JPL Java Coding Standard

Java Coding Standard

Added by Klaus Havelund, last edited by Klaus Havelund on Jan 25, 2010 (view change)
Labels: ADD LABELS

JPL Institutional Coding Standard
for the Java Programming Language

Jet Propulsion Laboratory

The research described in this document was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Version: 1.1
Date: January 25, 2010

Table of Contents

introduction

This document presents first version of a JPL institutional coding standard for the Java programming language. The primary purpose of the standard is to help Java programmers reduce the probability of run-time errors in their programs. A secondary, but related, purpose is to improve on dimensions such as readability and maintainability of code.

The standard is meant for ground software programming. The restrictions on ground software are less severe than the restrictions on flight software, mainly due to the richer resources available on ground software computers, and the often less time critical nature of ground applications. However, note that JPL ground software indeed can be mission critical (meaning that a loss of capability may lead to reduction in mission effectiveness). Amongst the most important general differences from the JPL institutional C coding standard for flight software references (JPL-C-STD) are: (1) the Java standard allows dynamic memory allocation (object creation) after initialization, (2) the Java standard allows recursion, and (3) does not require loop bounds to be statically verifiable. Apart from these differences most other differences are due to the different nature of the two languages.

The coding standard consists of 53 rules expressing mostly what programming patterns to avoid. The standard, however, also mentions a few rules concerned with *process*, including recommendations to use static analysis, document and unit test code. Rules are not prioritized, and should generally be considered as **guidelines**. Every rule has exceptions and should not be followed to the extreme.

The rules are formulated on the basis of studies of various resources, including websites, other (often very informal) standards, books, and static analyzers. Many rules have counterparts in one or more of these resources. The references at the end of the document lists all resources used to produce this standard. In addition, relevant resources are mentioned along each rule.

A set of rules such as those presented in this standard can appear arbitrary. They are indeed ultimately a result of personal judgment. An optimal solution would be a universal Java coding standard accepted by the general Java community. Unfortunately such a standard does not currently exist. Perhaps the most important message is that JPL programmers should use state-of-the-art static analyzers to check their code (rule R2). Such analyzers will likely check many of those rules that the Java community agrees on. This document recommends a set of such analyzers.

A rule checker implementing the rules in this standard is currently being developed using the Semmle static analyzer http://semmle.com.

The rules are numbered, and are divided onto groups reflecting constructs of the Java programming language. A group is named in Blue. Each rule within a group is named in Red. Each rule is described in a framed box:

You should follow this standard.

optionally followed by an elaboration (further explanation of the rule), a rationale, exceptions to the rule, examples, comments, and references to resources that have influenced the formulation of the rule.

User comments

User comments are encouraged and can be added at the end of the document or sent by email to: `klaus.havelund@jpl.nasa.gov`.

People contributing to this standard

The standard is being developed by:

- **Klaus Havelund** (`klaus.havelund@jpl.nasa.gov`)
- **Al Niessner** (`al.niessner@jpl.nasa.gov`)

The following additional people at JPL have contributed to the standard via their comments:

- Eddie Benowitz
- Thomas Crockett
- Bob Deen
- Dan Dvorak
- Gerard Holzmann
- Thomas Huang
- Rajeev Joshi
- David Wagner

process

R01. compile with checks turned on

All code should be compiled with all warnings and error checking turned on, with no errors or warnings resulting.

Elaboration

The default operation for some compilers, like `gcc`, is to not expose all the warnings it finds while parsing and compiling the input. Those compilers whose default is to display all the warnings, like `javac`, sometimes have them turned off because a programmer thinks the warnings are too trivial to worry about and explicitly turns them off. The intention with this rule is that the programmer ensures that the compiler will display all of the errors and warnings it could potentially find during the compilation process. The warnings and errors found by the compiler should then be fixed and never masked or filtered out.

It should not be thought that this rule limits the programmer to using only those warnings about syntax and semantics from the compiler. Many of the modern Integrated Development Environments (IDEs) also have warnings for ill formed code statements. When an IDE supports code checking, it should be used as well in the same context as warnings from a compiler. Many of the IDEs allow for configuration of what is a warning and what should be ignored and their defaults usually match the current industry preferences. Development teams should spend a little time to review the defaults to maximize the benefit of having these checks performed. Just as with the compiler, none of the produced warnings should be removed with masking or filtering.

