and lead to unresponsiveness of your application.

SonarQube rule

Multithreaded correctness - Unconditional wait

Don't call Thread.sleep() with a lock held

Do not call `Thread.sleep()` while the current thread of execution holds a lock.

Reasoning

This may result in very poor performance and scalability or even to deadlocks since other threads may be waiting to acquire the lock.

SonarQube rule

Multithreaded correctness - Method calls Thread.sleep() with a lock held

Call wait() with more than one lock judiciously

Call `wait()` with more than one lock judiciously.

Reasoning

Waiting on a monitor while two locks are held may cause deadlock. Performing a wait only releases the lock on the object being waited on, not any other locks. This not necessarily a bug, but is worth examining closely.

SonarQube rule

Multithreaded correctness - Wait with two locks held

Only use monitors for multiple conditions in a loop

When you use `java.lang.Object.wait()`, `java.util.concurrent.locks.Condition.await()` or variants thereof and the monitor is used for multiple conditions, make sure that the wait is placed in a loop.

Reasoning

If the monitor is used for multiple conditions, the condition the caller intended to wait for might not be the one that actually occurred and you have to keep waiting.

SonarQube rules

Multithreaded correctness - Condition.await() not in loop

Multithreaded correctness - Wait not in loop

Monitor-style waits

Don't use naked notifications

When using a monitor for a condition with `wait()` and `notify()` make sure that there exists a heap object that is accessible from both threads and enables the woken up thread to check whether the condition it has been waiting for is now satisfied.

Reasoning

Calling a notify method on a monitor is done because some condition another thread is waiting for has become true. However, for the condition to be meaningful, it must involve a heap object that is visible to both threads. This bug does not necessarily indicate an error, since the change to mutable object state may have taken place in a method which then called the method containing the notification.

SonarQube rule

Multithreaded correctness - Naked notify

Prefer concurrency utilities to wait and notify

As of release 1.5, the Java platform provides higher-level concurrency utilities that do the sorts of things you formerly had to hand-code atop wait and notify. Given the difficulty of using wait and notify correctly, you should use the higher-level concurrency utilities instead.

Reasoning

Using `wait` and `notify` directly is like programming in "concurrency assembly language", as compared to the higher-level language provided by `java.util.concurrent`. There is seldom, if ever, a reason to use wait and notify in new code.

SonarQube rules

Multithreaded correctness - Monitor wait() called on Condition

Using monitor style wait methods on util.concurrent abstraction

Ensure presence of lock for monitor-style waits

When calling `wait()`, `notify()` or `notifyAll()` on an object, make sure that you hold a lock on that instance.

Reasoning

Calling `wait()`, `notify()` or `notifyAll()` without holding a lock will result in a `java.lang.IllegalMonitorStateException` being thrown.

SonarQube rules

Multithreaded correctness - Mismatched notify()

Multithreaded correctness - Mismatched wait()

Don't use IllegalMonitorStateException

Ensure that no `java.lang.IllegalMonitorStateException` ever gets thrown or is caught, except by an application's overall exception handler.

Reasoning

An `IllegalMonitorStateException` is generally only thrown in case of a design flaw in the code when calling `wait()` or `notify()` on an object without holding a lock on the object.

SonarQube rule

Bad practice - Dubious catching of IllegalMonitorStateException

Don't use monitor-style waits on threads

Don't call any of the methods `wait()`, `notify()` or `notifyAll()` on instances of `Thread`.

Reasoning

Doing so will confuse the internal thread state behaviour causing spurious thread wakeups/sleeps because the internal mechanism also uses the thread instance for it's notifications.

Wake-up all threads on change of conditions

Use `notifyAll()` rather than `notify()` when using a monitor for more than one condition.

Reasoning

Calls to `notify()` only wake up one thread, meaning that the thread woken up might not be the one waiting for the condition that the caller just satisfied.

SonarQube rule

Multithreaded correctness - Using notify() rather than notifyAll()

Documentation

This chapter discusses conventions on how to document Java artifacts.

General documentation conventions

General documentation conventions which are independent from concrete Java artifact types.

Javadoc documentation

Rules and conventions on how to apply Javadoc in order to document Java artifacts.

General documentation conventions

This section discusses general documentation conventions which are independent from concrete Java artifact types.

Documentation metrics

Undocumented API

100 percent of all APIs with accessibility public, protected or package private must be documented in Javadoc.

This rule does not apply to

- anonymous inner classes,
- all elements of anonymous classes, independent from their accessibility,
- code that is partially or fully generated by frameworks, design applications or build tools.

Reasoning

Undocumented or partially documented APIs are useless, as the documentation expresses the APIs interface contract required by external users of the artifact. Without such documentation at hand, using the API is hazardous and typically results in errors which are hard to detect and fix.

Mandatory Javadoc documentation is limited to all externally visible artifacts.

▾ SonarQube rules

Javadoc Package

Javadoc Type

Javadoc Variable

Javadoc Method

Comment density

At least 25 percent of each manually written Java source code file must be comments.

▾ Reasoning

Undocumented or poorly documented code is hard to understand and maintain. The minimum threshold of 25 percent is fairly low and reflects the minimum Javadoc documentation. Well documented code should aim for 50 percent comments.

This KPI does not apply to generated code, as this type of code is not subject to maintenance.

▾ SonarQube rule

Insufficient comment density

Language

Documentation language

The documentation language is English.

▾ Reasoning

Documentation should be understandable within the entire BMW Group. External partners, i.e. during development, testing, maintenance and operations, do not necessarily speak the same language as the country that the code is developed for; they still need to understand the code and its documentation. English is the only global language within BMW Group to cross such language barriers.

Documentation can be kept similar or identical if external software libraries are used, as these typically come with APIs and API documentation in English.

Documentation is held free of mixing languages, as it would otherwise be the case if the documentation refers the documentation of external software components, i.e. by using `{@inheritDoc}` for the Javadoc documentati on of overriding methods.

Documentation of incomplete, buggy or empty code

Documentation of open issues

Mark incomplete code sections by inserting a "`// TODO: <`*`open issue`*`>`" comment, where <*`open issue`*> is a placeholder for a meaningful comment on what's left to do.

Mark defective code sections by inserting a "`// FIXME: <bug description>`" comment, where `<bug description>` is a placeholder for a meaningful description of the defect.

▾ Reasoning

Open issues and known bugs can be easily found in the code.

▾ SonarQube rule

Comment pattern matcher

The rule identifies and counts the TODO and FIXME comments, but does not check if each open issue or bug has been documented correctly.

Javadoc documentation

This section discusses rules and conventions on how to apply Javadoc in order to document Java artifacts.

Javadoc format

Javadoc block comment format

Every Javadoc must start with a "`/**`" line and end with a "` */`" line. Each line containing a Javadoc comment start with "` * `" (space + star character). Place an empty "` * `" line between the artifacts description and the documentation of the Javadoc block tags.

Have every sentence of a Javadoc documentation start at a new line.

▾ Reasoning

Adheres to the SUN/Oraccle Javadoc conventions and enhances readability.

Starting every sentence at a new line reduces the effort for reformatting line breaks if something is added to or removed from a Javadoc comment and the comment exceeds the maximum line length.

Sample

Javadoc block comment format

```
/**
 * First sentence of the artifact's description with a masked {@literal .} character;
 * this sentence will appear in Javadoc overviews.
 * Further artifact description, which will not appear in Javadoc overviews.
 *
 * @param   parameter <i>Mandatory.</i>
 *          Parameter documentation.
 * @return  Documentation of the value returned by the method.
 * @throws  IllegalArgumentException
 *          if {@code parameter} is {@code null} or empty.
 */
public ReturnType methodName(final ParameterType parameter) {
    ...
}
```

First sentence of Javadoc comments

The first sentence of the documentation of a Java element must contain a brief description of the artifact. Period characters contained in the first sentence must be masked as `{@literal .}`.

Reasoning

The Javadoc tool uses the period to determine the end of the first sentence. Only the first sentence is displayed in overview Javadoc such as the class or interface list of a Java package.

Unmasked period characters within the first sentence are interpreted as the end of the sentence, resulting in incomplete sentences in Javadoc overviews.

SonarQube rule

Javadoc Style

Javadoc block tag order

Javadoc block tags* must adhere to the following order, as defined by the Javadoc Tool:

- `@author` (packages, classes, interfaces and enum types; required)
- `@version` (packages, classes, interfaces and enum types; required)
- `@param` (methods, constructors and generic types)
- `@throws` (methods and constructors; `@exception` is not used)
- `@see` (all elements; optional)
- `@since` (all elements; optional)
- `@serial` or `@serialField` or `@serialData`
- `@deprecated` (all elements)

*Block tags initiate Javadoc comments with a predefined semantics. They typically appear on the very left side of Javadoc code, with the Javadoc comment starting right beside the block tag. In contrast to inline tags, block tags never come with curly brackets.

Reasoning

Adheres to the SUN/Oracle Javadoc conventions.

Documentation of deprecation

Deprecated elements must be documented with a `@deprecated` block tag. Document how the method's functionality is to be accessed in a non-deprecated way.

> The `@deprecated` tag can be used with types, fields, enum values and methods, but not with Javadoc of packages.

Reasoning

Enables users of the code to prepare for avoiding or removing code that is no longer intended to be used. This reduces refactoring efforts when a deprecated element disappears in a future software version.

SonarQube rule

Missing Deprecated

Don't nest Javadoc inline tags

Javadoc inline tags* must not be nested.

*Inline tags are tags which are used inside Javadoc comments, like `{@code ...}`, `{@inheritDoc}`, `{@link ...}`, `{@linkplain ...}`, `{@literal ...}` or `{@value ...}`. In contrast to block tags, inline tags are always surrounded by curly brackets.

Reasoning

The Javadoc tool does not support nesting of inline tags:

A nested Javadoc comment like "`The font size must be greater than or equal {@code {@value #MIN_FONT_SIZE}}.`" would not produce the desired Javadoc: Instead of "The font size must be greater than or equal `10`." if `10` is the value of `MIN_FONT_SIZE`, the generated Javadoc would be "The font size must be greater than or equal `{@value #MIN_FONT_SIZE}`."

Use fully qualified type names in Javadoc links

Links in Javadoc comments must use fully qualified type names except for types in the `java.lang` package.

Reasoning

Links to other types can appear in the `@see` Javadoc block tag and in Javadoc inline tags `{@link ...}`, `{@linkplain ...}` and `{@value ...}`. The Javadoc tool allows using unqualified type names in Javadoc comments for imported types, types in the `java.lang` package and types in the same package as the documented artifact.

To keep dependencies of the code to other types clean from dependencies which are only required for documentation purposes, import statements must not contain types which are only required by Javadoc (see *Don't import unused types*). This is supported by generally using fully qualified type names in Javadoc, except for `java.lang`. Javadoc is kept independent from changes in the code, as removing import statements doesn't affect the Javadoc.

Documentation of packages

Javadoc of packages

Each package must contain a file `package-info.java`, which holds the Javadoc documentation of the package's content. An HTML documentation in `package.html` as in Java versions prior to 1.5 is no longer supported.

The package's Javadoc documentation must contain the following elements in this order:

- Description:
  A short description of the package's content and purpose.
- `@author`:
  The name of the author of the latest package version.
- `@version`:
  The current version of the package, starting with `1.0.0`.

Reasoning

Provides a general overview of the package, which is also displayed in the package's Javadoc created by the Javadoc Tool.

Sample

Package documentation (package-info.java)

```
//======================================================================================================================
// Module: BMW Continuous Integration installer - Utilities.
// Copyright (c) 2011-2014 BMW Group. All rights reserved.
//======================================================================================================================
/**
 * A collection of utilities for the Continuous Integration custom application build and quality assurance environment.
 * <p/>
 * <ul>
 *   <li>
 *     Class {@linkplain com.bmw.shared.it4it.contint.util.LogoCreator} implements the functionality to
 *     create customized logos for the Continuous Integration server components Jenkins, Nexus and SonarQube.
 *     <p/>
 *   </li>
 *   <li>
 *     ...
 *   </li>
 * </ul>
 *
 * @author   Markus Dieterle <markus.dieterle@bmw.de>
 */
package com.bmw.shared.it4it.contint.util;

```

SonarQube rules

Javadoc Package

Package Annotation

Documentation of types

Javadoc of types

Every named type with accessibility public, protected or package private must be documented in Javadoc. Anonymous inner types don't require Javadoc documentation.

The type's Javadoc documentation must contain the following elements in this order:

- Description:
  A short description of the type's meaning and purpose.
- `@author`:
  The name of the author of the latest version of the type.
- `@version`:
  The current version of the type, starting with `1.0.0`.
- `@param`:

The parameter of a generics type; only required for generics.
- `@deprecated`:
  The reason why the type is deprecated and what is to be used instead. Only required if the type should no longer be used. If a type is documented as being deprecated, is must be tagged with the `@Deprecated` a nnotation and vice versa.

Reasoning

A type's API should be fully documented, since it cannot be reasonably used and maintained without proper documentation.

Sample

Type documentation

```
/**
 * Stand-alone tool to create customized logos for the Continuous Integration
 * server components Jenkins, Nexus and SonarQube.
 *
 * @author   Markus Dieterle <markus.dieterle@bmw.de>
 */
public final class LogoCreator {
    ...
}

```

SonarQube rule

Javadoc Type

Javadoc for type parameters of generics

Document the type parameters of generics, using the generic Javadoc's `@param` tag.

Reasoning

Facilitates correct usage of the generic type.

SonarQube rule

Javadoc Type

Documentation of variables

Javadoc of variables

Every static or member variable except variables of anonymous inner types must be documented in Javadoc. Local variables don't require documentation.

Reasoning

A type's API must be fully documented. Documentation of private variables enhances maintenance.

Sample

Variable documentation

```
/**
 * The logo type.
 */
private final LogoType m_logoType;
```

SonarQube rule

Javadoc Variable

Documentation of methods

Javadoc of constructors and methods

Every constructor or method with accessibility public, protected or package private must be documented in Java doc. Methods of anonymous inner types don't require Javadoc documentation.

The method's Javadoc documentation must contain the following elements in this order:

- Description:
  A short description of the variable's content.
- `@param <`*`parameter name`*`>`:
  For each parameter: A short description of the parameter's semantics and preconditions, e.g. if the parameter accepts `null` values or comes with other restrictions regarding its values.
  Mark non-nullable parameters as mandatory, e.g. by placing `<i>Mandatory.</i>` right next to the parameter name.
  If the parameter is unused by the method's implementation document this fact and, if the method overrides a method where the parameter is unused, document whether it is still unused by this method's implementation.
  To enhance readability, use a new line for the parameter documentation, with the first letter of the parameter documentation being immediately underneath the first letter of the `<`*`parameter name`*`>`.
  There is no `@param` documentation if the method's signature does not contain parameters.
- `@return`:
  The documentation of the method's return value.
  There is no `@return` documentation if the method returns `void` or is a constructor.
- `@throws`:
  The exceptions thrown by the method and the conditions for throwing the exceptions.
  Use the `@throws` tag and not the `@exception` tag to document exceptions.
- `@deprecated`:
  The reason why the method is deprecated and what is to be used instead. Only required if the method should no longer be used. If a method is documented as being deprecated, is must be tagged with the `@Deprecated` annotation and vice versa.

Reasoning

A type's API must be fully documented. Documentation of private constructors and methods enhances maintenance.

Sample

Method documentation

```java
/**
 * Gets the width of a given text rendered for a given font.
 *
 * @param   text <i>Mandatory.</i>
 *          The text to calculate its width for.
 * @param   font <i>Mandatory.</i>
 *          The font to be used for rendering the {@code text}.
 * @return  the width in pixels of the specified {@code text} for the specified {@code font}.
 * @throws  IllegalArgumentException
 *          if the specified {@code text} or {@code font} are {@code null}.
 * @see     java.awt.FontMetrics#stringWidth(String)
 */
public static int getTextWidth(final String text,
                               final Font   font) {
    ...
}
```

SonarQube rule

Javadoc Method

Document when method returns null

For methods that don't return `void` or a primitive, document under which conditions the method may return `null`.

Reasoning
Aids in interpreting the returned value correctly and avoid NullPointerExceptions. When returning a wrapper type for a primitive, the method can be invoked as though it returned a primitive value, and the compiler will insert automatic unboxing of the wrapper value. If a `null` value is returned, this will result in a `java.lang.NullPointerException`.

Document thrown runtime exceptions

Document the runtime exceptions thrown by the method and under which conditions they are being thrown.

Reasoning
Runtime exceptions shouldn't be declared in the `throws` clause of a method declaration, but are important to understand the method's behavior. The Javadoc is a much better place for documentation than the `throws` clause would be.

Document subtypes of thrown exceptions

If a method can throw both a supertype and a subtype instance of an exception, document both the supertype as two different `@throws` entries.

Reasoning
Facilitates to better understand the method's behavior and implement a more fine-grained exception handling.

Document thrown exceptions in alphabetical order

Document the exception's thrown by the method in alphabetically ascending order.

Reasoning
Facilitates verification if a method throws an exception of a specific type. This can be useful for the documentation of the exceptions that can be thrown by a method that uses another method that throws exceptions.

Formatting

This chapter discusses conventions on how to layout and structure Java source code.

General formatting issues

General layout and formatting issues like array formats, declaration formats, modifier orders and numeric formats.

Line formatting and whitespaces

Discusses how to format lines of code and how to use braces and whitespaces.

General formatting issues

This section discusses general layout and formatting issues like array formats, declaration formats, modifier orders and numeric formats.

General file format

Use UTF-8 character encoding

Text files must be encoded in UTF-8.

Reasoning
Increases portability across development and runtime environments. Using development platform specific encodings leads to misinterpretations of non-ASCII characters on other development platforms and on the runtime environment.

Setting the character encoding in Eclipse
In Eclipse, the default character encoding is specific to the operating system, which is code page 1252 on Windows.

> In the *Window/Preferences/General/Workspace* dialog, change *Text file encoding* to *UTF-8*.

Use Unix-style line delimiter

The line delimiter of text files must be Unix-style (just line feed, ASCII character 10).

Reasoning

Unix-style line delimiters are well supported by all common development environments.

Setting the line delimiter in Eclipse

In Eclipse, the default line delimiter is specific to the operating system, which is *carriage return* + *line feed* (ASCII characters 13 (CR) + 10 (LF)) on Windows, *line feed* on Linux and *carriage return* on OS X.

In the *Window/Preferences/General/Workspace* dialog, change *New text file line delimiter* to *Unix*.

Insert newline at end of file

Text files must be terminated with a newline.

Reasoning

Due to misinterpretations of what is the end of file, some tools ignore source code which is contained in the last line. Some SCM systems even require files to end with a newline character and print a warning when encountering a file that doesn't end with a newline. Placing a newline at the end of the file prevents such errors, which are otherwise hard to detect.

SonarQube rule

Newline At End Of File

Organize files according to Oracle conventions

Organize the elements of Java class files as recommended in the Oracle file organization conventions.

Reasoning

The structure is easy to read and widely used. Grouping of semantically closely related methods enhances readability.

SonarQube rules

Declaration Order

Inner Type Last

Array format

Don't use C-style for array definitions

Use Java-style array definitions rather than C-style definitions.

Reasoning

Java-style array definitions clearly separate the type from the identifier and are the de-facto standard for Java programming.

Samples

C-style array definition

```java
public static void main(final String args[]) {
    ...
}
```

Java-style array definition

```java
public static void main(final String[] args ) {
    ...
}
```

SonarQube rule

Array Type Style

Place comma after last array element

In array initializations, place a training comma after the last element of the array.

Reasoning

This facilitates adding new elements to the array and is encouraged by the Java language specification.

SonarQube rules

Annotation Use Style

Array Trailing Comma

Declarations and modifiers

Align type declarations

Align type literals in type declarations. Place the opening brace at the end of the line which contains the last element of the type declaration.

Reasoning

Enhances code readability.

Samples

Interface declaration

```java
public interface MyInterface
extends          MySuperInterface, MyInterface2 {
...
```

Class declaration

```java
public class MyClass
extends      MySuperclass
implements   MyInterface1, MyInterface2 {
...
```

Declare thrown exceptions in alphabetical order

In method declarations, declare the exceptions thrown by the method in alphabetical order.

Reasoning

Enhances code readability and facilitates maintenance.

Adhere to Java modifier order

The order of modifiers must conform to the suggestions in the Java Language specification, sections 8.1.1, 8.3.1

and 8.4.3. The correct order is:

- `public`
- `protected`
- `private`
- `abstract`
- `static`
- `final`
- `transient`
- `volatile`
- `synchronized`
- `native`
- `strictfp`

Reasoning
A unified code formatting makes reading and understanding source code easier especially if comparing revisions. To some extent, this behavior can be automated with a correctly configured code formatter.

SonarQube rule
Modifier Order

Import format

Import order

Place one import statement per line.

Order imports by groups:

- Java standard types: `java.*` / `javax.*`
- Types from public organizations: `org.*`
- Other non BMW types: `com.`<*`not bmw`*>`.*` / etc.
- Types from BMW packages: `com.bmw.*`

Separate import groups by a newline.

Maintain an alphabetically ascending order within each group.

Reasoning
Facilitates to find the detailed package information for a used type, and to understand the dependencies to other types.

SonarQube rule
Import Order

Numeric formats

Use upper case L for literals of type long

The trailing 'L' of literals of type `long` must be capitalized – and not be the lowercase 'l'.

Reasoning
Depending on the font used by code editors or when printing, the lowercase L 'l' can hardly be distinguished from the digit one '1', which easily results in misinterpretations of the code.

Sample

Upper L

```java
// Replace ...
long unitSize = 1l; // Likely to be read as int 11.
```

```java
// ... by:
long unitSize = 1L; // Clearly readable as long 1.
```

SonarQube rule
Upper Ell

Don't use octal values

If an octal value is required, use the decimal representation in the code and add a comment with the octal representation.

Reasoning
The representation of octal values is the same as for `int` values, except a leading `0`: octal `0172` equals decimal `122`. This makes an octal value not being very obvious and easily misinterpreted as a decimal value.

SonarQube rule
Avoid Using Octal Values

Don't use octal escape sequences for String literals

Use decimal or hexadecimal rather than octal escape sequences for String literals.

Reasoning
Errors in octal escape sequences are easily overseen, as e.g. in `"\038"`: The `"8"` is not an octal number. Thus, the literal is interpreted as the octal escape sequence `"\03"`, followed by the literal character `"8"`.

SonarQube rule
Suspicious Octal Escape

if ... else statements

If statements require braces

Surround the code of the branches of an `if` statement by curly braces, even if the blocks only consist of a single statement.

Reasoning
Prevents logical errors when more statements are added to the block and increases the readability of source code.

SonarQube rule
Need Braces

Line formatting and whitespaces

This section discusses how to format lines of code and how to use braces and whitespaces.

Line formatting

Max. 120 characters per line

The maximum line length is 120 characters with a tab width of 4 spaces.

Reasoning
120 characters/line is more adequate with respect to wide-screen resolutions and enhances readability. A tab width of 4 wastes less space than Oracle's recommendation of 8 without loss off code readability.

SonarQube rule
Line Length

Place operator of wrapped expression at beginning of new line

If statements span multiple lines, place the operator at the beginning of the new line rather than at the end of the line for all non-assignment operators.

Reasoning
Enhances code readability, since the statement can easily be identified as being wrapped.

SonarQube rule
Operator Wrap

Line must not contain more than one statement

Each statement is to be written on its own line.

Reasoning

Statements that are written on the same line as another statement may easily be overlooked.

SonarQube rule

One Statement Per Line

Only declare one variable per line

Place each variable declaration into a separate statement.

Reasoning

Enhances code readability.

SonarQube rule

Multiple Variable Declarations

Position left curly braces at end of line

Opening curly braces must be at the end of the line.

Reasoning

A uniform code style enhances code readability. Placing opening braces at the end of the line is the de-facto standard for Java, whereas placing it at the beginning of a new line is typical for programming languages like C/C++.

Sample

Left curly braces

```java
public void checkName(final String name) {
    if ((name == null) || (name.isEmpty())) {
        throw ...;
    }
}
```

SonarQube rule

Left Curly

Position right curly braces on a new line

Place the closing brace of a block on a new line.

In case of an `if` statement, the closing brace of the `if...then` block is followed by the `else` keyword if the statement contains an `else` branch.

In case of a `try` statement, the closing brace of the `try` block is followed by the `catch` or by the `finally` key word. The closing brace of the `catch` block may only be followed by the `finally` keyword.

Reasoning

Clearly visualizes the code block structure. Placing the `else`, `catch` and `finally` keywords on the same line as the closing curly brace of the preceding block avoids unnecessary waste of space without reducing code readability.

Sample

Right curly braces

```
try {
    if (...) {
        ...
    } else if (...) {
        ...
    } else {
        ...
    }
} catch (...) {
    ...
} finally {
    ...
}
```

SonarQube rule

Right Curly

Whitespaces

Avoid trailing spaces

Remove trailing spaces at end of lines.

Reasoning

Superfluous invisible characters may disturb code visualizations, e.g. when printing or displaying the code in a plain text editor.

SonarQube rule

Regexp Singleline

Avoid confusing preceding whitespaces

Don't place a whitespace before a semicolon '`;`' or a post-increment '++' or post-decrement '`--`' operator.

Reasoning

Such whitespaces are just confusing.

Sample

Confusing preceding whitespace

```
// Replace such confusing preceding whitespaces ...
x = i_++;
y = j
--;

// ... by better readable:
x = i++;
y = j--;
```

SonarQube rule

No Whitespace Before

Avoid confusing trailing whitespaces

Don't place a whitespace after a dot or a pre-increment '++', pre-decrement '--' or unary plus '+' or minus '-' operator.

Reasoning

Such whitespaces are just confusing.

Sample

Confusing trailing whitespace

```
// Replace such confusing trailing whitespaces ...
if (x._equals(y)) {...}
x = -_3;
y = ++_i;
z = --
j;
```

```
// ... by better readable:
if (x.equals(y)) {...}
x = -3;
y = ++i;
z = --j;
```

SonarQube rule

No Whitespace After

Place whitespace after comma, semicolon and typecast

Place a whitespace character after commas, semicolons and typecasts.

Reasoning

Enhances code readability.

SonarQube rule

Whitespace After

Surround curly braces, control flow literals and non-unary operators by blanks

Place a whitespace character left and right of:

Surround by whitespace	
curly braces	`{`, `}`
literals	`assert`, `catch`, `do`, `else`, `finally`, `for`, `if`, `return`, `synchronized`, `try`

non-unary operators	arithmetic:
	`+`, `-`, `*`, `/`, `%`
	
	arithmetic assignment:
	`+=`, `-=`, `*=`, `/=`, `%=`
	
	assignment:
	`=`
	
	bit shift:
	<<, >>, >>>
	
	bit shift assignment:
	<<=, >>=, >>>=
	
	bitwise:
	&, `|`, ~, ^
	
	bitwise assignment:
	&=, `|=`, ~=, ^=
	
	conditional:
	&&, `||`, `?:`
	
	equality:
	==, `!=`
	
	relational:
	<, <=, >, >=

'&' symbol when used in a generic upper or lower bounds constraint	

Note that unary operators `++`, `--`, `+`, `-` and `!` are exempted from this rule.

Reasoning

Enhances code readability. For unary operators this rule would be counter productive, as in "`int i = - 3;`" for the unary '`-`'.

SonarQube rule

Whitespace Around

Don't separate method identifier from parameter list

Don't place whitespaces or line breaks between a method's identifier and the left parenthesis of the method's parameter list.

Reasoning

A uniform code style enhances code readability. Keeping method names and method parameter lists together is the de-facto standard for Java.

Sample

Method parameter padding

```java
// Replace ...
public static boolean renameFile_(final String oldPathname,
                                  final String newPathname) {
    return new File_(oldPathname).renameTo_(new File_(newPathname));
}
```

```java
// ... by:
public static boolean renameFile(final String oldPathname,
                                 final String newPathname) {
    return new File(oldPathName).renameTo(new File(newPathname));
}
```

SonarQube rule
[Method Param Pad]

Don't surround parameter lists by whitespaces

Don't place whitespaces or line breaks after the opening brace or before the closing brace of a method's parameter list.

Reasoning
A uniform code style enhances code readability. Avoiding whitespaces around method parameter lists is the de-facto standard for Java.

SonarQube rule
[Paren Pad]

Don't surround type casts by whitespaces

In type casts, don't place whitespaces or line breaks between the type identifier and the type cast's enclosing braces.

Reasoning
A uniform code style enhances code readability. Avoiding whitespaces around type casts saves space without reducing code readability.

SonarQube rule
[Typecast Paren Pad]

Don't surround types of generics by whitespaces

Don't place whitespace around the generic tokens < and > which embrace the generic's type identifier.

Reasoning
Such whitespaces don't increase code readability at the cost of wasted space.

Sample

Generics format

```java
// Replace ...
List<_Integer_> x = new ArrayList<_Integer_>();
List<List_<Integer>_> y = new ArrayList<_List<Integer>_>();
```

```java
// ... for Java 5 and 6 by ...:
List<Integer> x = new ArrayList<Integer>();
List<List<Integer>> y = new ArrayList<List<Integer>>();
```

```java
// ... and for Java 7 and higher by:
List<Integer> x = new ArrayList<>();
List<List<Integer>> y = new ArrayList<<>>();
```

SonarQube rule
Generic Whitespace

Naming

This chapter discusses rules and conventions on how Java artifacts are to be named.

General naming conventions

Naming conventions which are independent from concrete Java artifact types.

Type names

Naming conventions for classes, interfaces, enumerations and exceptions.

Field names

Naming conventions for constants, static fields, members, local variables, loop indices, catch clause parameters and annotation fields.

Method names

Naming conventions for methods and method parameters.

General naming conventions

This section discusses naming conventions which are independent from concrete Java artifact types.

- Language
  - *Adhere to the Code Conventions for Java*
  - *Name language*
- Problematic names
  - *Avoid Java keywords*
  - *Avoid dollar signs in identifiers and type literals*
- Naming patterns
  - *Design patterns*
  - *Project-specific naming conventions*
- Packages

- *Package name*

Language

Adhere to the Code Conventions for Java

Java source code must follow the Code Conventions for the Java Programming Language from Oracle.

▾ Reasoning

Code conventions are important for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

▾ SonarQube rules

Correctness - Method defines a variable that obscures a field

Local Final Variable Name

Local Variable Name

Member Name

Naming - Suspicious constant field name

Static Variable Name

Name language

All Java artifacts must be named in *American English*, e.g. a method returning the normalization status of a customer address should be named `isNormalized()`, but not `isNormalised()` (British English), `istNormalisiert()` (German) or other translations of the same.

▾ Reasoning

Code should be understandable within the entire BMW Group. External partners, i.e. during development, testing, maintenance and operations, do not necessarily speak the same language as the country that the code is developed for; they still need to understand the code. English is the only global language within BMW Group to cross such language barriers.

Identifier names can be kept similar or identical if external software libraries are used, as these typically come with APIs in American English. Using American English over British English can avoid programming errors like incorrect or unintentional overriding of methods. Code is held free of mixing languages, as core Java APIs anyway require using artifacts with American English identifiers, thus affecting the entire code.

Problematic names

Avoid Java keywords

Do not use identifiers which are keywords in later versions of Java.

▾ Reasoning

Maintains the upward compatibility of your code to later versions of Java. Otherwise, the code will need to be changed in order to compile it in later versions of Java.

▾ SonarQube rules

Bad practice - Use of identifier that is a keyword in later versions of Java

Bad practice - Use of identifier that is a keyword in later versions of Java (member)

Avoid dollar signs in identifiers and type literals

Do not use dollar "$" signs in identifier names and type literals.

▾ Reasoning

Java uses the "$" sign for .class files as a separator between outer and inner class names. Thus, using "$" signs in identifier names is confusing.

▾ SonarQube rule

Naming - Avoid dollar signs

Naming patterns

Design patterns

When implementing a design pattern, try to adhere to that pattern's naming conventions.

▾ Reasoning

Other programmers can easily recognize the objects affected by the design pattern and their semantics.

Project-specific naming conventions

Define project-specific rules on how to compose artifact names, depending on the type and design of your application. For instance, if the design pattern "DataTransferObject" is applied, all corresponding classes should be marked with a consistent suffix, e.g. "TO". This should be equally applied to other patterns used in the code.

▾ Reasoning

Technical design patterns can easily be recognized in the code. The code becomes more self-explainable and can easier be maintained and enhanced.

Packages

Package name

Package names for productive code must begin with `com.bmw.`. See *Test package name* on how to name test code packages.

Reasoning

Code owned by BMW should be easily distinguishable from third-party code.

SonarQube rule

Package name

Type names

This section discusses naming conventions for classes, interfaces, enumerations and exceptions.

Types

Use camel case notation for types

Type (class, interface, enumeration or annotation) names are camel case, consisting of letters or digits, and must begin with an upper case letter.

Reasoning

Complies to Oracles "Code Conventions for the Java Programming Language".

SonarQube rule

Type Name

Shadowing type name

Type names must not shadow names of supertypes or implemented interfaces.

Reasoning

Avoids confusion on which class is being used and reduces risks from using syntactically ambiguous type names.

SonarQube rules

Bad practice - Class names shouldn't shadow simple name of implemented interface

Bad practice - Class names shouldn't shadow simple name of superclass

Enterprise bean name

JEE applications must adhere to the Component Architecture (CA) naming conventions for Enterprise Beans.

▾ Reasoning
This facilitates code maintenance, more readability and avoids conflicts with code generation tools.

▾ SonarQube rules
Local Home Naming Convention

Local Interface Session Naming Convention

Message Driven Bean And Session Bean Naming Convention

Remote Interface Naming Convention

Remote Session Interface Naming Convention

Generics

Generics type parameter name

Names of generics parameters should be a single upper case letter, which represents the first letter of the generics parameter's semantics, e.g. K for the key and V for the value generic.

▾ Reasoning
Established convention for Java; also the JDK strictly adheres to this convention.

▾ SonarQube rules
Class Type(Generic) Parameter Name

Method Type(Generic) Parameter Name

Exceptions

Use a correct exception name

A type name should end in "Exception" if and only if the type is a subtype of `java.lang.Exception`.

▾ Reasoning
Exception types can easily be identified by their names.

▾ SonarQube rule
Bad practice - Class is not derived from an Exception, even though it is named as such

Field names

This section discusses naming conventions for constants, static fields, members, local variables, loop indices, catch clause parameters and annotation fields.

- Constants
    - *Constant name*

Constants

Constant name

Constant names are all upper case and may include digits and underscores; the first character must be an upper case letter, e.g. `MY_CONSTANT`. This also applies to the names of enum values.

▾ Reasoning
Better readability; no need to deviate from the default Oracle Java coding conventions.

Although defined with a different syntax, `enum` values can also be seen as constants: Each value is an instance of the enum type. Naming `enum` values like constants has also been applied to the JDK.

▾ SonarQube rule
Constant Name

Catch clause parameters

Catch clause parameter name

Catch clause parameters can be named `e` or `t`, optionally followed by a single digit.

▾ Reasoning
`e` for Exceptions and `t` for Throwables are common one-letter identifiers in catch clauses.

Annotation field definitions

One-argument annotation field name

The field of an annotation that just contains a single field definition must be named `value`.

▾ Reasoning
This is the default convention and allows using such annotations without explicitly stating the field name in the code: `@MyAnnotation(value="myValue")` can be shorter expressed as `@MyAnnotation("myValue")`.

Shadowing

Use `this` to avoid field shadowing

To resolve ambiguity it is required to use the this keyword.

▾ Reasoning

The `this` keyword is mandatory to resolve the ambiguity. Without `this`, the compiler cannot know whether we are assigning the *count* method argument value to itself.

▾ SonarQube rule

Require This

Method names

This section discusses naming conventions for methods and method parameters.

Methods

Method name

Method names (except constructors) must begin with a lower case letter, followed by a camel case character sequence, consisting of letters or digits.

See *Test method name* on how to name methods of unit tests.

▾ Reasoning

Complies to standard Oracle "Code Conventions for the Java Programming Language".

▾ SonarQube rule

Method Name

Avoid enclosing type name as method name

Method names must be different from the name of their enclosing type.

▾ Reasoning

To avoid confusion and detect malformed constructors, only constructors have the same name as the enclosing type.

▾ SonarQube rule

Correctness - Apparent method/constructor confusion

Avoid method names with minimal name variations

Method names may not differ only by capitalization or other minor variations of method names. This also applies variations of method names used in supertypes or in implemented interfaces.

Reasoning

Avoid confusion from misleading or ambiguous method names. Variations of existing superclass method names appear to override the superclass method, but don't. This is a clear indication for unintended or unexpected runtime behavior.

A method named `equal(...)` indicates a failed attempt to override `java.lang.Object.equals(Object)`, resulting in unexpected runtime behavior.

SonarQube rules

Bad practice - Confusing method names
Rule refers to the method signature.

Bad practice - Very confusing method names (but perhaps intentional)
Rule refers to references to the ambiguous method.

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

Correctness - Very confusing method names
Rule refers to a method that would otherwise override a method which differs only by capitalization.

Naming - Suspicious equals method name
Rule refers to a method with a parameter and name variation of `equals(Object)`.

Naming - Suspicious Hashcode method name
Rule refers to a method with a parameter and name variation of `hashCode()`.

Boolean getter name

Getters which return a boolean value should be prefixed `is` or `has`, e.g. `isX()` instead of `getX()`. If the referred state is in the past, the prefix should be `was` or `had`, resp.

Reasoning

Enhances readability of code.

Method parameters

Parameter name

Parameter names must begin with a lower case letter, followed by a camel case character sequence, consisting of letters or digits.

Reasoning

Complies to standard Oracle "Code Conventions for the Java Programming Language".

SonarQube rule

Parameter Name

Parameter name in overriding method

Parameters of overriding methods must have the same name as in the overridden method.

Reasoning

Using the same parameter names in overridden and overriding methods makes it easier to understand the overriding code. If the JavaDoc of the overridden method can be shared by using `{@inheritDoc}`, documenta tion elements of the overridden type which refer to a parameter by name still refer to the right parameter. When overriding standard Java methods or methods from external libraries, the parameter names of overriding methods can be the same as of the overridden methods.

Performance

This chapter discusses programming rules with impact on the performance of applications written in Java.

General performance issues

Miscellaneous rather general performance related rules.

Efficient usage of arrays and collections

Rules and best practices on how to use arrays and collections in an efficient way.

Efficient object allocation and type conversions

Rules and best practices on how to allocate objects and convert between types in an efficient way.

Efficient iteration and loops

Rules and best practices on how to implement efficient repetitive application code.

Efficient string handling

Rules and best practices on how to process strings in an efficient way.

General performance issues

This section contains miscellaneous rather general performance related rules.

- Accessing static members
    - *Don't create objects to call static methods*
- Constants
    - *Share big constants*
    - *Avoid math operations on constants*
- Garbage collection
    - *Don't invoke garbage collection*
- Nested classes
    - *Favor static member classes over nonstatic*
- Optimization of user interface behavior
    - *Don't block the event dispatching thread (EDT)*
- URL processing
    - *URL's equals() and hashCode() may be slow*

Accessing static members

Don't create objects to call static methods

Don't create an object just to call a static method. In general even try to avoid calling static methods on objects, unless it enhances readability.

Reasoning
Static methods do not need available instances of the class in which they are defined, therefore getting those instances is superfluous code and should be replaced by a direct call on the given class. If the class is just instantiated to access the member, an unnecessary object must be allocated and garbage collected.

Constants

Share big constants

Big constants like String literals should be centralized and referenced between classes rather than be redefined in multiple classes.

Reasoning
Maintenance is improved due to elimination of redundancies. Furthermore this will reduce the size of the compiled byte code.

SonarQube rule
Performance - Huge string constants is duplicated across multiple class files

Avoid math operations on constants

Avoid math operations on constants. Instead, define the result of the math operation on a constant as another constant.

Reasoning
Math operations that process constant values should not be executed several times, but once if at all. Occurrences of this pattern should be made `static final`.

SonarQube rule
Performance - Method calls static Math class method on a constant value

Garbage collection

Don't invoke garbage collection

The `GarbageCollector` must not be called explicitly.

Reasoning
Garbage collection is best performed under control of optimized algorithms in the JVM.

SonarQube rule

Performance - Explicit garbage collection; extremely dubious except in benchmarking code

Nested classes

Favor static member classes over nonstatic

If possible, nested classes should declared as `static` nested classes.

Reasoning

If an nested class does not use its embedded reference to the object which created it except during construction of the inner object, it should be declared as a static nested class. This reference makes the instances of the class larger, and may keep the reference to the creator object alive longer than necessary.

SonarQube rules

Performance - Could be refactored into a static inner class

Performance - Should be a static inner class

Optimization of user interface behavior

Don't block the event dispatching thread (EDT)

Time consuming operations should not be performed in the event dispatching thread.

Reasoning

If time consuming operations are performed in the EDT, the UI is blocked till the operation has finished. This has a severe impact on the perceived performance of the user interface. If you need to execute a long-running task, you should usually create a separate worker thread, also known as background thread.

URL processing

URL's equals() and hashCode() may be slow

URL's `equals()` and `hashCode()` may employ DNS calls to resolve the IP addresses. This might take quite a while. Consider if `equals/hashCode` really needs to include information of resolved IP addresses.

Reasoning

The equals and hashCode method of URL perform domain name resolution, this can result in a big performance hit. Consider using `java.net.URI` instead.

SonarQube rule

Performance - The equals and hashCode methods of URL are blocking

Efficient usage of arrays and collections

This section discusses rules and best practices on how to use arrays and collections in an efficient way.

- Collection type

Collection type

Use simple rather than thread-safe collections

If used outside the context of multithreading, use non thread-safe collection types rather than thread-safe types.

Replace	with
`java.util.Hashtable`	`java.util.HashMap`
`java.util.Vector`	`java.util.ArrayList`

Reasoning
Thread-safe collection types have to synchronize most access methods. This comes with a hit on performance.

See Thread-safety on how to use simple collection types in a multithreaded environment.

SonarQube rules
Replace Hashtable With Map

Replace Vector With List

Use EnumMap or EnumSet in case of enum types

Use `java.uti.EnumMap` and `java.util.EnumSet` rather than ordinary Maps or Sets when the class used as the key type is an enum class.

Reasoning
EnumMaps and EnumSets yield superior performance over standard Collections. Furthermore, they offer some helpful methods to deal with Enums.

Collection key type

Don't use URLs as keys for Maps and Sets

Do not use `java.net.URL` as the key type for Maps and Sets; use `java.net.URI` instead.

Reasoning
Maps and Sets rely on the key type's `equals()` and `hashCode()` methods for comparisons and lookups. Since these methods perform blocking name server lookups to determine a normalized URL for instances of `jav`

`a.net.URL`, this can pose a critical performance issue.

▾ SonarQube rule
Performance - Maps and sets of URLs can be performance hogs

Using collections and arrays

Don't copy arrays manually

Copy array through one of the `java.lang.System.arraycopy()` or `java.util.Arrays.copyOf()` meth ods.

▾ Reasoning
The supplied methods are well tested and have a superior performance due to their native implementation.

▾ SonarQube rule
Avoid Array Loops

Pass correctly sized array to toArray()

The `toArray()` method of Collections should be supplied with a correctly sized target array.

▾ Reasoning
When `toArray()` is supplied with a too small array, then another array of the correct size has to be created using reflection, which is time consuming.

▾ SonarQube rule
Performance - Method uses toArray() with zero-length array argument

Use containsKey() rather than keySet().contains()

Use the `containsKey(<key>)` method rather than invoking `keySet().contains(<key>)` to check if a collection contains an element for a given key.

▾ Reasoning
Depending on the map implementation, calling `keySet()` may result in additional computation and object creation and garbage collection of a temporary set of the collection's keys.

Efficient object allocation and type conversions

This section discusses rules and best practices on how to allocate objects and convert between types in an efficient way.

BigInteger allocation

Don't create copies of BigInteger constants

Don't create copies of `BigInteger` constants. Use the constants instead.