Rationale

Since compiler warnings may suggest a problem not immediately recognized by the programmer, they all should be respected because the problem may result in a runtime error. Allowing problems to fester until runtime is too expensive.

Literature

references (JPL-C-STD), Rule 2 (routine checking).
references (Effective 2008), Item 24 (Eliminate unchecked warnings).

R02. apply static analysis

All code should be verified with a state-of-the-art static source code analyzer, with no errors or warnings resulting.

Elaboration

Compilers check the syntax and type correctness of a program. Static analyzers carry this a bit further and check for common mistakes or misinterpretations of semantics and mark them as issues that require attention. In other words, a compiler accepts all programs that are syntactically and type correct, while static analyzers constrain that space even more. The reported issues should never be filtered or masked out. The idea of zero issues applies even in cases where the static analyzer gives a false positive. If the static analyzer gets confused, then the code causing the confusion should be rewritten so that it becomes clearly valid.

The current recommendation is to use at least one of the following free tools:

- FindBugs : http://findbugs.sourceforge.net - a very solid bug finder tool, use at least this tool
- PMD : http://pmd.sourceforge.net - this tool can be extended by the user (requires some skills)
- CheckStyle : http://checkstyle.sourceforge.net - this tool specifically checks JavaDoc comments

Beyond these tools commercial tools are available:

- Semmle : http://semmle.com
- Coverity : http://www.coverity.com
- SureLogic : http://surelogic.com

The SQI group in Division 31 is offering help in getting started with the free static analysis tools. Application of Semmle can currently only be done on a research license by the LaRS group.

Rationale

As with a compiler, static analysis issues may suggest a problem not immediately recognized by the programmer, and they all should be respected because the problem may result in a runtime error.

Literature

references (JPL-C-STD), Rule 2 (routine checking).

R03. document public elements

Every exported (public) element should be documented with JavaDoc comments.

Elaboration

Class documentation should contain at least the following:

1. a description of what the class is meant for.
2. how the class is supposed to be used - create a new one or just have a singleton etc.
3. listing of any dependencies, such as shared state.
4. whether the class is intended for single or multithread use.

Methods should sufficiently describe the desired behavior to allow a unit test to be developed to check the implementation
for correctness. In order for Methods to meet the desired goal, it is necessary for the documentation to contain at least the following:

1. a description of the general behavior of the method.
2. pre-condition - allowable inputs.
3. post condition - given valid inputs, expected outputs and effect on state
4. thrown exceptions - what conditions cause them be thrown; should include the unchecked exceptions as well as the checked.
5. thread safety - whether or not the method is thread safe or requires the parent to make it thread safe.

The format and syntax of the comments should follow the rules in:

http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

Comments can be checked with the CheckStyle tool: http://checkstyle.sourceforge.net.

Rationale

When developers use foreign Java software, they use JavaDoc to determine what units (classes, methods, fields) represent/do. The IDEs for Java also use JavaDoc and present information to the developer as part of the auto-completion functionality. Documenting the exported elements makes the software usable.

Literature

references (SUN-doc).
references (Effective 2008), Item 44 (Write doc comments for all exposed API elements) and Item 62 (Document all exceptions thrown by each method) and Item 70 (Document thread safety).

R04. write unit tests

Unit tests should be written and used to test individual classes and/or collections of classes.

Elaboration

Some literature suggests to write unit tests before the code, and then develop the code until the unit tests no longer fail. However, this approach may not be natural to everyone. The main point of this approach is that unit tests should be written and the sooner the better.

The recommended tool for unit testing is Junit:

http://www.junit.org

Rationale

Unit testing has been shown to be incredibly effective in flushing out coding errors. When the unit tests are kept up to date with the production code, it also makes code modification much safer as tests will fail in the case of a bad code modification.

Literature

references (JUnit)

names

R05. use the standard naming conventions

The naming conventions proposed in the Java Language Specification (JLS) Section 6.8 should be followed. These are also explained in SUN's Java code conventions: http://java.sun.com/docs/codeconv, specifically Section 9: http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html#367.

Rationale

A common naming convention makes it easier to share code between people and teams, and generally makes it easier to read code.

Literature

references (Effective 2008), Item 56 (Adhere to generally accepted naming conventions).
references (FindBugs), Rule NM_CLASS_NAMING_CONVENTION, NM_FIELD_NAMING_CONVENTION, and NM_METHOD_NAMING_CONVENTION.

R06. do not override field or class names

Field and class names should not be redefined.