Reasoning
BigIntegers are immutable and can therefore be used directly without the need for clones.

SonarQube rule
Big Integer Instantiation

Class object allocation

Don't create objects to get the Class object

Retrieve a class's Class object by using its `.class` property. Don't create intermediate instances of the class and then invoke its `getClass()` method.

Reasoning
Creating a new instance just for calling `getClass()` is superfluous and unnecessarily clutters up the heap memory.

SonarQube rule
Performance - Method allocates an object, only to get the class object

Wrappers for primitives

Don't box boolean constants

Use `Boolean.TRUE` and `Boolean.FALSE` rather than creating new instances of `Boolean`.

Instead of	use
`new Boolean(true)`	`Boolean.TRUE`
`new Boolean(false)`	`Boolean.FALSE`

Reasoning
The `java.lang.Boolean` class contains wrapper constants for `true` and `false`. Using these constants avoids the overhead of creating new objects and improves garbage collection.

Obtain wrappers for primitives through factory methods

Instances of boxed types should be created from literals through the boxed type's `static valueOf(...)` fact ory methods, rather than calling the constructor.

Instead of	use
`new Boolean(<boolean expression>)`	`Boolean.valueOf(<boolean expression>)`
`new Byte(byte)`	`Byte.valueOf(byte)`
`new Character(char)`	`Character.valueOf(char)`
`new Double(double)`	`Double.valueOf(double)`
`new Float(float)`	`Float.valueOf(float)`
`new Integer(int)`	`Integer.valueOf(int)`
`new Long(long)`	`Long.valueOf(long)`
`new Short(short)`	`Short.valueOf(short)`

Reasoning
Wrapper objects for primitives are immutable, which makes them sharable without side effects. The JVM uses a cache for pre-constructed values, thus reducing the number of created instances and improving garbage collection. Creating instances of boxed types through the constructor unnecessarily circumvents the cache and creates superfluous objects that are subject to garbage collection.

To always obtain an instance of a boxed primitive via `valueOf()`, even for values which are not cached by the JVM, keeps the programming model consistent and open for future JVM optimizations.

SonarQube rule
Illegal Instantiation

Don't construct wrappers for primitives from String arguments

Create instances of wrapper types using the `valueOf()` factory method with a primitive value passed into. Don't pass the String representation of the primitive.

Reasoning
Passing a constant String as argument for a Boxed primitive type constructor increases the computational overhead required to parse the String, while adding no value.

Don't copy primitive wrapper types

Don't copy wrapper objects for primitives by passing wrappers to a constructor or the `valueOf()` method of the same primitive wrapper type. Use the wrapper object directly instead of copying it.

Reasoning
Boxed types are immutable and can therefore be used directly.

Type conversions

Prefer primitive types to boxed primitives

Use primitives in preference to boxed primitives whenever you have the choice.

Reasoning

Primitive types are simpler and faster. If you must use boxed primitives, be careful! Autoboxing reduces the verbosity, but not the danger, of using boxed primitives. When your program compares two boxed primitives with the == operator, it does an identity comparison, which is almost certainly not what you want. When your program does mixed-type computations involving boxed and unboxed primitives, it does unboxing, and when your program does unboxing, it can throw a NullPointerException. Finally, when your program boxes primitive values, it can result in costly and unnecessary object creations.

SonarQube rules

Boxed value is unboxed and then immediately reboxed

Performance - Primitive value is boxed and then immediately unboxed

Performance - Primitive value is boxed then unboxed to perform primitive coercion

Efficient iteration and loops

This section discusses rules and best practices on how to implement efficient repetitive application code.

Iteration

Use entrySet iterator rather than keySet iterator

For iterations over the entries of a Map, use an iterator over the Map's `entrySet()`, rather than an iterator over the Map's `keySet()` in combination with a second call to retrieve the entry for the key.

Reasoning

Iterating over Maps using the `entrySet()` method makes the code more readable and increases the performance as repetitive lookups of the values are no longer necessary.

SonarQube rule

Performance - Inefficient use of keySet iterator instead of entrySet iterator

Loops

Reuse results returned by method calls

If the result of a certain method call is needed multiple times, just cache the result instead of repeating the same method call over and over again.

Reasoning

This is especially important in regard to loops.

Avoid SQL queries in loops

SQL queries should not be executed in loops. Rather execute a query ahead of loops and iterate over the result set in the loop.

Reasoning

Repeatedly executing SQL queries puts a high load on the application and on the database. Executing a single, more complex query which comprises all data required in the loop is much more efficient, even if the construction of the more complex query requires additional code.

Efficient string handling

This section discusses rules and best practices on how to process strings in an efficient way.

String allocation

Don't create unnecessary String objects

The `java.lang.String(String)` constructor wastes memory because the object so constructed will be functionally indistinguishable from the String passed as a parameter. Just use the argument String directly.

Reasoning

It is often appropriate to reuse a String object instead of creating a new object each time it is needed. Reuse can be both faster and more stylish.

Consider this statement:

```
String s = new String("JCS");   // DON'T DO THIS!
```

The statement creates a new String object each time it is executed, and none of those object creations is necessary. The argument to the String constructor ("JCS") is itself a String instance, functionally identical to all of the objects created by the constructor. If this usage occurs in a loop or in a frequently invoked method, millions of String instances can be created needlessly.

The improved version is simply the following:

```
String s = "JCS";
```

This version uses a single String instance, rather than creating a new one each time it is executed.

SonarQube rules

Performance - Method invokes inefficient new String(String) constructor

Performance - Method invokes inefficient new String() constructor

String comparisons

Use String.indexOf(char) to check for individual characters in Strings

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

Reasoning

The `indexOf(char)` method has less computational overhead.

SonarQube rule

Use Index Of Char

Use isEmpty() to check if String is empty

Use `isEmpty()` to check strings for emptiness, rather than using `compareTo()`, `equalsIgnoreCase()` or `compareToIgnoreCase()` comparisons.

Reasoning

The `isEmpty()` method is computationally less expensive and doesn't require the construction of an intermediate empty String used for the comparison.

Avoid unnecessary case changes

Use `String.equalsIgnoreCase(String)` rather than `String.toUpperCase/toLowerCase().equals()` for case-insensitive String comparisons.

Reasoning

`String.toUpperCase/toLowerCase()` create new String objects, whereas `String.equalsIgnoreCase(String)` does not.

SonarQube rule
Unnecessary Case Change

Avoid case-insensitive String comparison after enforcing case

Don't compare Strings via `String.compareToIgnoreCase()` or `String.equalsIgnoreCase()` which have been converted to lower or upper case.

Reasoning
A simple comparison using the `equals()` method has less computational overhead and leads to the same result if both the compared String and the comparison String are either lower or upper case.

String concatenation

Concatenate String in loops via StringBuilder

In Loops concatenate String objects using a java.lang.StringBuilder rather than the + operator.

Reasoning
In each iteration, the String is converted to a StringBuffer/StringBuilder, appended to, and converted back to a String. This can lead to a cost quadratic in the number of iterations, as the growing string is recopied in each iteration. Better performance can be obtained by using a StringBuffer (or StringBuilder in Java 1.5) explicitly.

StringBuilder vs. StringBuffer

StringBuilder is designed for use as a drop-in replacement for StringBuffer in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations.

SonarQube rule
Performance - Method concatenates strings using + in a loop

Construct StringBuffer/Builder with expected maximum length

Construct StringBuffers and StringBuilders with their expected maximum length. If the maximum length cannot be determined, estimate a length which covers estimated the maximum length for most invocations.

Reasoning
StringBuilder and also StringBuffer maintain an internal buffer holding their content – the character sequence.

If an `append()` or `insert()` operation exceeds the buffer's capacity, the buffer is replaced by another buffer with the size of the character sequence after the concatenation. The old buffer's content is copied into the newly allocated buffer and the filled up with the characters to be appended. The old buffer is released. This is time consuming, requires additional garbage collection and fragments storage unnecessarily.

Thus, the performance gain from StringBuilders and StringBuffers can only be fully gained if they are constructed with a capacity which reflects the final length after all concatenations.

The default constructor and the constructor from an empty String initialize the buffer size with 16 bytes. If constructed from a non-empty String, the buffer's capacity is the String's length.

SonarQube rules
[Insufficient String Buffer Declaration]

Don't use String concatenation in StringBuilder/Buffer arguments

StringBuffers/-Builders should be used for String concatenation. Strings should not be concatenated with the + operator in `append()` or `insert()` arguments.

Rather use the outer StringBuffer/Builder for all concatenations by invoking multiple appends or inserts, each with a single literal.

Reasoning
Concatenating String literals in StringBuffer/Builder constructors, `append()` or `.insert()` calls will create an intermediate StringBuffer to concatenate the argument Strings before they are passed to the invoked method.

Sample

StringBuilder usage

```java
import java.util.StringBuilder;
...
public static String foo(final String name) {
    // Allocate StringBuilder with expected maximum length:
    StringBuilder builder = new StringBuilder(40);

    // Replace concatenated argument ...
    builder.append("Name: " + name); // Unnecessary creates a temporary String object.

    // ... by less expensive:
    builder.append("Name: ")
           .append(name);

    return builder.toString();
}
```

An even briefer implementation is:

```java
import java.util.StringBuilder;
...
public static String foo(final String name) {
    return new StringBuilder(40).append("Name: ")
                                .append(name)
                                .toString();
}
```

SonarQube rule
[Inefficient String Buffering]

Retrieve StringBuffer/StringBuilder length directly

To retrieve the length of the String held by a StringBuilder/Buffer, call the `length()` method on the StringBuilder/Buffer, rather than converting the StringBuilder/Buffer to a String first an then calling `length()` on the String.

▾ Reasoning
Creating a String out of a StringBuffer/-Builder just to determine the length is wasteful as it unnecessarily creates an intermediate String object.

▾ SonarQube rule
Use String Buffer Length

Append character with append(char)

To append a single character to a StringBuffer/StringBuilder, use `append(char)`, and not the `append(String)` method.

▾ Reasoning
`StringBuilder/Buffer.append(String)` would have to create an intermediate String object, whereas `append(char)` doesn't.

▾ SonarQube rule
Append Character With Char

Append primitives directly to StringBuffer/StringBuilder

To append a primitive p to StringBuffer/StringBuilder, use `append(p)` rather than `append(String.valueOf(p))`.

▾ Reasoning
Prevents creation of a temporary intermediate String object and reduces garbage collection.

▾ SonarQube rule
Useless String Value Of

String conversions

Don't concatenate an empty string with a literal for conversion to String

Don't concatenate an empty String with a literal in order to convert the literal into a String. Rather use the `toString(literal)` method of the literal's wrapper type, or the `java.lang.String.valueOf()` factory methods.

▾ Reasoning
String concatenation unnecessarily creates one or more intermediate StringBuilder objects, depending on the length of the String, and calls methods on the StringBuilder to construct the concatenated String.

▾ SonarQube rule
Add Empty String

Don't use intermediate objects for conversions between primitives or wrappers and strings

Use static conversion methods of wrapper classes to create primitives, instead of creating intermediate wrapper instances.

For conversions from	to	use
`primitive`	`String`	`<WrapperType>.toString(primitive)`
`<WrapperType>`	`String`	`<WrapperType>.toString()`
`String`	`primitive`	`<WrapperType>.parse<Type>(String)`
`String`	`<WrapperType>`	`<WrapperType>.valueOf(String)`

where `<WrapperType>` is a placeholder for the primitive wrapper class, e.g. `java.lang.Integer`, and `<Type>` is a placeholder for the name of the primitive, e.g. `Int`.

Depending on the wrapper type, the wrapper class's static `parse<Type>()`, `toString()` and `valueOf()` methods provide additional arguments in order to support a variety of string representations of numeric values. If this is a requirement, use these methods rather than the simple ones.

Reasoning

The wrapper classes for primitives provide conversion methods to primitives and factory methods to obtain wrapper class instances from Strings, which eliminate the cost of creating unnecessary temporary wrapper instances to be garbage collected later.

Samples

String conversions

```java
String  myString = "123";
int     myInt;
Integer myInteger;

// (1) String to int conversion

// (1.1) Bad - waste of an object:
myInt = Integer.valueOf(myString).intValue();

// (1.2) Good - wrapper class conversion method:
myInt = Integer.parseInt(myString);


// (2) String to Integer conversion

// (2.1) Bad - waste of an object:
myInteger = Integer.valueOf(Integer.parseInt(myInt));

// (2.2) Good - wrapper class factory method:
myInteger = Integer.valueOf(myString);


// (3) int to String conversion

// (3.1) Bad - waste of an object:
myString = Integer.valueOf(myInt).toString();

// (3.2) Good - wrapper class conversion method:
myString = Integer.toString(myInt);
```

SonarQube rule

Performance - Method allocates a boxed primitive just to call toString

String retrieval

Use String.charAt() to retrieve a character from a String

Individual characters should be extracted from Strings using the `charAt()` method instead of first creating a `char` array from the String and then retrieving the character from the `char` array.

▾ Reasoning

Creating a `char` array to get specific characters from a String is cumbersome, as it the `char` array gets released shortly after creation. This produces avoidable computational efforts and memory consumption for an array which is then subject to garbage collection.

Don't call String.toString()

Don't call `toString()` on a String. Use the String object directly.

▾ Reasoning

`String.toString()` just does a "return this;".

▾ SonarQube rule

Performance - Method invokes toString() method on a String

String vs. char method arguments

Don't pass String argument when char argument is sufficient

If a class exposes similar methods, one with a `char` and one with a `String` argument, invoke the method with the `char` argument rather than passing a String of length 1 to the method with the `String` argument.

▾ Reasoning

Handling `char` values is more performant than using heavy weight `String` objects. Additionally, using characters instead of strings with length 1 increases the understandability of code as the length of 1 can be deduced from the method call.

Programming

This chapter discusses best practices and pitfalls which are related to programming applications written in Java.

Pitfalls with arrays and collections

Best practices and pitfalls related to programming with arrays and collections.

Pitfalls with comparisons

Best practices and pitfalls related to comparisons and the equals() and hashCode() methods.

Pitfalls with numbers, dates and times

Best practices and pitfalls related to bitwise operations and programming with numeric values, dates, calendars and times.

Pitfalls with resources

Best practices and pitfalls related to programming with resources like files, streams, databases, properties and resource bundles.

Pitfalls with strings

Best practices and pitfalls related to programming with Strings.

Pitfalls with arrays and collections

This section discusses best practices and pitfalls which are related to programming with arrays and collections.

Using arrays and collections

Keys in HashMap and elements in HashSet must have immutable hash codes

Only use keys in HashMap and elements in HashSet that have immutable hash codes.

Reasoning

HashMap and HashSet store the hash code in the collection at the time when the element is added. If the element's hash code changes as a result of a modification of the element, the element's hash code and the hash code stored in the collection become inconsistent, resulting in unexpected behavior of the collection when adding or retrieving elements.

Don't call hashCode() on arrays to consider the content

If you would like to consider the contents of an array, the hash code must be computed using `java.util.Arrays.hashCode(<array>)`.

Reasoning

Calling hashCode on an array returns the same value as `System.identityHashCode`, and ignores the contents and length of the array. If you need a `hashCode` that depends on the contents of an array a, use `java.util.Arrays.hashCode(a)`.

SonarQube rule

Correctness - Invocation of hashCode on an array

Don't call collections with unrelated type arguments

Don't call collection methods like `contains(Object)` or `remove(Object)` with arguments whose type is incompatible with the generic type that the collections has been defined for.

Reasoning

The collection will not contain an object of the argument type, which makes the method call superfluous. Most likely, this is an indication for the wrong value being passed to the method.

SonarQube rule

Correctness - No relationship between generic parameter and method argument

Pass array of same type as the collection member type to Collection.toArray()

The `java.util.Collection.toArray(MyObject[])` method may only be called with the correct array given as parameter.

Reasoning

A `java.lang.ClassCastException` will be thrown if the array passed to the `toArray(<T>[])` method doesn't have the same type as the collection's generic type or is a supertype of the generic type.

SonarQube rule

Class Cast Exception With To Array

Modifying collections

Don't modify collection directly while iterating

Don't call modifying operations on a collection while iterating over that collection. Use the iterator's `remove()` method to remove an element from the collection while iterating.

Reasoning

Modifying a collection directly while iterating invalidates the iterator and will cause a `ConcurrentModificationException`.

The iterator's `remove()` method removes the element from the collection and preserves the iterator's integrity.

Don't add collection to itself

Don't add a collection to itself, using the collection's `addAll()` method.

Reasoning

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

SonarQube rule

Correctness - A collection is added to itself

Collection must not contain itself

A collection must not contain itself as an element.

Reasoning

Collections that contain themselves will lead to several problems like entering an infinite loop when computing the element's hash code on an invocation of `Collection c.contains(c)`.

SonarQube rule
Correctness - Collections should not contain themselves

Don't retain Map.Entry objects

`Map.Entry` objects may not be retained for later usage.

Reasoning
`Map.Entry` objects are only guaranteed to contain valid data during iteration. If such objects are kept, e.g. in a separate collection, then their contents is not defined - they may be empty, or all of them may contain the values of the last entry objects, as they may be reused during iteration. Especially, the `Set` returned from calling `Map.entrySet()` may not be put into another collection, as the `Entry` objects are not guaranteed to be in a valid state.

SonarQube rule
Adding elements of an entry set may fail due to reuse of Entry objects

Pitfalls with comparisons

This section discusses best practices and pitfalls which are related to comparisons and the `equals()` and `hashCode()` methods.

General comparison rules

Avoid repeated comparisons

Don't repeat the same comparison in a boolean expression, such as `(x == 0) && (x == 0)`.

Reasoning

The repeated comparison has no effect. It is just confusing and indicates a copy/paste error. Perhaps something like `(x == 0) && (y == 0)` has been intended.

SonarQube rule

Correctness - Repeated conditional tests

Don't compare values with themselves

Don't compare fields or variables with themselves.

Reasoning

The comparison is useless, as it always evaluates to `true` for `equals()` and == comparisons and to false for `!=` comparisons. Such a comparison is always a clear indicator for a logic error or typo in the code.

SonarQube rules

Correctness - Self comparison of field with itself

Correctness - Self comparison of value with itself

Don't compare unrelated types

Don't compare types that don't have a common interface or base class other than Object – neither via `equals()`, nor via == or `!=` operators.

Reasoning

The comparison will always evaluate `false` at runtime, which is a clear indication for a logic error.

SonarQube rules

Correctness - Call to equals() comparing different interface types

Correctness - Call to equals() comparing different types

Correctness - Call to equals() comparing unrelated class and interface

Correctness - Using pointer equality to compare different types

Guard call to compareTo() and equals() with null check

Guard calls to `compareTo()` or `equals()` on variable operands with `null` checks.

Reasoning

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

SonarQube rule

Unused Null Check In Equals

equals() implementation

equals() must return false for null arguments

The `equals()` method must return `false` for `null` arguments.

Reasoning
This behavior is part of the `equals()` contract defined by `java.lang.Object`.

SonarQube rule
Bad practice - equals() method does not check for null argument

equals() must be symmetric

The `equals()` method must be symmetric: For two non `null` objects a and b of possibly different types, `a.equals(b)` must evaluate to the same result as `b.equals(a)`. Especially, `equals()` must not always return `true`.

Reasoning
Defined by the contract of the `Object.equals(Object)` method, and a prerequisite for many types like maps and sets to work as designed.

An `equals(Object)` implementation that always returns `true` is definitively not symmetric for arguments of a type that doesn't override `Object.equals(Object)`. It would be impossible to create useful maps or sets of objects of such a type as adding an element to the collection replaces a previously contained element that equals. So there could not be more than one element of the type in the collection.

SonarQube rules
Correctness - equals method always returns true

Correctness - equals method overrides equals in superclass and may not be symmetric

equals() must be reflexive

The `equals(Object)` method must return `true` when invoked on itself. Especially, `equals(Object)` must not always return `false`.

Reasoning
Defined by the contract of the `Object.equals(Object)` method, and a prerequisite for many types like maps and sets to work as designed. For objects of a type with an `equals(Object)` implementation that always returns `false` it would be impossible to create useful maps or sets of objects of such a type as, since the collection cannot detect elements that equal.

SonarQube rule
Correctness - equals method always returns false

equals() must be transitive for equality

The `equals(Object)` method must be transitive for equality: For three non `null` objects a, b and c of possibly different types, if `a.equals(b)` and `b.equals(c)` evaluate to `true`, then `a.equals(c)` must also evaluate to `true`.

Note that transitivity must not be given for objects that don't equal: if `a.equals(b)` and `b.equals(c)` evaluate to `false`, then `a.equals(c)` may still evaluate to `true`.

Reasoning

Defined by the contract of the `Object.equals(Object)` method.

Don't assume types in equals()

Implementations of the `equals()` method may not assume anything about their enclosing type or the type of the argument passed to the method. Instead, use dynamic type lookups rather than than hard coded type literals as reference for type comparisons. Return `false` if the enclosing type and the type of the argument don't match.

Reasoning

An equals method should simply return `false` if the type of the given argument is not equal to the enclosing type. Any further assumptions about the parameter type are not necessary and misleading.

An `equals()` method that uses a static type for comparisons will break when it is inherit by a subclass that doesn't override the method.

Sample

Broken type check in equals

```
public class Foo {
    ...
    public boolean equals(final Object obj) {
        if (obj == null) {
            return false;
        }

        // This type check will fail when inherited by subclasses ...
        if (Foo.class == obj.getClass()) {

        // ... whereas this will also work for subclasses:
        if (this.getClass() == obj.getClass()) {

            ...
        }
    }
}
```

SonarQube rules

[Bad practice - equals method fails for subtypes]()

[Bad practice - Equals method should not assume anything about the type of its argument]()

equals() must not only compare types

The evaluation result of the `equals(Object)` method must not only depend on the type of the argument. Even if the class has been declared as `final` and can only be instantiated once (a singleton), provide a safer implementation like

```
public boolean equals(final Object obj) {
    return (this == obj);
}
```

Note that this is the default implementation provided by `java.lang.Object`. Overriding the equals method with this implementation only makes sense if the class is a subclass of a class that overrides the `equals(Object)` method.

Reasoning

When invoked with an argument of the same type, returning `true` would make the type unusable for maps or sets as the collection cannot detect elements that equal, and returning `false` would violate *equals() must be reflexive*. Furthermore, such an evaluation of equality could easily fail for subclasses if the comparison is based on the class object or class name of the passed `obj` argument.

SonarQube rule

Correctness - equals method compares class names rather than class objects

Don't have equals() compare incompatible operands

An `equals()` method may only return `true` if the object type of the given parameter is the same as of the enclosing object.

Reasoning

Checking for other types than that of the enclosing object is considered bad practice, since it makes it very hard to implement an equals method that is symmetric and transitive. Without those properties, unexpected behavior is likely.

Sample

Incompatible type comparison in equals()

```java
public class Foo {
    private String name;
    ...

    // !!!Beware of such code!!!
    public boolean equals(final Object obj) {
        if (obj instanceof Foo) {
            return this.name.equals(((Foo) obj).name);
        } else if (obj instanceof String) {
            // The illegal comparison:
            return this.name.equals(obj); // Compares String with Foo!
        }
        return false;
    }
}


// In code that invokes such an equals() method, rather compare
// the String with Foo's name property than with the Foo instance:
String name = "Some Name";
Foo    foo  = new Foo();
...
// Replace ...
if (foo.equals(name)) {
    ...
}

// ... by:
if (name.equals(foo.getName()) {
    ...
}
```

SonarQube rule

Bad practice - Equals checks for noncompatible operand

Using equals()

Use equals() to compare objects

Compare objects using the `equals()` method rather than the `==` or `!=` operators. Primitive wrappers and Strings must be compared using `equals()`.

Reasoning

The == and `!=` operators check for identity of object references. Since logical equal objects don't need to have identical references, only the `equals()` method reliably verifies equality of two objects.

Equal instances of Boolean only have the same identity of created through `Boolean.valueOf()`, which returns `Boolean.TRUE` or `Boolean.FALSE`, whereas the Boolean constructor creates equal instances with a different identity.

Equal Strings are only identical if interned.

Also see subsequent rules for type specific comparison rules.

There are a few exceptions for that rule, e.g. Enums.

SonarQube rules

Bad practice - Suspicious reference comparison

Correctness - Suspicious reference comparison of Boolean values

Correctness - Suspicious reference comparison to constant

String Literal Equality

Don't compare arrays using equals()

Rather than using the `equals()` method, use `java.util.Arrays.equals(Object[],Object[])` to compare the contents of two arrays, and the == and `!=` operators to compare arrays for identity.

Reasoning

As arrays are Objects, the compiler allows invocations of `equals()` on an array. The method only checks for identity, which is better expressed by the == and `!=` operators.

SonarQube rule

Correctness - Invocation of equals() on an array, which is equivalent to ==

Don't compare arrays with instances of incompatible types

Don't compare arrays and non-arrays or arrays of incompatible types.

Reasoning

Such comparisons always yield `false`, which makes them superfluous or indicates a programming error.

SonarQube rules

Correctness - equals() used to compare array and nonarray

Correctness - equals(...) used to compare incompatible arrays

Don't call equals() with null argument

Don't pass a `null` value for the argument of the `equals()` method.

Reasoning

According to the interface contract, `equals(null)` always returns `false`. Thus, calling `equals()` with `null` is either superfluous or indicates a programming error.

SonarQube rule

Correctness - Call to equals() with null argument

Using hashCode()

Don't compute absolute value of hash codes

Don't compute the absolute value of the result returned by `hashCode()`.

Reasoning

The value returned by `Math.abs()` for a hash code might still be negative, as `Math.abs(Integer.MIN_VALUE)` is equal to `Integer.MIN_VALUE`. One out of 2^{32} objects have a hash code of `Integer.MIN_VALUE`, e.g. the strings `"polygenelubricants"`, `"GydZG_"` and `"DESIGNING WORKHOUSES"`.

Generally avoid arithmetics on `hashCode()` results.

SonarQube rule

Correctness - Bad attempt to compute absolute value of signed 32-bit hashcode

Pitfalls with numbers, dates and times

This section discusses best practices and pitfalls which are related to bitwise operations and programming with numeric values, dates, calendars and times.

- Numeric types
    - *Avoid float and double if exact answers are required*
    - *Don't use BigDecimal constructor with floating point arguments*
- Numeric calculations
    - *Don't execute modulo 1 operations on integers*
    - *Don't cast an integral value to double and then pass it to Math.ceil*
    - *Don't cast an int value to float and then pass it to Math.round*
    - *Ensure required value range and precision for intermediate results of numeric calculations*
    - *Use & operator to check if an integer is odd*
    - *Use >>> operator to compute average of two non-negative integers*
- Numeric comparisons
    - *Don't implement vacuous numerical comparisons*
    - *Don't compare value with constant of complementary sign*
    - *Don't compare integral types with constants of larger range*
    - *Compare floating points with difference of delta*
    - *Use isNaN() to check for Not A Number values*
- Bitwise operations
    - *Don't use binary shifts with vacuous or confusing shift amounts*
    - *Avoid unsigned right shift operations with casts to smaller integral types*
    - *Suppress sign extensions with byte packaging*
    - *Compare with zero to check if bits are set*
    - *Don't use vacuous bit masks in comparisons*

Numeric types

Avoid float and double if exact answers are required

`Prefer BigDecimal` rather than `float` or `double` when decimal precision is required.

Reasoning

Don’t use float or double for any calculations that require an exact answer. Use BigDecimal if you want the system to keep track of the decimal point and you don’t mind the inconvenience and cost of not using a primitive type. Using BigDecimal as the added advantage that it gives you full control over rounding, letting you select from eight rounding modes whenever an operation that entails rounding is performed. This comes in handy if you’re performing business calculations with legally mandated rounding behavior. If performance is of the essence, you don’t mind keeping track of the decimal point yourself, and the quantities aren’t too big, use int or long. If the quantities don’t exceed nine decimal digits, you can use int; if they don’t exceed eighteen digits, you can use long. If the quantities might exceed eighteen digits, you must use BigDecimal.

Don't use BigDecimal constructor with floating point arguments

Create `java.math.BigDecimal` object's via `BigDecimal.valueOf()` factory methods, rather than using a constructor.

Reasoning

Code that creates a BigDecimal from a double value doesn't translate well to a decimal number. For example, one might assume that writing new BigDecimal(0.1) in Java creates a BigDecimal which is exactly equal to 0.1 (an unscaled value of 1, with a scale of 1), but it is actually equal to 0.1000000000000000055511151231257827021181583404541015625. You probably want to use the BigDecimal.valueOf(double d) method, which uses the String representation of the double to create the BigDecimal (e.g., BigDecimal.valueOf(0.1) gives 0.1).

SonarQube rule

BigDecimal constructed from double that isn't represented precisely

Numeric calculations

Don't execute modulo 1 operations on integers

Don't execute modulo `1` (`% 1`) operations on integral primitive types.

Reasoning

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

SonarQube rule

Correctness - Integer remainder modulo 1

Don't cast an integral value to double and then pass it to Math.ceil

Don't cast integral values to double values and then invoke `ceil()`.

▾ Reasoning

Code that converts an integral value (e.g., int or long) to a double precision floating point number and then passing the result to the Math.ceil() function, which rounds a double to the next higher integer value, is often a no-op. Since converting an integer to a double should give a number with no fractional part. It is likely that the operation that generated the value to be passed to Math.ceil was intended to be performed using double precision floating point arithmetic.

▾ SonarQube rule

Correctness - int value cast to double and then passed to Math.ceil

Don't cast an int value to float and then pass it to Math.round

Don't cast integer-based values to floating-point values and then invoke `round()` on the floating point value.

▾ Reasoning

This operation should always be a no-op, since the converting an integer to a float should give a number with no fractional part. It is likely that the operation that generated the value to be passed to Math.round was intended to be performed using floating point arithmetic.

▾ SonarQube rule

Correctness - int value cast to float and then passed to Math.round

Ensure required value range and precision for intermediate results of numeric calculations

When performing arithmetic operations, assure that the type used for intermediate values doesn't overflow and preserves the required precision. Especially, don't use types for intermediate results with a lower precision or a lower value range than the result's type.

If the arithmetic operation contains divisions, consider using floating point arithmetic rather than integer arithmetic, even if the expected result is an integer.

▾ Reasoning

Intermediate calculation results can exceed the value range of the operands, thus unexpectedly changing sign and lead to wrong arithmetic results.

In case of divisions, integer calculations continuously ignore fractional parts and can lead to numeric overflows, whereas `double` values maintain a high precision and the correct sign throughout the arithmetic operations.

▾ Sample

Arithmetic type error

```java
// The following statement will return -1.616.567.296:
static final long MILLIS_IN_MAY = 31 * 24 * 60 * 60 * 1000;

// ... whereas this will correctly return 2.678.400.000:
static final long MILLIS_IN_MAY = 31L * 24 * 60 * 60 * 1000;


// The following statement will return 0.0 ...
double value = 1 / 4;

// ... whereas this will correctly return 0.25:
double value = 1 / 4.0d;
```

SonarQube rules

Dodgy - int division result cast to double or float

Dodgy - Result of integer multiplication cast to long

Use & operator to check if an integer is odd

Use `((x & 1) == 1)` rather than `(x %2 == 1)` to check if an integer-based value `x` is odd.

Reasoning

The `(x %2 == 1)` comparison won't work for negative numbers, since it returns `-1` for negative odd numbers. The `(x % 2 != 0)` comparison also works correctly to detect odd numbers, but comes with a higher computational overhead than `((x & 1) == 1)`.

SonarQube rule

Dodgy - Check for oddness that won't work for negative numbers

Use >>> operator to compute average of two non-negative integers

Use `(a + b) >>> 1` rather than `(a + b) / 2` to calculate the average of two non-negative integers a and b. Note that this rule does not apply to situations where a or b can have negative values.

Reasoning

The `(a + b) / 2` calculation can lead to an intermediate overflow when calculating `a + b`, which results in the computation of a negative average, although a and b are non-negative.

SonarQube rule

Dodgy - Computation of average could overflow

Numeric comparisons

Don't implement vacuous numerical comparisons

Don't compare numerical values in a way that always yield the same result, as in `if (x <= Integer.MAX_VALUE)` for an `int` value `x`.

Reasoning

Such a comparison is either superfluous or a result of a logic error or typo.

SonarQube rule

Dodgy - Vacuous comparison of integer value

Don't compare value with constant of complementary sign

Don't compare a value whose sign is known with a constant or value that has the complementary sign.

Reasoning

The comparison is vacuous, as it is guaranteed to fail. This indicates a logic error or typo.

SonarQube rule

Correctness - Bad comparison of nonnegative value with negative constant

Don't compare integral types with constants of larger range

Don't compare integer-based values (`byte`, `short`, `int`) with constants of a greater type that hold a value which is outside the range of the compared value.

Reasoning

Code that compares an int value with a long constant that is outside the range of values can be represented as an int value. This comparison is vacuous and possibily incorrect. Comparing a signed byte with a value outside that range is also vacuous and likely to be incorrect. To convert a signed byte b to an unsigned value in the range 0..255, use `0xff & b.`

SonarQube rules

Bad comparison of int value with long constant

Correctness - Bad comparison of signed byte

Compare floating points with difference of delta

Calculated float and double values may not be accurate

Reasoning

Because floating point calculations may involve rounding, calculated float and double values may not be accurate. For values that must be precise, such as monetary values, consider using a fixed-precision type such as BigDecimal. For values that need not be precise, consider comparing for equality within some range, for example: `if ( Math.abs(x - y) < .0000001 )`. See the Java Language Specification, section 4.2.4.

SonarQube rule

Dodgy - Test for floating point equality

Use isNaN() to check for Not A Number values

Because of the special semantics of `NaN,` no value is equal to Nan, including `NaN`.

Reasoning

Because of the special semantics of NaN, no value is equal to Nan, including NaN. Thus, `x == Double.NaN` always evaluates to false. To check to see if a value contained in `x` is the special Not A Number value, use `Double.isNaN(x)` (or `Float.isNaN(x)` if `x` is floating point precision).

SonarQube rule

Correctness - Doomed test for equality to NaN

Bitwise operations

Don't use binary shifts with vacuous or confusing shift amounts

Don't perform a shift of a 32 bit int by a constant amount outside the range -31..31.

Reasoning

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

SonarQube rule

Correctness - Integer shift by an amount not in the range 0..31

Avoid unsigned right shift operations with casts to smaller integral types

Avoid *unsigned* right shift operations (>>>) on integers and then cast the result to an integral type with a smaller number of bits.

Reasoning

Since the upper bits are discarded, there may be no difference between a signed and unsigned right shift (depending upon the size of the shift). There are only very rare situations where such a shift is useful. Typically, a *signed* shift operation (>>) is more adequate.

SonarQube rule

Dodgy - Unsigned right shift cast to short/byte

Suppress sign extensions with byte packaging

When packing a `byte` array representation of a `short`, `int` or `long` value into a value of its actual type, set all bits of the byte value to zero except its significant 8 trailing bits.

Reasoning

Values loaded from a byte array are sign extended to 32 bits before any any bitwise operations are performed on the value. Thus, if `b[0]` contains the value `0xff`, and `x` is initially 0, then the code `((x << 8) + b[0])` will sign extend `0xff` to get `0xffffffff`, and thus give the value `0xffffffff` as the result.

In particular, the following code for packing a byte array into an int is badly wrong:

```
int result = 0;
for(int i = 0; i < 4; i++)
  result = ((result << 8) + b[i]);
```

The following idiom will work instead:

```
int result = 0;
for(int i = 0; i < 4; i++)
  result = ((result << 8) + (b[i] & 0xff));
```

Byte values are sign extended to 32 bits before any bitwise operations are performed on the value. Thus, if `b[0]` contains the value `0xff`, and `x` is initially `0`, then the code `((x << 8) | b[0])` will sign extend `0xff` to get `0xffffffff`, and thus give the value `0xffffffff` as the result.

In particular, the following code for packing a byte array into an int is badly wrong:

```
int result = 0;
for(int i = 0; i < 4; i++)
  result = ((result << 8) | b[i]);
```

The following idiom will work instead:

```
int result = 0;
for(int i = 0; i < 4; i++)
  result = ((result << 8) | (b[i] & 0xff));
```

SonarQube rules

Correctness - Bitwise add of signed byte value

Correctness - Bitwise OR of signed byte value

Compare with zero to check if bits are set

When checking an integral type for bit status via bit masking ([code]variable & mask[/code]), do not use ordering relations (< and >). Use equality relations (== and !=) instead.

Reasoning

Negative values are represented in two's complement, the MSB being a 1. E.g. an AND operation with two negative arguments will have a negative result, while positive arguments will lead to a positive result. Thus a check with >, <, >= or <= 0 will be less robust than a check for == or != 0 regarding further development.

SonarQube rules

Bad practice - Check for sign of bitwise operation

Correctness - Check for sign of bitwise operation

Don't use vacuous bit masks in comparisons

Don't compare expressions of the form `(e & C1)` or `(e | C1)` to `C2` where the comparison result only depends on the constants `C1` and `C2`, but not on the expression `e`.

▾ Reasoning
The comparison is just superfluous and indicates a logic error or a typo.

▾ SonarQube rules
Correctness - Incompatible bit masks (BIT_AND)

Correctness - Incompatible bit masks (BIT_IOR)

Don't use vacuous bit masks for bitwise operations

Don't use vacuous bit masks like `0x00000000` or `0xffffffff` for bitwise operations (and, or, or exclusive or).

▾ Reasoning
The operation is useless, as its result is predefined.

▾ SonarQube rule
Dodgy - Vacuous bit mask operation on integer value

Arguments for bitwise floating point conversions must have correct number of bits

Don't pass a value with less than 32 bits to `Float.intBitsToFloat(int)` or a value with less than 64 bits to `Double.longBitsToDouble(long)`.

▾ Reasoning
The `Double.longBitsToDouble(long)` method expects the argument to be a representation of a floating-point value according to the 64-bit IEEE 754 floating-point "double format" bit layout. A value with less than 64 bits (a `byte`, `char`, `short` or `int`) will undergo a widening conversion into a long, but is very unlikely to be the correct bit representation of a `double`.

The same applies to `Float.intBitsToFloat(int)` if the passed value has less than 32 bits.

▾ SonarQube rule
Correctness - Double.longBitsToDouble invoked on an int

Dates and times

Use Calendar constants for month representations

When using int values to represent months, ensure that the values range from `0` (Jan) to `11` (Dec), rather than `1` ... `12`. Whenever possible, use the month constants defined in `java.util.Calendar` rather than `int` literals.

▾ Reasoning
Not very intuitive, Java month constants range from `0` (Jan) to `11` (Dec), which is a frequent cause for program errors.

▾ SonarQube rule
Correctness - Bad constant value for month

Use long values to represent absolute times

Only use 64-bit `long` representations for absolute times. Avoid any casts to 32-bit `int` representations.

Reasoning

An absolute time value is the number of milliseconds since the standard base time, known as "the epoch", namely January 1, 1970, 00:00:00 GMT. 32-bit `int` representations can only represent small sub-ranges of absolute times.

Sample

Absolute time calculation

```java
// Fails for dates before Dec 1969 and after Jan 1970:
Date getDate(final int secondsSinceNewYear1970) {
    return new Date(secondsSinceNewYear1970 * 1000);
}

// Fails for dates after 2037:
Date getDate(final int secondsSinceNewYear1970) {
    return new Date(secondsSinceNewYear1970 * 1000L);
}

// Works for all dates:
Date getDate(final long secondsSinceNewYear1970) {
    return new Date(secondsSinceNewYear1970 * 1000L);
}
```

SonarQube rule

int value converted to long and used as absolute time

Call Calendar.before() and Calendar.after() with Calendar argument

Ensure that the object passed to `java.util.Calendar.before(Object)` and `java.util.Calendar.after(Object)` is a `java.util.Calendar` instance.

Reasoning

The methods always return `false` for arguments other than `java.util.Calendar`, which makes the method call superfluous and indicates a programming error.

Pitfalls with resources

This section discusses best practices and pitfalls which are related to programming with resources like files, streams, databases, properties and resource bundles.

Database resources

Don't use vendor specific database APIs

Use standard JDBC APIs rather than vendor-specific methods and types for direct database access.

Reasoning
JDBC decouples the database API from vendor specific solutions.

Use 1..n as indices for parameters of prepared statements

When using prepared statements, never use a parameter index lower than `1`.

Reasoning
In contract to array element indices, parameter indices for prepared statements range from `1..n`.

SonarQube rule
Correctness - Method attempts to access a prepared statement parameter with index 0

Use 1..n as indices for fields of ResultSets

When accessing ResultSets, never use a field index lower than `1`.

Reasoning
In contract to array element indices, field indices for ResultSets range from `1..n`.

SonarQube rule
Correctness - Method attempts to access a result set field with index 0

Check return values when navigating through ResultSets

Always check the return values of navigation methods (`absolute(..)`, `first()`, `last()`, `next()`, `previous()`, `relative(..)`) of a ResultSet and handle situations properly where `false` is returned.

Reasoning
The navigation methods of a `java.sql.ResultSet` return `false` to indicate that the cursor is positioned before the first or after the last row. If this is the case, accessing a row or field of the ResultSet will fail and throw a `java.sql.SQLException`.

SonarQube rule

Check ResultSet

Ensure that database resources are released under all conditions

Ensure that all database resources such as database connections or row sets allocated by the application are properly released when they are no longer required.

In situations where database access is not managed by application code, but encapsulated by a persistence framework, ensure that the persistence framework has been configured properly to avoid unnecessary consumption of database resources.

Reasoning

Database resources are precious and limited. If they are not properly released, this may lead to poor application performance or completely disable the application to obtain database resources when these are required.

It is not acceptable to rely on the database to automatically release resources after a certain timeout, as this jeopardizes stability: Before the timeout has been reached, the application might request database resources more often than granted by the database. In this case, the database will deny to allocate new resources before blocked resources are released.

Furthermore, actively releasing database resources in the application makes the application more robust against changes in the database configuration for runtime parameters such as timeouts for automated release of locks and resources.

SonarQube rules

Bad practice - Method may fail to close database resource

Bad practice - Method may fail to close database resource on exception

File and stream resources

Check return code when reading from streams

When reading input from a stream, check the return code before processing the read content.

Reasoning

All read methods will return special values (like -1 or null) when the end of the stream is reached.

SonarQube rules

Bad practice - Method ignores results of InputStream.read()

Dodgy - Dereference of the result of readLine() without nullcheck

Dodgy - Immediate dereference of the result of readLine()

Don't ignore content read from stream

When reading input from a stream, don't ignore the read content.

▾ Reasoning

Reading from a stream and discarding the result may indicate a logic error. Note that re-reading the same content by invoking the read operation again at a later point of time is no option, as the read content will not be the same as the content read from the previous invocation.

▾ SonarQube rule

Dodgy - Method discards result of readLine after checking if it is nonnull

Check return code of InputStream.skip()

When invoking the `skip()` method on a `java.io.InputStream`, check the return code before further processing the stream.

▾ Reasoning

The return code of the `skip()` method returns the number of skipped bytes. The number might be smaller than requested when the end of the stream has been reached. When the result is ignored, the program might sporadically run into logical errors which can be hard to debug.

▾ SonarQube rule

Bad practice - Method ignores results of InputStream.skip()

Ensure that stream resources are released under all conditions

Ensure that all stream resources are properly released when they are no longer required. Note that this also applies to exceptional conditions.

▾ Reasoning

Open streams consume JVM and operating system resources and may impose locks. If not released properly, deadlocks may arise.

▾ SonarQube rules

Bad practice - Method may fail to close stream

Bad practice - Method may fail to close stream on exception

Don't call File.deleteOnExit()

Avoid using `java.io.File.deleteOnExit()`. Rather delete the file explicitly before the application terminates and, to be safe in case of an abnormal JVM shutdown, remove remaining files from a previous application execution when the application is started.

▾ Reasoning

The `deleteOnExit()` method will only be executed when the application terminates normally, but not when it is killed or crashes. Furthermore, the memory associated to this file deletion handler is only released at the end of the process.

▾ SonarQube rule

Do not use File#deleteOnExit()

Ensure that binary content is written to archive

When adding content to a binary archive like a .jar or a .zip file, ensure that the `write(byte[], int, int)` method is invoked between `putNextEntry()` and `closeEntry()`.

Reasoning

The `putNextEntry()` methods of classes `java.util.JarOutputStream` and `java.util.ZipOutputStream` only write the metadata, but not its binary content. Before closing the entry, the `write(byte[], int. int)` method must be called to write the binary content to the file. Otherwise, only an empty .jar/.zip file entry would be written.

SonarQube rules

Bad practice - Creates an empty jar file entry

Bad practice - Creates an empty zip file entry

Open wrapped file output streams in random access mode when appending content to the file

Code that opens a file in append mode and then wraps the result in an object output stream won't allow you to append to an existing object output stream stored in a file.

Reasoning

If you want to be able to append to an object output stream, you need to keep the object output stream open. The only situation in which opening a file in append mode and the writing an object output stream could work is if on reading the file you plan to open it in random access mode and seek to the byte offset where the append started.

SonarQube rule

Correctness - Doomed attempt to append to an object output stream

Properties

Use setProperty() to store a String into a Properties object

To store a value into a `java.util.Properties` object, use the `setProperty()` method rather than `put()`.

Reasoning

Although `java.util.Properties` is derived from `java.util.Hashtable`, don't use the inherit methods to add elements. The class's `setProperty()` method has been designed for this purpose instead and offers better readability of the code.

Obtain system property via System.getProperty()

Use the `System.getProperty(<key>)` method if you intend to obtain a system property, rather than using `System.getProperties().getProperty(<key>)`.

Reasoning

Retrieving the system property directly is more efficient and circumvents generating a temporary Hashtable that needs to be garbage collected.

Don't use Class.getResource()

Don't retrieve resources via `getClass().getResource(String)` in a non-final class.

Reasoning

If the class gets subclassed in another package and the resource name is not an absolute path, the resource will be obtained from a different location. If there is no resource for the given name at that location, `null` will be returned, otherwise the returned `java.net.URL` will point to a resource with unexpected content. The same applies if a subclass uses a different class loader that implements another resolution of the resource name to a file path. For further information follow the link.

SonarQube rule

Bad practice - Usage of GetResource may be unsafe if class is extended

HTTP headers and cookies

Protect HTTP headers and cookies from untrusted input

User supplied input must be sanitized if it is to be embedded into HTTP headers or HTTP cookies.

Reasoning

When a user is able to embed arbitrary data into an HTTP header, cross-site scripting attacks, cross-user defacement, web cache poisoning, and similar exploits can be performed. The same risk arises when a user is able to embed arbitrary data into an HTTP cookie when the cookie is added to the HTTP header. Make sure that the data is at least sanitized, or better don't let user generated input at all get into HTTP headers and cookies.

SonarQube rules

Security - HTTP cookie formed from untrusted input

Security - HTTP Response splitting vulnerability

Servlets and Java Server Pages

Don't write untrusted content into webpages

User supplied content may not be directly written into webpages.

Reasoning

Writing the contents of an HTTP parameter to the JSP output, to the servlet output or to the server error page enables cross site scripting attacks, as a malicious user may embed malicious JavaScript Code into the page by forging a special URL, and distributing the URL to unsuspecting victims.

SonarQube rules

Security - JSP reflected cross site scripting vulnerability

Security - Servlet reflected cross site scripting vulnerability (send error)

Security - Servlet reflected cross site scripting vulnerability (servlet writer)

Sanitize pathnames constructed from HTTP request parameters

Ensure that pathnames constructed from HTTP request parameters point to a restricted directory. Especially, sanitize such pathnames from absolute paths or pathnames containing "`../`" pointing to some other location outside the restricted directory.

▾ Reasoning

Pathnames used by servlets should be within a restricted directory. If such pathnames are constructed from HTTP request parameters and the code does not check if the constructed pathname is within the restricted directory, attackers may traverse the file system to access files or directories that are outside of the restricted directory. Attacker may exploit this vulnerability to create or overwrite critical files that are used to execute code, such as programs or libraries.

See Absolute Path Traversal and Relative Path Traversal for details.

▾ SonarQube rules

Absolute path traversal in servlet

Relative path traversal in servlet

Pitfalls with strings

This section discusses best practices and pitfalls which are related to programming with Strings.

Regular expressions

Double-check regular expressions before coding

Intensively verify regular expressions before they are incorporated into program code. For instance, use an interpreter for regular expressions and test that the expression is syntactically correct and reliably filters as expected.

▾ Reasoning

The power of regular expressions comes with an inherent complexity. Even expressions that appear to be simple and clear can easily show an unexpected behavior.

▾ SonarQube rule
Correctness - Invalid syntax for regular expression

Be careful when using "." in regular expressions

In regular expressions, always use the dot (`"."`) sign with a preceding "\" unless it is intended to be a wild-card for an arbitrary character.

▾ Reasoning
In regular expressions, the `"."` sign is interpreted as a wild-card: The result of `new String("com.bmw.it4it").replaceAll(".", "/")` is `"////////////"`, not `"com/bmw/it4it"` as one might expect.

▾ SonarQube rule
Correctness - "." used for regular expression

Don't construct regular expression using File.separator

Don't use `java.io.File.separator` for the construction of regular expressions.

▾ Reasoning
The `File.separator` character varies between operating systems: It is "/" for Unix-based platforms like Linux or OS X, but "\" for Windows. Since the "\" character is interpreted as an escape character in regular expressions, a regular expression which is constructed from `File.separator` would have a different syntax and interpretation, depending on the operating system of the runtime environment. Amoung other options, you can just use `File.separatorChar=='\\' ? "\\\\" : File.separator` instead of `File.separator`

▾ SonarQube rule
Correctness - File.separator used for regular expression

String comparisons

Check if result of String.indexOf() is negative

To find out if a substring is contained in a String object, check if the index returned from `String.indexOf(...)` is negative or non-negative, rather than checking if it is positive.

▾ Reasoning
If the String object begins with the substring checked for, the index returned from the `indexOf(...)` methods is `0`. It is only positive if the substring occurs at some other place than the beginning of the String.

▾ SonarQube rule
Dodgy - Method checks to see if result of String.indexOf is positive

String formatting

Use argument indices in format strings

Use argument indices like `1$` for parameter arguments in format strings. For instance, use format strings like `"There are %1$,d days left until %2$tb %2$te, %2$tY."`[1] rather than `"There are %,d days left until %tb %<te, %<tY."`[2].

[1]The String constructed from the above format string together with an Integer value that represents 1,234 and a Date that represents the 4th of March 2013 (in this order) is `"There are 1,234 days left until Mar 4, 2013."` if formatted with an English locale.

[2]If used without argument indices, the same argument can only be used multiple times if used in a row and with a relative index `"<"`: The Date argument is used three times (`"%tb", "%<te"` and `"%<tY"`) to display different elements of the same Date value.

Reasoning

Without argument indices, the first argument is interpreted as `1$`, the second as `2$`, and so forth. At runtime, the arguments are replaced by values that are passed as an array of Objects to the Formatter, with `1$` being replaced by the first Object, `2$` by the second Object, etc. The String can only be constructed correctly from the format string and the array of values if both maintain the same order, but will fail if the values come in a different order than the argument order.

Furthermore, in many situations the same value needs to be expressed by multiple arguments in the format string: A Date value is typically expressed by multiple format string arguments in order to state the Date's components like day of week, day, month, year and time elements. Using an argument index is safer than referring to the previous argument using the `"<"` operator.

Finally, argument indices provide a better flexibility with respect to locale specific argument order in format strings or in situations where format strings are replaced at a later point of time by a more meaningful format string that requires a different argument order.

Argument index vs. value index

The format string argument indices used by `java.util.Formatter` and the `String.format(...)` methods range from `1..n`, whereas the indices in the Object array that holds the arguments' values as well as argument indices used by `java.text.MessageFormat` range from `0..n-1`.

Ensure valid format string syntax

Ensure that the syntax of the format string adheres to the Formatter syntax and that argument syntax corresponds to the argument types and their positions.

Reasoning

Erroneous format strings are a painful cause for errors which remain undetected by the compiler, but cause a RuntimeException to be thrown when the format string is processed by the Formatter.

Typical errors are:

- type mismatches like using `"%b"` for a non-Boolean argument;
- using higher argument indices than values are passed to the formatter;
- the first argument index is a relative index `"<"`;
- using `printf()` formats like `"\n"` rather than `"%n"` for line breaks.

▾ SonarQube rules

Correctness - Format string placeholder incompatible with passed argument

Correctness - Format string references missing argument

Correctness - Illegal format string

Correctness - MessageFormat supplied where printf style format expected

Correctness - More arguments are passed that are actually used in the format string

Correctness - No previous argument for format string

Correctness - Number of format-string arguments does not correspond to number of placeholders

Correctness - The type of a supplied argument doesn't match format specifier

Dodgy - Non-Boolean argument formatted using %b format specifier

Format string should use %n rather than \n

Don't pass arrays as arguments of format strings

Don't pass arrays as arguments of format strings to a Formatter. Consider wrapping the array using `java.util.Arrays.asList(..)`.

▾ Reasoning

Arrays cannot be formatted to anything useful. The output will be something like `"[C@16f0472"`, which doesn't represent the actual content of the array.

▾ SonarQube rule

Correctness - Array formatted in useless way using format string

Usage of toString() method

toString() must not return null

The `toString()` method method must not return `null`. Rather return an empty string.

▾ Reasoning

A `toString()` method invocation that returns `null` on an existing object could cause other code to break.

▾ SonarQube rule

Bad practice - toString method may return null

Don't invoke toString() on an array

Don't invoke `toString()` on an array.

▾ Reasoning

As arrays are objects, `toString()` can be invoked on them. However, the call produces nothing but unusable output such as "[C@16f0472", which doesn't represent the actual content of the array.

SonarQube rules

Correctness - Invocation of toString on an anonymous array

Correctness - Invocation of toString on an array

Construction of StringBuffer and StringBuilder

Don't construct StringBuffer/Builder from char

Don't create a StringBuffer or StringBuilder with a `char` argument passed to the constructor. Rather pass an `int` argument if you intend to provide an initial capacity, as recommended by *Construct StringBuffer/Builder with expected maximum length*.

Reasoning

Since StringBuffer and StringBuilder have no constructor from a `char` parameter, the `char` argument will be converted to an `int` and interpreted as the initial buffer size by the constructor from an `int` argument. The character passed to the constructor gets lost.

SonarQube rules

String Buffer Instantiation With Char

Security

This chapter discusses security-related rules for applications written in Java.

Conformity with BMW security standards

Strict adherence of Java code to the subsequent security-related programming rules only assures that basic conditions for secure applications are met. Further security rules as defined by BMW security standards must be adhered to.

The rules listed here do not substitute intensive security testing. Rather consider to conduct a special static code analysis and dynamic penetration testing for security-related issues. Furthermore, adherence to the rules listed in this section does not give any positive evidence that an application is secure or meets fundamental requirements like data protection.

Details on required application security at BMW can be obtained from http://it.muc/rc/IT_Intranet/en/010_govern/020_Compliance/Tabcontent/IT_security_rules2656365/index.htm.

General security rules

Precautions in Java code aiming to prevent unintended or unauthorized modifications of sensitive data.

Exposure of internal state

Precautions in Java code aiming to protect Java objects against unintended external manipulations.

Random numbers

Best practices how to securely create randomized numerical values.

General security rules

This section discusses precautions in Java code which aim to prevent unintended or unauthorized modifications of sensitive data.

Credentials

Don't hard-code any credentials

Source code may not contain sensitive information like database user ids and passwords.

▾ Reasoning

Hard-coded credentials most likely raise a serious security issue. Even if the credentials do not give access to production data, this unveils a false understanding of security since user ids and passwords can easily be extracted from the compiled application and are directly visible to anyone who has access to the source code or to the compiled byte code.

▾ SonarQube rule

Security - Hardcoded constant database password

Don't use empty passwords

Access to sensitive data must be restricted using a password, unless being used for access to local in-memory data storage, e.g. in-memory databases, which cannot be accessed from outside the application.

▾ Reasoning

Access to sensitive data without password protected access are widely open to misuse, e.g. to access or manipulate data without authorization. This may be legitimate when embedded data stores which are not externally exposed are used for caching, but should be reviewed.

▾ SonarQube rule

Security - Empty database password

Query statement

Use PreparedStatements over plain Statements

Use `java.sql.PreparedStatement` rather than `java.sql.Statement` when directly programming against the database.

Reasoning

In contrast to plain SQL statements, prepared statements are database query statements which can be pre-compiled by the database prior to execution. They only allow to insert certain parameter values dynamically, but reliably prevent the insertion of SQL code. Thus, prepared statements are an effective means to protect applications against SQL injection attacks, unless being dynamically constructed from Java code using SQL strings which have been externally manipulated.

Create PreparedStatement from constant string

Instances of `java.sql.PreparedStatement` should only be created from constant strings.

Reasoning

SQL prepared statements that are dynamically constructed enable SQL injection attacks if they are constructed from content which can be externally manipulated. The only uncritical dynamic aspect of using prepared statements is setting the prepared statement's parameter values by using the `setParameter(...)` methods.

There are some cases where there is no alternative to dynamically building SQL strings, e.g. when creating IN clauses with varying value sets. In these cases, build the SQL query without the actual data, and use the question mark placeholder instead. Inject the actual data using the `setParameter(...)` method afterwards.

SonarQube rule

Security - A prepared statement is generated from a nonconstant String

Execute Statement on constant string

Instances of `java.sql.Statement` must not be executed using non constant data.

Reasoning

Creating SQL statements from dynamically created strings bears the risk that these strings contain unintended, malicious SQL statements, e.g. when embedding external input into the SQL statement. This poses a major security risk, as this pattern can be misused for SQL injection attacks.

If the statement is to be dynamically composed, consider using a `java.sql.PreparedStatement` dynamicall y constructed from constant substrings instead. This makes the query robust against unintended manipulations.

SonarQube rule

Security - Nonconstant string passed to execute method on an SQL statement

Execution of sensitive code

Execute sensitive code inside a doPrivileged block

Code which requires a security permission check may only be invoked inside a `doPrivileged` block.

Reasoning

If the code is invoked by code with lower permissions, the `doPrivileged` block assures that permissions a checked prior to executing the code. A complete treatment of that topic is beyond the scope of the JCS, but the interested reader is directed to the official document.

SonarQube rule

Bad practice - Method invoked that should be only be invoked inside a doPrivileged block

Creating class loaders

Create ClassLoaders carefully

It can be very tricky to understand and implement Java class loaders, hence it should be done very judiciously.

Reasoning

Java class loaders can be broadly classified into the following categories (bootstrap, extensions and system class loader). All class loaders depend on each other and use three principles (delegation, visibility and uniqueness). To understand all implications for an own classloader are complicated and should be avoided. Therefore you should only create your own class loader in exceptional situations.

One of the reasons to write an own class loader is to control the JVM's class loading behavior. Let us say that we are running an application and we are making use of a class called Student. Assuming that the Student class is updated with a better version on the fly, i.e. when the application is running, and we need to make a call to the updated class. If you are wondering that the bootstrap class loader that has loaded the application will do this for you, then you are wrong. Java's class loading behavior is such that, once it has loaded the classes, it will not reload the new class. How to overcome this issue has been the question on every developers mind.

The answer is simple. Write your own class loader and then use your class loader to load the classes. When a class has been modified on the run time, then you need to create a new instance of your class loader to load the class. Remember, that once you have created a new instance of your class loader, then you should make sure that you no longer make reference to the old class loader, because when two instances of the same object is loaded by different class loaders then they are treated as incompatible types.

Create ClassLoaders inside a doPrivileged block

Class loaders may only be created inside a `doPrivileged` block.

Reasoning

Creating a class loader requires security permissions to be granted if a security manage is installed. The `doPrivileged` block assures that these permissions are checked.

SonarQube rule

Bad practice - Classloaders should only be created inside doPrivileged block

Exposure of internal state

This section discusses precautions in Java code which aim to protect Java objects against unintended external manipulations.

Security impact of exposing internal state

Unintended manipulations by unnecessary exposure of internal state can raise security issues. For instance, the manipulated data could be directly or indirectly used to access sensitive data, for instance via a call to a database or a web service, or the data could be written to a webpage.

Direct field access

Hide mutable fields

Mutable fields should not be externally accessible and must not be defined in interfaces.

Reasoning

Mutable fields like arrays or collections can be directly manipulated from external code if they are accessible from outside their surrounding type, thus changing the object's state in an uncontrolled way. For instance, elements of arrays could be replaced by other objects or `null` values, collections can be modified in an arbitrary way.

Besides the fact that defining constants in interfaces is a bad coding practice, never use interfaces to define constants with mutable objects, as the accessibility of each constant in an interface is `public` by default. Instead, constants should be defined in classes and made `private` for mutable objects. Use `static` getters to make the constants accessible from outside the class where required.

SonarQube rules

Malicious code vulnerability - Field should be moved out of an interface and made package protected

Malicious code vulnerability - Field should be package protected

Strictly control access to fields

Only fields that are declared as `static final` (constants) may be `public`. All other field may only be accessed via getter and setter methods. The only exemption from this rule are fields of `private` non-static inner classes and anonymous inner classes; for such classes, fields may be declared as package private (without access modifier), but never `public` or `protected`.

Reasoning

Prevents modifications of fields beyond control of their owning class.

Fields of `private` non-static inner classes and anonymous inner classes are only accessible from their surrounding class. Thus, modifications of such fields can only me made from code within the same class file. Declaring such fields as `public` or `protected` is useless and just confusing due to missing accessibility from outside the surrounding class.

SonarQube rule

Visibility Modifier

Access mutable fields via secure getters

Read access to the content of mutable fields may only be granted by getters which return a copy of the field, but not the reference to the field itself.

Reasoning

Returning a reference to a mutable object value stored in one of the object's fields or in a static field exposes the internal representation of the object or class. Thus, untrusted external code can access and manipulate the mutable field in an arbitrary way.

SonarQube rules

Malicious code vulnerability - May expose internal representation by returning reference to mutable object

Malicious code vulnerability - Public static method may expose internal representation by returning array

Manipulate mutable fields via secure setters

Write access to the content of mutable fields may only be granted by setters which internally store the provided state into a copy of the object given to the setter. The reference to the mutable object given as a setter argument must not be stored in the mutable field.

Reasoning

If the provided reference to a mutable object is stored in a mutable field, external code can access and manipulate the mutable field in an arbitrary way.

SonarQube rules

Malicious code vulnerability - May expose internal representation by incorporating reference to mutable object

Malicious code vulnerability - May expose internal static state by storing a mutable object into a static field

Random numbers

This section discusses best practices how to securely create randomized numerical values. As such values are frequently used to secure applications at runtime, security issues may arise if random values are not created a way which ensures a reasonably fair numerical distribution of the generated values.

Obtaining randomizer securely

Use SecureRandom instead of Random for cryptographical needs

Use `java.security.SecureRandom` over `java.util.Random` for cryptographical needs.

Reasoning

`SecureRandom` is a subclass of `Random`. It provides improvements for generating good cryptographical random numbers without breaking the `Random` API. The `instance(...)` methods added by `SecureRandom` enable creating random objects only once across your application, which significantly simplifies generating random objects that produce cryptographical random numbers at a high quality. Of course, you could always use `SecureRandom`, but `SecureRandom` is often slower than pure `Random`. And there are cases where you are only interested in good statistical properties and excellent performance, but you don't really care about security: Monte-Carlo simulations are a good example.

Reuse random object

Reuse random object, instead of creating random objects multiple times.

Reasoning

If objects of `java.util.Random` are created multiple times, the generated random numbers are easily guessable, resulting in mediocre quality random numbers and being inefficient. Instead, save the created random object and each time a new random number is required invoke a method on the existing random object to obtain it.

SonarQube rule

Bad practice - Random object created and used only once

Creating evenly distributed random numbers

Correctly generating non-negative random int

Don't use `Math.abs()` or modulo operations to generate a non-negative `int` from a random `int`. Use `Random.nextInt(int)` or `SecureRandom.nextInt(int)` instead.

Reasoning

`Math.abs(Integer.MIN_VALUE)` is again `Integer.MIN_VALUE`. Thus, if the random `int` is `Integer.MIN_VALUE`, you still end up with a negative `int`. Such errors are hard to detect as they occur once every 2^{32} times by average.

Modulo operations return negative values if applied on negative values.

SonarQube rules

Correctness - Bad attempt to compute absolute value of signed 32-bit random integer

Dodgy - Remainder of 32-bit signed random integer

Don't coerce floating point random numbers to int

Use `Random.nextInt(int)` or `SecureRandom.nextInt(int)` to coerce a random number from `0.0` to `1.0` to an `int`.

Reasoning

Obtaining an evenly distributed `int` from a random number ranging from `0.0` to `1.0` by multiplication is a frequent cause for errors due to the characteristics of integer arithmetic.

For instance, multiplying a random number from `0.0` to `1.0` by `4.0` (a `float` or a `double` value) and assigning the computational result to an `int` results in almost 25% of all values being `0` , `1`, `2` and `3`, but almost no value being `4` by average, as `4` requires the random value to be precisely `1.0`. `Random.nextInt(5)` produces roughly equally distributed values between `0` and `4`.

`Random.nextDouble()` and `Random.nextFloat()` create random numbers from `0.0` to `1.0`, resulting in the same uneven distribution as documented in the rule above when assigned to an int via `(int)(r.nextDouble() * n)`, where `r` is an instance of `java.util.Random`.

SonarQube rules

Correctness - Random value from 0 to 1 is coerced to the integer 0

Performance - Use the nextInt method of Random rather than nextDouble to generate a random integer

Testing

This chapter discusses rules for automated testing of applications written in Java. It is limited to tests implemented in Java and does not cover aspects like user acceptance tests, load tests or penetration tests.

Test metrics

Metrics on regression testing of Java code.

Best practices for testing

General best practices for regression testing.

JUnit test design

Unit test frameworks and guidelines on the granularity of unit test artifacts.

JUnit test implementation

Conventions and best practices on how to implement unit tests.

JUnit assertions

Rules and hints on the usage of JUnit assertions in regression tests.

JUnit naming conventions

Conventions on how to name JUnit test packages, classes and methods.

JUnit test suites

Rules on how to use test suites for regression testing.

Test metrics

This section discusses metrics on regression testing of Java code.

- Test coverage
    - *Line coverage*

Test coverage

Line coverage

At least 60 percent of manually written Java source code file must be covered by unit tests.

▾ Reasoning
This minimum line coverage gives a reasonable confidence in the functional correctness of the manually written Java code.

▾ SonarQube rule
Insufficient line coverage by unit tests

Branch coverage

At least 60 percent of all branches of manually written Java source code file must be covered by unit tests.

▾ Reasoning
This minimum branch coverage gives a reasonable confidence in the functional correctness and stability of the manually written Java code.

▾ SonarQube rule
Insufficient branch coverage by unit tests

Test success

Test failure

All regression tests must complete successfully.

▾ Reasoning
A failed regression test indicates a malfunction of the tested code that needs to be fixed. If the reason for the test failure is a defect in the test itself, fix the test. If the test is no longer relevant, remove the test, since over time it will soon be unclear if the failed test is still relevant and needs to be fixed / maintained or not.

Skipped tests

No regression tests may be skipped.

Reasoning

A skipped regression test indicates a malfunction of the test of of the code that is being tested. If the reason for skipping the test is a defect in the test or a test suite itself, fix the test or test suite. If the test is no longer relevant, remove the test, since over time it will soon be unclear if the failed test is still relevant and needs to be fixed / maintained or not.

Best practices for testing

This section discusses general best practices for regression testing.

Test strategy

Test systematically

Organize your tests hierarchically with the test scope being the main denominator for the test levels:

- *Unit tests* are the low-level building blocks. They verify the functional correctness of a single Java artifact, which can be a single method, a class or an enumeration.
- *Unit test suites*, consisting of a collection of unit tests, verify the functional correctness of a group of classes like all classes of a Java package or an entire software module, or they can be used to group all unit tests for a specific aspect like smoke tests at unit test level.
- *Integration tests* verify the collaboration of software components which typically run in different containers (e.g. EJB container and Web container) or even on different machines (e.g. tests verifying the collaboration between your application and an external service or another application).
- *GUI tests* verify the functional correctness of human interfaces; they rely on the correct behavior of the underlying layers, tested by unit and integration tests and test suites.

Reasoning

A well defined test structure facilitates testing in many ways:

- It provides a clear structure with respect to the test purpose (the type of test) and the test target (the artifacts which are tested).
- If source code is changed, affected test artifacts can easily be identified.
- Dependencies between test code can be minimized, as each test artifact has a well-defined, minimal scope.
- The separation of unit tests and integration tests enables developers to execute unit tests locally, as they don't require other systems to be running and available to conduct the tests.

Implement regression tests

For each manually written class or enumeration of productive code, implement regression tests which verify the functional correctness of the code.

▾ Reasoning

Regression tests a proven and very effective way to detect flaws in the code. Typically, such flaws are hard to detect by other means, and they come with significantly increased efforts if they not being detected early.

Once written, regression tests can be executed automatically to verify if the productive code works as desired. Effects of code modifications during enhancements or maintenance of the software can easily be detected and localized, especially if such modifications have been done on other code than the code subject to the test, like code which is used by the tested code or code which itself uses the tested code.

Another important benefit of regression testing is the analysis of issues introduced by the underlying middleware or used software libraries: Such changes can affect the application's behavior, e.g. if room parameters change, libraries or middleware get updated by newer versions (e.g. JDK, OS or application server upgrades), or if libraries or middleware are replaced (e.g. replacing WebLogic by Glassfish). In such cases, regression tests can be used for so called smoke testing to roughly verify if and how the application is affected.

This rule does not apply to generated code, as the correctness of code generators is not subject to regression testing.

Test early

Implement regression tests in parallel to writing the productive code.

▾ Reasoning

If the structure and functionality of the code is planned thoroughly, regression tests can be written prior to implementing the productive code which is subject to the tests, as regression tests check the expected behavior of the productive code. This should be defined and known prior to its implementation.

Test boundary conditions and violations

Implement tests which verify the correct behavior of the tested code under the expected runtime conditions ("positive tests"), as well as tests which test the code's behavior under exceptional conditions which violate these conditions ("exceptional tests").

Ensure that boundary conditions are well tested, e.g.

- If a method accepts a collection argument, test the method's behavior on a `null` argument value and on an empty collection.
- Methods which do calculations should be called with `0` or `0.0f`/`0.0d` arguments to find out if they run into errors by using the passed argument values for divisions.
- Arguments which are defined for a certain value range should be tested with values which equal the outer

limits of the value range, as well as the values immediately outside the defined value range.
- Test numerical integer arguments against zero, minimum and maximum values: To test a method which accepts an argument of type `int`, invoke the method with values `Integer.MIN_VALUE`, `-1`, `0`, `1` and `Integer.MAX_VALUE`.
  Apply the same to arguments of types `short` and `long`, and to their counterparts `Integer`, `Short` and `Long`, using the types' Java constants for minimum and maximum values.
- Test `String` arguments with `null`, empty strings, blanks and non-ASCII or special characters like German umlauts (`'ä'`, `'ö'`, `'ü'`, `'Ä'`, `'Ö'`, `'Ü'`), the euro sign (`'€'`) or other Unicode characters.

Reasoning
Positive tests verify that the code works fine under intended or "normal" conditions. Such tests are suitable for smoke testing.

Exceptional tests intentionally violate the interface contract of the tested code. They focus on the stability of the tested code by provoking errors and checking if these are detected and taken care of in a controlled way, e.g. by passing invalid (e.g. `null`) argument values to a tested method where a valid (e.g. non-null) value is expected.

Boundaries of values of value ranges are a main cause for errors and often handled wrong in the code. By testing boundary values, you can be very sure that the method will also work correctly on values between or beyond these boundaries.

The euro sign (€) is a good test candidate for string processing, as it is a non-ASCII character and UTF-8 encodes it as a three-byte character, which may cause errors if single- or double-byte length is expected by the tested code or an underlying component like a software library or a database.

Test internationalization

Code which is subject to internationalization must be tested against multiple locales, especially with respect to string processing and numerical, date and currency formats.

Reasoning
Internationalization is a permanent cause for errors, as it must handle a large variety of formats and characters. If not tested well, code tested for just one country is very likely not to run correctly for other countries.

Test documentation

Document implemented test scenarios

Use the Javadoc header of test methods to document the configurations used to test your code, and use inline comments (`//` comments) in the test code which refer to the test configurations documented in the test method's Javadoc.

Reasoning
Gives a quick overview of the tested scenarios and facilitates locating the test code for each scenario.

Bad practices

Don't use main() method for testing

Productive code may not contain static `main(final String args[])` methods which serve for testing the correctness of the enclosing class's implementation. As a general rule, such code must be moved to a unit test class in a test package.

In case of classes which can be directly called from the command line, place the following comment immediately above the signature of the `main` method:
`// Suppress UncommentedMain check: <reason>.`, where *`<reason>`* is placeholder for a brief explanation why the class requires the `main` method.

Reasoning
Clear separation of productive and test code that keeps the APIs of production code clean, as they are not populated with test functionality.

> **Suppress UncommentedMain check comment**
>
> The *`// Suppress UncommentedMain check ...`* comment can be used to disable the *Uncommented Main* rule check, thus allowing implementations of `main` in productive code to fulfill the class's responsibilities. In this case, the code's purpose must not be testing the correctness of the enclosing class's implementation.

SonarQube rule
Uncommented Main

JUnit test design

This section discusses the usage of unit test frameworks and gives guidelines on the granularity of unit test artifacts.

- Unit test frameworks
    - *Use JUnit*
    - *Use DBUnit*
- Unit test granularity
    - *Unit test class per tested type*
    - *TestCase must contain tests*
- Reuse of test parametrizations
    - *Use constants and fixtures*

Unit test frameworks

Use JUnit

Use JUnit to implement unit tests for your code. Use JUnit 4.x over earlier versions unless technical restrictions require an earlier version.

Reasoning
JUnit is very powerful and flexible, yet easy to use. Extensions like DBUnit enhance JUnit 3.x for special test purposes. Code coverage tools like Cobertura or Emma/EclEmma/Jacoco as well as the Maven Java build tool and the BMW Continuous Integration solution well support JUnit. Furthermore, JUnit is widely spread and most Java developers are familiar with this unit test framework

In contrast to earlier versions, JUnit 4.x uses descriptive means implemented by annotations instead of inheritance to mark the elements of unit tests like test setup and tear-down methods, test methods or test suites.

Use DBUnit

Use DBUnit for relational database related tests that are attached via JDBC, e.g. for tests which need to setup test data in a relational database prior to running the tests, or validate data in a relational database to determine if the tested code behaves in the expected way.

Reasoning

DBUnit offers simple to use means to maintain, populate and compare data sets, which would otherwise require a lot of test code if implemented as plain JUnit tests.

Unit test granularity

Unit test class per tested type

Try to implemented all tests for a tested type – a class or enumeration of the production code – within the same JUnit test class.

Reasoning

Grouping test code by the tested item and using clearly defined unit test class names facilitates identifying the right unit tests for regression testing or for modifications of test code if the tested item gets modified.

TestCase must contain tests

For JUnit versions prior to 4.0, each class derived from `junit.framework.TestCase` must contain at least one test.

Reasoning

Empty test case classes are useless, as they only suggest the presence of tests without actually implementing one.

SonarQube rule

Correctness - TestCase has no tests

Reuse of test parametrizations

Use constants and fixtures

Use constants and fixtures for test data which can be used by multiple unit tests of the same test class. Constants are best to hold test data which can be shared by multiple instances of the test class. Member variables, are suitable for data which can be shared by multiple unit tests, but not by multiple instances of the unit test class.

Reasoning

Unit tests typically require the construction of test objects prior to invoking the tested code, e.g. in order to create argument values with the right state for the intended tests. Instead of creating such objects for each test, try to create them just once when the test class is loaded or when the instance of the test class is created. Thus, test preparation code can be eliminated from the test cases and is only written once.

For JUnit 4.x, initialize constants in a `public static` method without arguments which is annotated `@BeforeClass`; use a `static public` method without arguments which is annotated `@AfterClass` to release the test data. To initialize fixtures, use `public` instance methods without arguments, annotated `@Before` for the setup and `@After` for the cleanup method.

For JUnit 3.x, methods `protected void setup()` and `protected void teardown()` handle initialization and cleanup of fixtures.

JUnit test implementation

This section discusses conventions and best practices on how to implement unit tests.

System output

Don't produce system output

Don't write test output to `System.out` or `System.err`. Instead, use message arguments in JUnit `assertXxx(...)` methods to indicate failures.

Reasoning

JUnit uses messages for test output. Any output sent to `System.out` and `System.err` makes the output from build jobs difficult to read. So keep it free from output produced by the test code and use meaningful message arguments in JUnit assertions instead.

Exceptions

Test if expected exceptions are thrown

When implementing exceptional test, don't just check that some exception is thrown, but check that the thrown exception is of the correct type as defined in the API of the tested code.

Reasoning

A methods only fulfills its interface contract if it throws exactly the defined exceptions for the exceptional conditions defined by its API. Thus, checking for just some exception does not verify if the tested method works as designed.

Samples

JUnit 3 exception check

```java
// Tests Order.add(Option, int):
public void testAddForNullArgument() {
    final Order order = new Order();
    try {
        order.add(null, 2); // IllegalArgumentException expected.
        Assert.fail("No IllegalArgumentException thrown for null argument."); //$NON-NLS-1$
     } catch (final Exception e) {
        if (! (e instanceof IllegalArgumentException)) {
            Assert.fail(e.getClass().getName()
                        + " instead of IllegalArgumentException thrown for null argument."); //$NON-NLS-1$
        }
    }
}
```

JUnit 4 exception check

```java
// Tests Order.add(Option option, int amount) for an invalid option (null) and a valid positive amount argument:
@SuppressWarnings("static-method")
@Test(expected = IllegalArgumentException.class)
public void testAddforNullArgument() {
    final Order order = new Order();
    order.add(null, 2); // IllegalArgumentException expected.
}
```

Note that the JUnit 3 programming pattern is also suitable for JUnit 4: The JUnit 4 sample only allows a single invocation of the tested method and requires dedicated test method implementations for each combination of method arguments that are expected to throw an exception, whereas the JUnit programming pattern allows to test multiple method argument combinations in the same test method.

Avoid catching Throwable or Error

Avoid catching `Throwable` or `Error` in unit tests.

Reasoning

Catching `junit.framework.AssertionFailedError` without re-throwing it intercepts JUnit's test failure indication.

SonarQube rule

Illegal Catch

Unit test set-up and tear-down

setUp() must call super.setUp()

For JUnit versions prior to 4.0, the first statement of the `protected void setUp()` method must call `super.setUp()`.

Reasoning

Ensures that the fixture setup done by the the JUnit framework is not intercepted.

SonarQube rule

Correctness - TestCase defines setUp that doesn&apos;t call super.setUp()

tearDown() must call super.tearDown()

For JUnit versions prior to 4.0, the last statement of the `protected void tearDown()` method must call `super.tearDown()`.

Reasoning

Ensures that the fixture cleanup done by the the JUnit framework is not intercepted.

SonarQube rule

Correctness - TestCase defines tearDown that doesn&apos;t call super.tearDown()

JUnit assertions

This chapter discusses how JUnit assertions are to be used in regression tests.

Assertion usage

JUnit test must include assertion

JUnit tests must include at least one assertion.

Reasoning

Using assertions over plain Java boolean expressions clearly separates the logic to implement the control flow of the test from the logic to verify the correctness of tested code. This makes the tests more robust, and using assert with messages provide the developer a clearer idea of what the test does.

SonarQube rule

JUnit tests should include an assert

Don't place assertion into run() method

For JUnit versions prior to 4.0, don't place any assertions in `public TestResult run()` and `public void run(TestResult)`.

Reasoning

Failed JUnit assertions in `run(...)` methods just result in exceptions being thrown. Thus, if this exception occurs in a thread other than the thread that invokes the test method, the exception will terminate the thread instead of indicating a test failure.

SonarQube rule

Correctness - JUnit assertion in run method will not be noticed by JUnit

Assertion must include message

Assertions in JUnit tests must include a message which briefly describes the error detected if the assertion fails.

Reasoning

All JUnit assertion methods are provided in two variants: one with a leading String argument for the error message, and one without the message argument. Only use variants which include the message argument in order to facilitate the interpretation why the test failed.

SonarQube rule

JUnit assertions should include a message

Use most specific assertion

JUnit tests should use the most specific assertions: `assertNull` is to be used for `null` checks, `assertEquals` for equality checks and `assertSame` for identity checks.

Reasoning

Assertions should be done with the most specific method, instead of using `assertTrue(...)` for arbitrary checks, as the specific methods are easier to read.

SonarQube rules

Use assertEquals instead of assertTrue

Use assertNull instead of assertTrue

Use assertSame instead of assertTrue

Use tolerance to compare floats and doubles

Floating point values must not be directly compared to each other, but use a small delta as tolerated difference. Thus, always use the JUnit comparison methods with the delta argument for comparing two floating point values, but not the deprecated methods without the delta argument.

The delta must be chosen with the nature of the numbers in mind. Use the biggest number that won't make a difference.

Delta computation for floating point comparisons

As a general hint, `Math.ulp(x) * 10.0` for x being the expected value is a reasonable value for the delta by which an actual value can differ from an expected value without being considered as different.

Reasoning

Doing math with floating point numbers has the fundamental problem that the precision is not absolute. When comparing to different floating point values, the loss of precision has to be considered and it has to be decided, in which case two numbers can be seen as equivalent.

Boolean assertions

Use assertTrue() and assertFalse() for boolean expression

Use `assertTrue(...)` and `assertFalse(...)` to evaluate boolean expressions, rather than `assertEquals()`.

Reasoning
Assertions should be done with the most specific method. For arbitrary checks, `assertTrue("My error message", booleanExpression)` is more specific and easier to read than `assertEquals("My error message", Boolean.TRUE, booleanExpression)`.

Simplify boolean assertions

Avoid negation in `assertTrue` or `assertFalse` tests. For example, rephrase `assertTrue("My error message", ! booleanExpression)` as `assertFalse("My error message", booleanExpression)`.

Reasoning
The evaluations in JUnit tests should be as simple as possible.

SonarQube rule
Simplify boolean assertion

Remove unnecessary boolean assertions

Boolean assertions (`assertTrue(...)` and `assertFalse(...)`) should not be called with boolean constants.

Reasoning
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("My error message", false)` or similar) or simply removing unnecessary statements like `assertTrue(true)` and `assertFalse(false)`. If you just want a test to halt after finding an error, use the `fail("My error message")` method, providing an indication message of why it did.

SonarQube rule
Unnecessary boolean assertion

JUnit naming conventions

This section discusses conventions on how to name JUnit test packages, classes and methods.

Test packages

Test package name

Test code packages like packages for unit tests should either be the same as the packages of the classes they are testing (starting with `com.bmw.`) or begin with `test.bmw.`.

▾ Reasoning

For test packages, using `test.bmw.` over `com.bmw.` allows test code to access the tested code as a true black box, as the test classes do not reside in the same packages as the tested code. Elements of the tested code with `package` accessibility cannot be accessed from the test code. This facilitates testing if the access modifiers in the tested code have been set correctly.

Another positive side effect of using different package names for productive and for test code is the ability to separate package Javadoc for productive packages from Javadoc for test packages. If both productive and test code reside in the same package, there may only be one file `package-info.java`, which would then contain the package's Javadoc both for the productive and for the test code, although the test code and its documentation is not intended to be accessibility from outside the tested code.

▾ SonarQube rule

Package name

Test classes

Test suite name

Construct the name of a test suite class from the name of the tested type or aspect and a "`TestSuite`" suffix; e.g. the unit test class to all enumeration types of a package could be named `EnumerationTestSuite`.

If unit tests require other classes which are only intended for test purposes, ensure that the names of such classes don't end with "`TestSuite`".

▾ Reasoning

A meaningful class name together with exclusively using the "`TestSuite`" suffix for JUnit test suite implementations make the class's purpose obvious and facilitate to locate the right place for maintenance measures.

Test class name

Construct the name of a class which contains one or more unit tests from the name of the tested type and a "`Test`" suffix; e.g. the unit test class to test the functional correctness of a class named `LogoCreator` would be `LogoCreatorTest`.

If unit tests require other classes which are only intended for test purposes, ensure that the names of such classes don't end with "`Test`".

▾ Reasoning

A meaningful class name for classes which implement unit tests makes the class's purpose obvious and facilitates to locate the right place for maintenance measures. By exclusively using the "`Test`" suffix for such classes, test code is well separated from test suites and other supporting code.

Test methods

Spelling of JUnit 3.x (de-)initializer methods

For JUnit versions prior to 4.0, methods `protected void setUp()` and `protected void tearDown()` must be spelled correctly and annotated with `@Override`.

Reasoning
Ensures that the methods are called correctly by the JUnit framework. If misspelled, e.g. `setup()` or `teardown()`, methods won't get called. The annotation causes a compiler warning in case of a misspelled method name.

SonarQube rule
JUnit spelling

JUnit test suites

This section discusses rules on how to use test suites for regression testing.

- Test suite design
    - *Group test cases into suites*
- Test suite implementation
    - *Define test suite methods correctly*

Test suite design

Group test cases into suites

Use JUnit test suites to group test cases, e.g. to group regression tests, smoke tests or tests which require a common test setup.

Reasoning
Grouping of tests facilitates testing of dedicated aspects, e.g. if "positive" tests (tests of the application's intended behavior) are grouped into a test suite for smoke tests, by invoking the test suite one can quickly get a good indication if the code is affected by a middleware change, without having to execute all regression tests.

Test suites can also be used to improve the runtime of unit tests: Since test suites come with their own set-up and tear-down methods, it can be more efficient to use a single test suite initialization/de-initialization rather than individual implementations in every test class.

The test setup implemented in the test suite's setup method is no longer required to be implemented in the test cases called by the test suite. Thus, the setup code is only executed once when the test suite is started, but not every time then a new test case is invoked. The same applies to the cleanup code which is typically implemented in `tearDown()` methods.

Test suite implementation

Define test suite methods correctly

For JUnit versions prior to 4.0, the `suite()` method in a JUnit test needs to be declared as `public static junit.framework.Test suite()`.

▾ Reasoning

If this rule is violated, the `suite()` method won't get called.

▾ SonarQube rules

Correctness - TestCase declares a bad suite method

Correctness - TestCase implements a non-static suite method

Samples

This chapter contains code samples which serve to illustrate coding issues addressed by the rules of the Java Coding Standard.

Code design samples

The samples used to illustrate the rules in the Code design chapter.

- Anonymous inner class initialization sample — Illustrates how to emulate a constructor for an anonymous inner class.
- Delegating constructor sample — Illustrates how to avoid redundant initialization code in constructors.
- Non-static initializer sample — Illustrates the deficiencies of non-static initializers vs. constructors.
- Overload wrapper with primitive sample — Illustrates overloading with wrappers and primitive.

Coding style samples

The samples used to illustrate the rules in the Coding style chapter.

- Apache Commons static logger sample — Illustrates how to define a logger for the Apache Commons logging framework.
- Apache Log4J static logger sample — Illustrates how to define a logger for the Apache Log4J framework.
- Boolean assignment sample — Illustrates how to efficiently assign boolean values.
- Broken null check samples — Illustrates errors caused by broken null checks.
- Class initialization sample — Illustrates how to ensure initialization of Class objects.
- Confusing multiple unary operators sample — Samples for confusing usage of multiple unary operators in the same statement.
- Elimination of non short-circuit boolean operator sample — Illustrates how to eliminate non short-circuit boolean operators.
- Ignored return value error sample — Illustrates errors caused by ignoring the return value of methods of immutable objects.
- Inner assignment sample — Sample for a valid inner assignment.
- Inner boolean assignment error sample — Illustrates errors caused by boolean assignments in conditions.
- Intended fall-through sample — Illustrates how to deal with intended fall-throughs in switch statements.
- Iteration over collection via loop sample — Illustrates how to efficiently iterate over a collection.
- Lost logger configuration sample — Illustrates a wrong initialization of the Java logger which will work for the Oracle JDK, but fail for OpenJDK.
- Misplaced null check sample — Illustrates errors caused by misplaced null checks.
- Non short-circuit boolean operator failure sample — Illustrates errors caused by non short-circuit boolean operators.
- Outer loop index modification in inner loop sample — Illustrates errors caused by loop index modifications inside the loop's body.
- Return null for primitive error sample — Illustrates errors caused by returning null for a primitive wrapper object.
- Robust logger configuration sample — Illustrates how to implement a logger configuration that runs correctly on both the Oracle JDK and the OpenJDK.
- Self-assignment error sample — Illustrates errors caused by unary operators in self assignments.

- Simplified boolean return sample — Illustrates how to express boolean returns in a brief an clear way.
- Ternary operator sample — Illustrates how express assignments which are guarded by a boolean condition in a short and easy to understand way.
- Unintentional fall-through sample — Illustrates an unexpected behavior in case statements due to a missing break or other exit statement from the case block.

Concurrency samples

The samples used to illustrate the rules in the Concurrency chapter.

- Double-checked locking sample — Illustrates errors caused by double checked locking.
- Lock usage without timeout sample — Illustrates how to obtain a lock without a timeout.
- Lock usage with timeout sample — Illustrates how to obtain a lock with a timeout.

Documentation samples

The samples used to illustrate the rules in the Documentation chapter.

- Javadoc block comment format sample — Javadoc block comment format for Java source code files.
- Method documentation sample — Illustrates how to document constructors and methods in Javadoc.
- Package documentation sample (package-info.java) — Illustrates how to document Java packages in Javadoc.
- Type documentation sample — Illustrates how to document class, interface, enum and annotation types in Javadoc.
- Variable documentation sample — Illustrates how to document static and member fields in Javadoc.

Formatting samples

The samples used to illustrate the rules in the Formatting chapter.

- Class declaration sample — Illustrates how to format the declaration section of a class.
- Confusing preceding whitespace sample — Illustrates confusing whitespace characters before semicolons or certain unary operators.
- Confusing trailing whitespace sample — Illustrates confusing whitespace characters after certain unary operators.
- C-style array definition sample — A C-style array definition, which is not to be used.
- Generics format sample — Illustrates how to format generics.
- Interface declaration sample — Illustrates how to format the declaration section of an interface.
- Java-style array definition sample — A Java-style array definition, which is expected to be used.
- Left curly braces sample — Illustrates where to set left curly braces.
- Method parameter padding sample — Illustrates the padding of method identifiers and method parameter lists.
- Right curly braces sample — Illustrates where to set right curly braces.
- Upper L sample — Illustrates how to enhance code readability for literals of type long.

Performance samples

The samples used to illustrate the rules in the Performance chapter.

- StringBuilder usage sample — Illustrates how to efficiently use the java.util.StringBuilder.
- String conversions sample — Illustrates how to efficiently convert between Strings, primitives and primitive wrapper classes.

Programming samples

The samples used to illustrate the rules in the Programming chapter.

- Absolute time calculation sample — Illustrates how to avoid date and time overflows caused by using types with

too small value ranges for time representations.
- Append to output stream sample — Illustrates how to avoid errors caused by also opening both a file output stream and a wrapping stream in append mode.
- Arithmetic type error samples — These samples illustrate how to avoid errors caused by intermediate value types which overflow or don't preserve the required precision.
- Bit check sample — Illustrates how to avoid logic errors caused by checking if a bit is set via greater than or smaller than relations rather than via a comparison with zero.
- Broken type check in equals sample — Illustrates how to avoiding errors caused by assuming anything about the equals() method's enclosing type or the type of the argument passed to the method.
- Byte array packaging sample — Illustrates how to avoid arithmetic errors caused by not setting all bits of the byte value to zero except its significant 8 trailing bits when packing a byte array representation of a short, int or long value into a value of its actual type.
- Erroneous array typecast sample — Illustrates errors caused by casting arrays to a super type of their element type.
- Hash code arithmetics sample — Illustrates errors caused by computing the reminder of a hash code modulo another value.
- Incompatible type comparison in equals() sample — Illustrates how to avoid errors caused by equals() methods which return true if the object type of the given parameter is the same as of the enclosing object.

Testing samples

The samples used to illustrate the rules in the Testing chapter.

- JUnit 3 exception check sample — Illustrates how to check if an expected exception is thrown for JUnit 3.
- JUnit 4 exception check sample — Illustrates how to check if an expected exception is thrown for JUnit 4.

Code design samples

This section contains the samples used to illustrate the rules in the Code design chapter.

- Anonymous inner class initialization sample — Illustrates how to emulate a constructor for an anonymous inner class.
- Delegating constructor sample — Illustrates how to avoid redundant initialization code in constructors.
- Non-static initializer sample — Illustrates the deficiencies of non-static initializers vs. constructors.
- Overload wrapper with primitive sample — Illustrates overloading with wrappers and primitive.

Anonymous inner class initialization sample

This sample illustrates the *Use init() method to initialize anonymous inner class* rule on how to emulate a constructor for an anonymous inner class.

Anonymous inner class initialization

```java
import java.util.Collection;
import java.util.Iterator;

public class MyCollectionType<E>
implements   Collection<E> {

    ...

    // Returns an instance of an anonymous inner class:
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private int index;

            // The emulated constructor:
            Iterator<E> init(final int index) {
                this.index = index;
                return this; // An emulated constructor must always return this.
            }

            public E next() {
                ...
            }
            ...
        }.init(0); // The invocation of the emulated constructor.
    }
}

```

Delegating constructor sample

This sample illustrates the *Use delegating constructors* rule on how to avoid redundant initialization code in constructors. See *Non-static initializer sample* for a far less elegant implementation of the same type, using a non-static initializer for object initialization.

Delegating constructor

```java
1   public class Iso3166Alpha2CountryCode {
2
3       // The initial version number.
4       public static final int INITIAL_VERSION = 1;
5
6       // Objects of this class shall be immutable. Thus, all fields are declared as final.
7       private final String alpha2Code;  // Mandatory, consisting of 2 upper case letters.
8       private final int    codeVersion;
9
10      // 2-argument constructor which initializes all member fields with parameter values.
11      public Iso3166Alpha2CountryCode(final String alpha2Code,
12                                      final int    codeVersion) {
13          super();
14          if (alpha2Code == null) {
15              throw new IllegalArgumentException("Missing ISO 3166-1 alpha-2 country code."); //$NON-NLS-1$
16          }
17          if ((alpha2Code.length() != 2) || (! alpha2Code.equals(alpha2Code.toUpperCase()))) {
18              throw new IllegalArgumentException("Invalid ISO 3166-1 alpha-2 country code."); //$NON-NLS-1$
19          }
20          if (codeVersion < INITIAL_VERSION) {
21              throw new IllegalArgumentException("Invalid code version."); //$NON-NLS-1$
22          }
23          this.alpha2Code  = alpha2Code;
24          this.codeVersion = codeVersion;
25      }
26
27      // Constructor from a parameter list that doesn't contain values for all member fields.
28      public Iso3166Alpha2CountryCode(final String alpha2Code) {
29          // Instead of repeating initialization code from the constructor above as in ...
30          super();
31          if(alpha2Code == null) {
32              throw new IllegalArgumentException("Missing ISO 3166-1 alpha-2 country code."); //$NON-NLS-1$
33          }
34          if ((alpha2Code.length() != 2) || (! alpha2Code.equals(alpha2Code.toUpperCase()))) {
35              throw new IllegalArgumentException("Invalid ISO 3166-1 alpha-2 country code."); //$NON-NLS-1$
36          }
37          this.alpha2Code  = alpha2Code;
38          this.codeVersion = INITIAL_VERSION;
39
40          // ... delegate initialization to the other constructor and replace lines 30-38 by just:
41          this(alpha2Code, INITIAL_VERSION); // The delegating call of the other constructor.
42      }
43
44      // Only getters, no setters, since this is an immutable class.
45      public final String getAlpha2Code() {
46          return this.alpha2Code;
47      }
48
49      public final int getCodeVersion() {
50          return this.codeVersion;
51      }
52  }
53
```

Non-static initializer sample

This sample illustrates the *Don't use non-static initializers* rule on the drawback of implementing a non-static initializer. See *Delegating constructor sample* on how to better implement object initialization and avoid a non-static initializer.

Non-static initializer

```java
public class Iso3166Alpha2CountryCode {

    // The initial version number.
    public static final int INITIAL_VERSION = 1;

    // Objects of this class shall be immutable. Thus, all fields should be declared as final.
    private final String mAlpha2Code;  // Mandatory, consisting of 2 upper case letters.
    private int codeVersion; // Cannot be final, since it is assigned a value twice.

    // The non-static initializer, setting the default value for codeVersion.
    {
        this.codeVersion = INITIAL_VERSION; // First assignment to codeVersion.
    }

    // 2-argument constructor which initializes all member fields with parameter values.
    public Iso3166Alpha2CountryCode(final String alpha2Code,
                                    final int    codeVersion) {
        this(alpha2Code);
        if (codeVersion < INITIAL_VERSION) {
            throw new IllegalArgumentException("Invalid code version."); //$NON-NLS-1$
        }
        this.codeVersion = codeVersion; // Second assignment to mCodeVersion.
    }

    // Constructor from a parameter list that doesn't contain values for all member fields.
    // codeVersion is initialized by the non-static initializer.
    public Iso3166Alpha2CountryCode(final String alpha2Code) {
        if (alpha2Code == null) {
            throw new IllegalArgumentException("Missing alpha-2 ISO country code."); //$NON-NLS-1$
        }
        if ((alpha2Code.length() != 2) || (! alpha2Code.equals(alpha2Code.toUpperCase()))) {
            throw new IllegalArgumentException("Invalid alpha-2 ISO country code."); //$NON-NLS-1$
        }
        this.alpha2Code = alpha2Code;
    }

    // Only getters, no setters, since this is an immutable class.
    public final String getAlpha2Code() {
        return this.alpha2Code;
    }

    public final int getCodeVersion() {
        return this.codeVersion;
    }
}

```

Overload wrapper with primitive sample

This sample illustrates the overload wrapper parameters with primitives of a higher value range problem.

Overload wrapper with primitive

```java
public class Autoboxing {

    public static void a(Character x) {
        System.out.println("Character " + x);
    }

    public static void a(int x) {
        System.out.println("int " + x);
    }

    public static void main(String[] args) {
        a('z'); // Output: int 122
    }
}
```

Coding style samples

This section contains the samples used to illustrate the rules in the Coding style chapter.

- Apache Commons static logger sample — Illustrates how to define a logger for the Apache Commons logging framework.
- Apache Log4J static logger sample — Illustrates how to define a logger for the Apache Log4J framework.

- Boolean assignment sample — Illustrates how to efficiently assign boolean values.
- Broken null check samples — Illustrates errors caused by broken null checks.
- Class initialization sample — Illustrates how to ensure initialization of Class objects.
- Confusing multiple unary operators sample — Samples for confusing usage of multiple unary operators in the same statement.
- Elimination of non short-circuit boolean operator sample — Illustrates how to eliminate non short-circuit boolean operators.
- Ignored return value error sample — Illustrates errors caused by ignoring the return value of methods of immutable objects.
- Inner assignment sample — Sample for a valid inner assignment.
- Inner boolean assignment error sample — Illustrates errors caused by boolean assignments in conditions.
- Intended fall-through sample — Illustrates how to deal with intended fall-throughs in switch statements.
- Iteration over collection via loop sample — Illustrates how to efficiently iterate over a collection.
- Lost logger configuration sample — Illustrates a wrong initialization of the Java logger which will work for the Oracle JDK, but fail for OpenJDK.
- Misplaced null check sample — Illustrates errors caused by misplaced null checks.
- Non short-circuit boolean operator failure sample — Illustrates errors caused by non short-circuit boolean operators.
- Outer loop index modification in inner loop sample — Illustrates errors caused by loop index modifications inside the loop's body.
- Return null for primitive error sample — Illustrates errors caused by returning null for a primitive wrapper object.
- Robust logger configuration sample — Illustrates how to implement a logger configuration that runs correctly on both the Oracle JDK and the OpenJDK.
- Self-assignment error sample — Illustrates errors caused by unary operators in self assignments.
- Simplified boolean return sample — Illustrates how to express boolean returns in a brief an clear way.
- Ternary operator sample — Illustrates how express assignments which are guarded by a boolean condition in a short and easy to understand way.
- Unintentional fall-through sample — Illustrates an unexpected behavior in case statements due to a missing break or other exit statement from the case block.

Apache Commons static logger sample

This sample illustrates the *Define logger properly* rule on how to define a logger for the Apache Commons logging framework. See Apache Log4J static logger sample on how to define a logger for the Apache Log4J framework.

Apache Commons static logger

```java
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class MyClass {
    ...
    private static final Log LOG = LogFactory.getLog(MyClass.class);
    ...
    public void doSomething(final Object obj) {
        LOG.trace("Entering doSomething(Object)");
        ...
    }
}
```

Apache Log4J static logger sample

This sample illustrates the *Define logger properly* rule on how to define a logger for the Apache Log4J framework. See Apache Commons static logger sample on how to define a logger for the Apache Commons logging framework.

Apache Log4J static logger

```java
import org.apache.log4j.Logger;

public class MyClass {
    ...
    private static final Logger LOG = Logger.getLogger(MyClass.class);
    ...
    public void doSomething(final Object obj) {
        LOG.trace("Entering doSomething(Object)");
        ...
    }
}
```

Boolean assignment sample

This sample illustrates the *Assign result of boolean expression directly* rule on how to efficiently assign boolean values.

Boolean assignment

```java
public void foo(final Collection<ElementType> collection,
                final ElementType             element) {
    ...
    boolean elementFound;

    // Replace a boolean assignment like ...
    if (collection.contains(element)) {
        elementFound = true;
    } else {
        elementFound = false;
    }

    // ... or like ...
    elementFound = collection.contains(element)
                   ? true
                   : false;

    // ... by a less expensive and clearer direct assignment:
    elementFound = collection.contains(element);

    ...
}
```

Broken null check samples

This sample illustrates the *Avoid broken or misplaced null checks* rule on avoiding errors caused by broken null checks.

Broken null checks

```java
public void foo(final String name) {

    // These null checks will fail ...
    if ((name == null) && name.isEmpty()) ...
    if ((name != null) || (! name.isEmpty())) ...

    // ... whereas these will work:
    if ((name == null) || name.isEmpty()) ...
    if ((name != null) && (! name.isEmpty())) ...

}
```

Class initialization sample

This sample illustrates the *Ensure initialization of Class objects* rule on avoiding errors by using uninitialized Class objects.

Class initialization

```java
import com.bmw.MyClass;

// Up to Java 1.4, this would initialize the class ...
Class clazz = MyClass.class;

// ... whereas as of Java 5, this is required:
Class<MyClass> clazz = MyClass.class;
Class.forName(clazz.getName()); // Invoke a method on the Class object.
```

Confusing multiple unary operators sample

This sample illustrates the *Avoid multiple unary operators* rule on avoiding errors caused by multiple unary operators in the same statement.

Confusing multiple unary operators

```java
// The subsequent combinations of multiple unary operators ...
int i = +-+1;         // Unnecessary signs.
boolean b = ! ! true; // Double negation.
int j = -~j;          // Negative unary complement of j.
```

```java
// ... can be simplified to better readable:
int i = -1;
boolean b = true;
int j = j+1;
```

Elimination of non short-circuit boolean operator sample

This sample illustrates the *Use short-circuit boolean operators* rule on avoiding errors caused by non short-circuit boolean operators.

Elimination of non short-circuit boolean operator

```java
public void foo(final boolean b,
                final MyClass anObj) {

    // The following non-short-circuit sample ...
    if (b & anObj.booleanMethod()) { // booleanMethod is also called if b == false.
        ...
    }

    // ... is equivalent to:
    boolean retVal = anObj.booleanMethod(); // Assures that booleanMethod is called.
    if (b && retVal) {
        ...
    }
}
```

Ignored return value error sample

This sample illustrates the *Check return value on immutable objects* rule on avoiding errors caused by ignoring the return value of methods of immutable objects.

Ignored return value error

```java
String aString = " Some chars ";

// The following code doesn't modify aString's value...
aString.trim();

// ... whereas this statement does:
aString = aString.trim();
```

Inner assignment sample

This sample illustrates the *Avoid inner assignments* rule on sparsely using inner assignments.

Inner assignment

```java
import java.io.BufferedReader;
...
private void foo(final BufferedReader reader) {
    String line;
    while((line = reader.readLine()) != null) {
        ...
    }
}
...
```

Inner boolean assignment error sample

This sample illustrates the *Don't assign booleans in conditions* rule on avoiding errors caused by boolean assignments in conditions.

Inner boolean assignment error

```java
// This inner boolean assignment causes an infinite loop ...
while (booleanValue = true) {
    ...
}
```

```java
// ... whereas this is a straight forward boolean condition:
while (booleanValue == true) {
    ...
}
```

Intended fall-through sample

This sample illustrates the *Mark fall-through in switch statements* rule on how to deal with intended fall-throughs in switch statements.

Intended fall-through

```java
switch (color) {
    case BLACK:
    case BLUE:
        doSomething();
        // Fallthru
    case GREEN:
        doSomethingElse();
        break;
    default:
        doSomeDefaultHandling();
}
```

Iteration over collection via loop sample

This sample illustrates the *Use loops rather than iterators* rule on how to efficiently iterate over a collection.

Iteration over collection via loop

```java
public void foo(final Collection<ElementType> collection) {
    ...

    // The subsequent iterator ...
    Iterator<ElementType> iter = collection.iterator();
    ElementType element;
    while (iter.hasNext()) {
        element = iter.next();
        ...
    }

    // ... is equivalent to the subsequent loop:
    for (ElementType element : collection) {
        ...
    }

    ...
}
```

Lost logger configuration sample

This sample illustrates the *Don't loose Java Logger configuration* rule by providing a wrong initialization of the Java logger which will work for the Oracle JDK, but fail for OpenJDK. See Robust logger configuration sample o n how to implement a logger configuration that runs correctly both for the Oracle JDK and the OpenJDK.

Lost logger configuration

```java
import java.util.logging.FileHandler;
import java.util.logging.Logger;
...
private static void initLogger() {
    Logger logger = Logger.getLogger("com.bmw.myapp");
    logger.addHandler(new FileHandler());
    logger.setUseParentHandlers(false);
}

// In OpenJDK, the logger is lost when garbage collection
// is called immediately after initialization:
public static void main(final String[] args) {
    initLogger(); // Adds a file handler to the logger.
    System.gc(); // Logger configuration gets lost.
    // This won't be logged to the file as expected:
    Logger.getLogger("com.bmw.myapp").info("My message");
}
```

Misplaced null check sample

This sample illustrates the *Avoid broken or misplaced null checks* rule on avoiding errors caused by misplaced null checks.

Misplaced null check

```java
public void foo(final String name) {

    // This null check will fail ...
    if (name.isEmpty() || (name == null)) ...

    // ... whereas this will work:
    if ((name == null) || (name.isEmpty())) ...

}
```

Non short-circuit boolean operator failure sample

This sample illustrates the *Use short-circuit boolean operators* rule on avoiding errors caused by non short-circuit boolean operators.

Non short-circuit boolean operator failure

```java
// The following code will throw a NullPointerException for a being null ...
if ((a == null) | a.isEmpty()) {
    ...
}
```

```java
// ... whereas this works as expected:
if ((a == null) || a.isEmpty()) {
    ...
}
```

Outer loop index modification in inner loop sample

This sample illustrates the *Don't modify loop indices of for loops* rule on avoiding errors caused by loop index modifications inside the loop's body.

Outer loop index modification in inner loop

```java
for (int i = 0; i < 10; i++) {
    // Illegal modification of i in inner loop:
    for (int j = 0; j < 20; i++) { // Should probably be j++.
        ...
    }
}
```

Return null for primitive error sample

This sample illustrates the *Avoid returning null for primitive wrapper types* rule on avoiding errors caused by returning `null` for a primitive wrapper object.

Return null for primitive error

```java
public static Boolean nullForPrimitive() {
    return null; // Critical return of null for primitive wrapper.
}

public static void callToNullForPrimitive() {
    // This will compile, but throw a NullPointerException:
    boolean b = nullForPrimitive();
}
```

Robust logger configuration sample

This sample illustrates the *Don't loose Java Logger configuration* rule on how to implement a logger configuration that runs correctly on both the Oracle JDK and the OpenJDK. See Lost logger configuration sampl e on how to implement a logger configuration that runs correctly on an Oracle JRE, but fails for OpenJDK.

Robust logger configuration

```java
import java.util.logging.FileHandler;
import java.util.logging.Logger;
...
private static final Logger LOG;

static {
    LOG = Logger.getLogger("com.bmw.myapp");
    LOG.addHandler(new FileHandler());
    LOG.setUseParentHandlers(false);
}

// This code also works with OpenJDK:
public static void main(final String[] args) {
    System.gc(); // Logger configuration isn't lost.
    // This will be logged to the file as expected:
    LOG.info("My message");
}
```

Self-assignment error sample

This sample illustrates the *Don't use ++/-- operators in self assignments* rule on avoiding errors caused by unary operators in self assignments.

Self-assignment error

```java
// This self assignment is vacuous ...
i = i++; // i remains unmodified, as it is assigned before its value is incremented.
```

```java
// ... whereas these assignments will increment i by 1:
i++;
i = i + 1;
```

Simplified boolean return sample

This sample illustrates the *Simplify boolean returns* rule on how to express boolean returns in a brief an clear way.

Simplified boolean return

```java
// Replace ...
if (<boolean expression>) {
    return true;
} else {
    return false;
}
```

```java
// ... with:
return <boolean expression>;
```

Ternary operator sample

This sample illustrates the *Use inline conditionals for assignments which depend on conditions* rule on how express assignments which are guarded by a boolean condition in a short and easy to understand way.

Ternary operator

```java
// The following code ...
if (myParam == null) {
    myVariable = new MyClass();
} else {
    myVariable = myParam;
}
```

```java
// ... can be expressed shorter as:
myVariable = (myParam == null)
             ? new MyClass()
             : myParam;
```

Unintentional fall-through sample

This sample illustrates the *Watch out for unintentional fall-through* rule on unexpected behavior in `case` stateme nts due to a missing `break` or other exit statement from the `case` block.

Unintentional fall-through

```java
String myValue = "zero";

switch (color) {
    case BLACK:
        myValue = "black";
        // Unintended fall-through!!! In this case, myValue will be set to "blue" in the next label.
    case BLUE:
        myValue = "blue";
        break;
    case GREEN:
        myValue = "green";
        break;
    default:
        myValue = "white";
}
```

Concurrency samples

This section contains the samples used to illustrate the rules in the Concurrency chapter.

- Double-checked locking sample — Illustrates errors caused by double checked locking.
- Lock usage without timeout sample — Illustrates how to obtain a lock without a timeout.
- Lock usage with timeout sample — Illustrates how to obtain a lock with a timeout.

Double-checked locking sample

This sample illustrates the *Avoid double-checked locking* rule on avoiding errors caused by double checked locking.

Double-checked locking

```java
package com.bmw.my.package;
...
public class MyClass {
    ...
    private static MyClass instance;
    ...
    public static MyClass instance() {
        // Beware of such code. It might not work as expected.
        if (instance == null) {
            synchronized (Class.forName("com.bmw.my.package.MyClass")) {
                if (instance == null) { // The erroneous double-check.
                    instance = new MyClass();
                }
            }
        }
        return instance;
    }
    ...
}

```

Lock usage without timeout sample

This sample illustrates the *Use Lock.tryLock() with timeout* rule on how to obtain a lock without a timeout. See Lock usage with timeout sample on how to obtain a lock with a timeout.

Lock usage without timeout

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
...
public class MyClass {
    ...
    private final Lock lock = new ReentrantLock();
    ...
    public final void myThreadSafeMethod() {
        ... // Code that does not require synchronization.
        // Acquire lock, possibly waiting forever (see
        // "Lock usage with timeout sample" for a safer implementation):
        this.lock.lock();
        // Execute synchronized code in a try block to ensure proper lock release:
        try {
            ... // Code that requires synchronization on this instance.
        } finally {
            // Reliably release lock in the finally block:
            this.lock.unlock();
        }
        ... // Code that does not require synchronization.
    }
    ...
}

```

Lock usage with timeout sample

This sample illustrates the *Use Lock.tryLock() with timeout* rule on how to obtain a lock with a timeout. See Lock usage without timeout sample on how to obtain a lock without a timeout.

Lock usage with timeout

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import com.bmw.my.package.global.Constants; // The global constants used in my app.
...
public class MyClass {
    ...
    private final Lock lock = new ReentrantLock();
    ...
    public final void myThreadSafeMethod() {
        ... // Code that does not require synchronization.
        try {
            // Acquire lock before timeout:
            if (this.lock.tryLock(Constants.DEFAULT_LOCK_TIMEOUT, TimeUnit.MILLISECONDS)) {
                // Execute synchronized code in a try block to ensure proper lock release:
                try {
                    ... // Code that requires synchronization on this instance.
                } finally {
                    // Reliably release lock in the finally block:
                    this.lock.unlock();
                }
            } else {
                // Error handling in case that lock could not be acquired before timeout:
                ...
            }
        } catch(final InterruptedException e) {
            // Error handling in case that the current thread was interrupted:
            ...
        }
        ... // Code that does not require synchronization.
    }
    ...
}

```

Documentation samples

This section contains the samples used to illustrate the rules in the Documentation chapter.

- Javadoc block comment format sample — Javadoc block comment format for Java source code files.
- Method documentation sample — Illustrates how to document constructors and methods in Javadoc.
- Package documentation sample (package-info.java) — Illustrates how to document Java packages in Javadoc.

- Type documentation sample — Illustrates how to document class, interface, enum and annotation types in Javadoc.
- Variable documentation sample — Illustrates how to document static and member fields in Javadoc.

Javadoc block comment format sample

This sample illustrates the *Javadoc block comment format* for Java source code files.

Javadoc block comment format

```
/**
 * First sentence of the artifact's description with a masked {@literal .} character;
 * this sentence will appear in Javadoc overviews.
 * Further artifact description, which will not appear in Javadoc overviews.
 *
 * @param   parameter <i>Mandatory.</i>
 *          Parameter documentation.
 * @return  Documentation of the value returned by the method.
 * @throws  IllegalArgumentException
 *          if {@code parameter} is {@code null} or empty.
 */
public ReturnType methodName(final ParameterType parameter) {
    ...
}
```

Method documentation sample

This sample illustrates the *Javadoc of constructors and methods* rule on how to document constructors and methods in Javadoc.

Method documentation

```
/**
 * Gets the width of a given text rendered for a given font.
 *
 * @param   text <i>Mandatory.</i>
 *          The text to calculate its width for.
 * @param   font <i>Mandatory.</i>
 *          The font to be used for rendering the {@code text}.
 * @return  the width in pixels of the specified {@code text} for the specified {@code font}.
 * @throws  IllegalArgumentException
 *          if the specified {@code text} or {@code font} are {@code null}.
 * @see     java.awt.FontMetrics#stringWidth(String)
 */
public static int getTextWidth(final String text,
                               final Font   font) {
    ...
}
```

Package documentation sample (package-info.java)

This sample illustrates the *Javadoc of packages* rule on how to document Java packages in Javadoc.

Package documentation (package-info.java)

```
//=======================================================================================================================
// Module: BMW Continuous Integration installer - Utilities.
// Copyright (c) 2011-2014 BMW Group. All rights reserved.
//=======================================================================================================================
/**
 * A collection of utilities for the Continuous Integration custom application build and quality assurance environment.
 * <p/>
 * <ul>
 *   <li>
 *     Class {@linkplain com.bmw.shared.it4it.contint.util.LogoCreator} implements the functionality to
 *     create customized logos for the Continuous Integration server components Jenkins, Nexus and SonarQube.
 *     <p/>
 *   </li>
 *   <li>
 *     ...
 *   </li>
 * </ul>
 *
 * @author   Markus Dieterle <markus.dieterle@bmw.de>
 */
package com.bmw.shared.it4it.contint.util;

```

Type documentation sample

This sample illustrates the *Javadoc of types* rule on how to document class, interface, enum and annotation types in Javadoc.

Type documentation

```java
/**
 * Stand-alone tool to create customized logos for the Continuous Integration
 * server components Jenkins, Nexus and SonarQube.
 *
 * @author   Markus Dieterle <markus.dieterle@bmw.de>
 */
public final class LogoCreator {
    ...
}

```

Variable documentation sample

This sample illustrates the *Javadoc of variables* rule on how to document static and member fields in Javadoc.

Variable documentation

```java
/**
 * The logo type.
 */
private final LogoType m_logoType;
```

Formatting samples

This section contains the samples used to illustrate the rules in the Formatting chapter.

- Class declaration sample — Illustrates how to format the declaration section of a class.
- Confusing preceding whitespace sample — Illustrates confusing whitespace characters before semicolons or certain unary operators.
- Confusing trailing whitespace sample — Illustrates confusing whitespace characters after certain unary operators.
- C-style array definition sample — A C-style array definition, which is not to be used.
- Generics format sample — Illustrates how to format generics.
- Interface declaration sample — Illustrates how to format the declaration section of an interface.
- Java-style array definition sample — A Java-style array definition, which is expected to be used.
- Left curly braces sample — Illustrates where to set left curly braces.
- Method parameter padding sample — Illustrates the padding of method identifiers and method parameter lists.
- Right curly braces sample — Illustrates where to set right curly braces.
- Upper L sample — Illustrates how to enhance code readability for literals of type long.

Class declaration sample

This sample illustrates the *Align type declarations* rule on how to format the declaration section of a class.

Class declaration

```java
public class MyClass
extends      MySuperclass
implements   MyInterface1, MyInterface2 {
...
```

Confusing preceding whitespace sample

This sample illustrates the *Avoid confusing preceding whitespaces* rule on avoiding whitespace characters before semicolons or certain unary operators.

Confusing preceding whitespace

```
// Replace such confusing preceding whitespaces ...
x = i_++;
y = j
--;
```

```
// ... by better readable:
x = i++;
y = j--;
```

Confusing trailing whitespace sample

This sample illustrates the *Avoid confusing trailing whitespaces* rule on avoiding whitespace characters after certain unary operators.

Confusing trailing whitespace

```
// Replace such confusing trailing whitespaces ...
if (x._equals(y)) {...}
x = -_3;
y = ++_i;
z = --
j;
```

```
// ... by better readable:
if (x.equals(y)) {...}
x = -3;
y = ++i;
z = --j;
```

C-style array definition sample

This sample illustrates the *Don't use C-style for array definitions* rule on how to format array definitions. The code shows a C-style array definition which is not to be used. See Java-style array definition sample for a sample of the expected style for array definitions.

C-style array definition

```
public static void main(final String args[]) {
    ...
}
```

Generics format sample

This sample illustrates the *Don't surround types of generics by whitespaces* rule on how to format generics.

Generics format

```
// Replace ...
List<_Integer_> x = new ArrayList<_Integer_>();
List<List_<Integer>_> y = new ArrayList<_List<Integer>_>();
```

```
// ... for Java 5 and 6 by ...:
List<Integer> x = new ArrayList<Integer>();
List<List<Integer>> y = new ArrayList<List<Integer>>();
```

```
// ... and for Java 7 and higher by:
List<Integer> x = new ArrayList<>();
List<List<Integer>> y = new ArrayList<<>>();
```

Interface declaration sample

This sample illustrates the *Align type declarations* rule on how to format the declaration section of an interface.

Interface declaration

```
public interface MyInterface
extends          MySuperInterface, MyInterface2 {
...
```

Java-style array definition sample

This sample illustrates the *Don't use C-style for array definitions* rule on how to format array definitions. The code shows the expected Java-style array definition format. See C-style array definition sample for a sample of an array definition which shall not be used.

Java-style array definition

```
public static void main(final String[] args ) {
    ...
}
```

Left curly braces sample

This sample illustrates the *Position left curly braces at end of line* rule on where to set left curly braces.

Left curly braces

```
public void checkName(final String name) {
    if ((name == null) || (name.isEmpty())) {
        throw ...;
    }
}
```

Method parameter padding sample

This sample illustrates the *Don't separate method identifier from parameter list* rule on avoiding whitespaces between method identifiers and method parameter lists.

Method parameter padding

```
// Replace ...
public static boolean renameFile_(final String oldPathname,
                                  final String newPathname) {
    return new File_(oldPathname).renameTo_(new File_(newPathname));
}

// ... by:
public static boolean renameFile(final String oldPathname,
                                 final String newPathname) {
    return new File(oldPathName).renameTo(new File(newPathname));
}
```

Right curly braces sample

This sample illustrates the *Position right curly braces on a new line* rule on where to set right curly braces.

Right curly braces

```
try {
    if (...) {
        ...
    } else if (...) {
        ...
    } else {
        ...
    }
} catch (...) {
    ...
} finally {
    ...
}
```

Upper L sample

This sample illustrates the *Use upper case L for literals of type long* rule on how to enhance code readability for literals of type `long`.

Upper L

```java
// Replace ...
long unitSize = 1l; // Likely to be read as int 11.
```

```java
// ... by:
long unitSize = 1L; // Clearly readable as long 1.
```

Performance samples

This section contains the samples used to illustrate the rules in the Performance chapter.

- StringBuilder usage sample — Illustrates how to efficiently use the java.util.StringBuilder.
- String conversions sample — Illustrates how to efficiently convert between Strings, primitives and primitive wrapper classes.

StringBuilder usage sample

This sample illustrates the *Don't use String concatenation in StringBuilder/Buffer arguments* rule on how to efficiently use the `java.util.StringBuilder`.

StringBuilder usage

```java
import java.util.StringBuilder;
...
public static String foo(final String name) {
    // Allocate StringBuilder with expected maximum length:
    StringBuilder builder = new StringBuilder(40);

    // Replace concatenated argument ...
    builder.append("Name: " + name); // Unnecessary creates a temporary String object.

    // ... by less expensive:
    builder.append("Name: ")
            .append(name);

    return builder.toString();
}
```

An even briefer implementation is:

```java
import java.util.StringBuilder;
...
public static String foo(final String name) {
    return new StringBuilder(40).append("Name: ")
                                .append(name)
                                .toString();
}
```

String conversions sample

This sample illustrates the *Don't use intermediate objects for conversions between primitives or wrappers and strings* rule on how to efficiently convert between Strings, primitives and primitive wrapper classes.

String conversions

```java
String  myString = "123";
int     myInt;
Integer myInteger;

// (1) String to int conversion

// (1.1) Bad - waste of an object:
myInt = Integer.valueOf(myString).intValue();

// (1.2) Good - wrapper class conversion method:
myInt = Integer.parseInt(myString);


// (2) String to Integer conversion

// (2.1) Bad - waste of an object:
myInteger = Integer.valueOf(Integer.parseInt(myInt));

// (2.2) Good - wrapper class factory method:
myInteger = Integer.valueOf(myString);


// (3) int to String conversion

// (3.1) Bad - waste of an object:
myString = Integer.valueOf(myInt).toString();

// (3.2) Good - wrapper class conversion method:
myString = Integer.toString(myInt);
```

Programming samples

This section contains the samples used to illustrate the rules in the Programming chapter.

- Absolute time calculation sample — Illustrates how to avoid date and time overflows caused by using types with too small value ranges for time representations.
- Append to output stream sample — Illustrates how to avoid errors caused by also opening both a file output stream and a wrapping stream in append mode.
- Arithmetic type error samples — These samples illustrate how to avoid errors caused by intermediate value types which overflow or don't preserve the required precision.
- Bit check sample — Illustrates how to avoid logic errors caused by checking if a bit is set via greater than or smaller than relations rather than via a comparison with zero.
- Broken type check in equals sample — Illustrates how to avoiding errors caused by assuming anything about the equals() method's enclosing type or the type of the argument passed to the method.
- Byte array packaging sample — Illustrates how to avoid arithmetic errors caused by not setting all bits of the byte value to zero except its significant 8 trailing bits when packing a byte array representation of a short, int or long value into a value of its actual type.
- Erroneous array typecast sample — Illustrates errors caused by casting arrays to a super type of their element type.
- Hash code arithmetics sample — Illustrates errors caused by computing the reminder of a hash code modulo another value.
- Incompatible type comparison in equals() sample — Illustrates how to avoid errors caused by equals() methods which return true if the object type of the given parameter is the same as of the enclosing object.

Absolute time calculation sample

This sample illustrates the *Use long values to represent absolute times* rule on avoiding date and time overflows caused by using types with too small value ranges for time representations.

Absolute time calculation

```java
// Fails for dates before Dec 1969 and after Jan 1970:
Date getDate(final int secondsSinceNewYear1970) {
    return new Date(secondsSinceNewYear1970 * 1000);
}

// Fails for dates after 2037:
Date getDate(final int secondsSinceNewYear1970) {
    return new Date(secondsSinceNewYear1970 * 1000L);
}

// Works for all dates:
Date getDate(final long secondsSinceNewYear1970) {
    return new Date(secondsSinceNewYear1970 * 1000L);
}
```

Append to output stream sample

This sample illustrates the *Open wrapped file output streams in random access mode when appending content to the file* rule on avoiding errors caused by also opening the wrapper stream in append mode.

Append to output stream

```java
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
...
public static void appendToFile(final byte[] content,
                                final String filename)
throws IOException {
    final OutputStream       fileOut;
    final ObjectOutputStream objOut;
    try {

        // This code will fail due to the true
        // argument in the next statement...
        fileOut = new FileOutputStream(filename, true);
        objOut = new ObjectOutputStream(fileOut);
        objOut.write(content); // Throws IOException.

        // ... whereas this will work:
        fileOut = new FileOutputStream(filename, false);
        objOut = new ObjectOutputStream(fileOut);
        objOut.write(content); // No exception gets thrown.

        ...
```

Arithmetic type error samples

This sample illustrates the *Ensure required value range and precision for intermediate results of numeric calculations* rule on avoiding errors caused by intermediate value types which overflow or don't preserve the required precision.

Arithmetic type error

```java
// The following statement will return -1.616.567.296:
static final long MILLIS_IN_MAY = 31 * 24 * 60 * 60 * 1000;

// ... whereas this will correctly return 2.678.400.000:
static final long MILLIS_IN_MAY = 31L * 24 * 60 * 60 * 1000;


// The following statement will return 0.0 ...
double value = 1 / 4;

// ... whereas this will correctly return 0.25:
double value = 1 / 4.0d;
```

Bit check sample

This sample illustrates the *Compare with zero to check if bits are set* rule on avoiding logic errors caused by checking if a bit is set via greater than or smaller than relations rather than via a comparison with zero.

Bit check

```java
final byte i = (byte) 0x80;
final byte j = (byte) 0x81;

// The following comparison will fail ...
if ((i & j) ≥ 0) { ... }

// ... whereas this will evaluate true:
if ((i & j) != 0) { ... }
```

Broken type check in equals sample

This sample illustrates the *Don't assume types in equals()* rule on avoiding errors caused by assuming anything about the `equals()` method's enclosing type or the type of the argument passed to the method.

Broken type check in equals

```java
public class Foo {
    ...
    public boolean equals(final Object obj) {
        if (obj == null) {
            return false;
        }

        // This type check will fail when inherited by subclasses ...
        if (Foo.class == obj.getClass()) {

        // ... whereas this will also work for subclasses:
        if (this.getClass() == obj.getClass()) {

            ...
        }
    }
}
```

Byte array packaging sample

This sample illustrates the *Suppress sign extensions with byte packaging* rule on avoiding arithmetic errors caused by not setting all bits of the byte value to zero except its significant 8 trailing bits when packing a `byte` ar ray representation of a `short`, `int` or `long` value into a value of its actual type.

Byte array packaging

```java
final byte[] byteArray = new byte[] { (byte) 0x80,
                                      (byte) 0x81,
                                      (byte) 0x82,
                                      (byte) 0x83, };
int result;

// The subsequent code fails to pack a 4-byte
// big-endian int representation into an int ...
result = 0;
for (int i = 0; i < 4; i++) {
    result = ((result << 8) + byteArray[i]);
}
// result now contains 0x7f808183.

// ... neither works the non short-circuit or operator as in ...
result = 0;
for (int i = 0; i < 4; i++) {
    result = ((result << 8) | byteArray[i]);
}
// result now contains 0xffffff83.

// ... whereas this code works:
result = 0;
for (int i = 0; i < 4; i++) {
    result = ((result << 8) + (byteArray[i] & 0xff));
}
// result now correctly contains 0x80818283.
```

Erroneous array typecast sample

This sample illustrates the *Don't cast array type to super type* rule on avoiding errors caused by casting arrays to a super type of their element type.

Erroneous array typecast

```java
public class A {
    ...
}

public class B
extends      A {
    ...
}

public class C {
...
    public void foo() {
        ...
        B[] b = new B[2];
        A[] a = b;         // Illegal cast of b to super type.
        a[0] = new A();    // Throws ArrayStoreException.
        ...
    }
}

```

Hash code arithmetics sample

This sample illustrates the *Don't compute reminder of hash codes* rule on avoiding errors caused by computing the reminder of a hash code modulo another value.

Hash code arithmetics

```java
// Rather than using ...
int derivedFromHashCode = x.hashCode() % n; // Is < 0 for negative hash codes.

// ... consider using:
int derivedFromHashCode = x.hashCode() & (n - 1); // Is always >= 0.
```

Incompatible type comparison in equals() sample

This sample illustrates the *Don't have equals() compare incompatible operands* rule on avoiding errors caused by `equals()` methods which return `true` if the object type of the given parameter is the same as of the enclosing object.

Incompatible type comparison in equals()

```java
public class Foo {
    private String name;
    ...

    // !!!Beware of such code!!!
    public boolean equals(final Object obj) {
        if (obj instanceof Foo) {
            return this.name.equals(((Foo) obj).name);
        } else if (obj instanceof String) {
            // The illegal comparison:
            return this.name.equals(obj); // Compares String with Foo!
        }
        return false;
    }
}


// In code that invokes such an equals() method, rather compare
// the String with Foo's name property than with the Foo instance:
String name = "Some Name";
Foo    foo  = new Foo();
...
// Replace ...
if (foo.equals(name)) {
    ...
}

// ... by:
if (name.equals(foo.getName()) {
    ...
}
```

Testing samples

This section contains the samples used to illustrate the rules in the Testing chapter.

- JUnit 3 exception check sample — Illustrates how to check if an expected exception is thrown for JUnit 3.

- JUnit 4 exception check sample — Illustrates how to check if an expected exception is thrown for JUnit 4.

JUnit 3 exception check sample

This sample illustrates the *Test if expected exceptions are thrown* rule on how to check if an expected exception is thrown for JUnit 3. See JUnit 4 exception check sample for an equivalent JUnit 4 implementation.

JUnit 3 exception check

```
// Tests Order.add(Option, int):
public void testAddForNullArgument() {
    final Order order = new Order();
    try {
        order.add(null, 2); // IllegalArgumentException expected.
        Assert.fail("No IllegalArgumentException thrown for null argument."); //$NON-NLS-1$
     } catch (final Exception e) {
        if (! (e instanceof IllegalArgumentException)) {
            Assert.fail(e.getClass().getName()
                        + " instead of IllegalArgumentException thrown for null argument."); //$NON-NLS-1$
        }
    }
}
```

JUnit 4 exception check sample

This sample illustrates the *Test if expected exceptions are thrown* rule on how to check if an expected exception is thrown for JUnit 3. See JUnit 3 exception check sample for an equivalent JUnit 3 implementation.

JUnit 4 exception check

```
// Tests Order.add(Option option, int amount) for an invalid option (null) and a valid positive amount argument:
@SuppressWarnings("static-method")
@Test(expected = IllegalArgumentException.class)
public void testAddforNullArgument() {
    final Order order = new Order();
    order.add(null, 2); // IllegalArgumentException expected.
}
```