Elaboration

Violating examples include hiding a field in a superclass by defining a field with the same name in a subclass, defining a local variable with the same name as a field, defining a method with the same name as its defining class (in which case it may be confused with the constructor of the class), or defining a class with the same name as a class defined elsewhere, including in the Java library.

Rationale

Code clarity is one of the principal guides for developing robust and easily maintainable software. When hiding an entity (field or class) with an entity of the same name, it may lead to flawed assumptions about which entity is being referred to.

Examples

The following class illustrates a redefinition of the Java platform class `String`. The redefinition causes the `main` method to be ignored as the main method (since it now has the wrong type) to be called by the JVM when running the program, causing an exception of the form: `Exception in thread "main" java.lang.NoSuchMethodError: main` to be thrown.

```
// it is not ok to reuse/redefine the class String:

class String {
  private final java.lang.String s;

  public String(java.lang.String s) {
    this.s = s;
  }

  public java.lang.String toString() {
    return s;
  }
}

class StringConfusion {
  public static void main(String[] args) { // not the correct main method
    String s = new String("Hello world");
    System.out.println(s);
  }
}
```

Another example is a method with the same name as the class in which it is defined. The programmer may think that a constructor is being defined, whereas this is not the case: when an object of the class is being created, this method will not be called.

Literature

references (AmbySoft), Section 1.3 What Makes Up a Good Name, Item 6.
references (AmbySoft), Section 3.1.5 Do Not Hide Names.
references (CERT), DCL00-J: Use visually distinct identifiers.
references (CERT), SCP03-J: Do not reuse names.
references (Lea), Recommendations, Item 16.
references (JPL-C-STD), Rule 13 (limited scope).
references (JPL-C-STD), MISRA-C Rules 20.1 and 20.2 (Standard libraries).

referecnes (Puzzlers 2005), Rule 9.3 (Programs that hide entities are difficult to understand).
references (Puzzlers 2005), Rule 9.4 (Programs that shadow entities are difficult to understand).
references (Puzzlers 2005), Rule 9.5 (Programs that obscure entities are difficult to understand).
references (Puzzlers 2005), Rule 9.6 (A method with the same name as its class looks like a constructor).
references (Puzzlers 2005), Rule 9.7 (Programs that reuse platform class names are difficult to understand).
references (Elements 2001), Rule 14 (Do not use names that differ only in case).
references (FindBugs), Rules M_SAME_SIMPLE_NAME_AS_INTERFACE, NM_SAME_SIMPLE_NAME_AS_SUPERCLASS, MF_CLASS_MASKS_FIELD, MF_METHOD_MASKS_FIELD, NM_METHOD_CONSTRUCTOR_CONFUSION, NM_CONFUSING, NM_VERY_CONFUSING_INTENTIONAL, and NM_VERY_CONFUSING.
references (PMD) Rule MethodWithSameNameAsEnclosingClass.

packages, classes and interfaces

R07. make imports explicit

Imports should be explicit, identifying the particular classes imported. This is in contrast to implicit imports that use the `*` operator to include all classes from a package.

Rationale

Explicit is better than implicit as it cannot be misinterpreted. The precedence of Java imports (explicit, package, wildcards) is not well known and often misused leading to hard to find and understand problems. Also, using explicit imports result in a precise list of all dependencies. This makes it easier to comprehend how dependent a class is on other classes (the higher dependency, the more complex the class).

Exceptions to the rule

In case where an application uses a very large number of the classes in a package, or the majority of the classes in a package, implicit imports can be used to improve redability.

Examples

Consider the scenario where a programmer has defined a class `Set` in a package `p`. Consider now that someone else attempts to define a class `UseSet` in package `p`, which intends to use Java's own `java.util.Set` (and not `p.Set`). If the import is explicit, hence `import java.util.Set`, then the right version will be accessed. However, if the import is implicit, hence `import java.util.*`, then `p.Set` will be accessed.

```
package p;

import java.util.HashSet; // ok
import java.util.Set;     // ok

//import java.util.*;     // this would not be ok

public class UseSet {
  private Set<Integer> set = new HashSet<Integer>();

  public void addElement(int i) {
    set.add(i);
  }

  ...
}
```

Literature

references (Lea), Recommendations, Item 1.

R08. do not have cyclic package and class dependencies

Packages and classes should not be dependent on each other in a cyclic manner.

Rationale

Cyclic dependencies make it difficult to understand and modify a software system.

Literature

references (Elements 2001), Rule 104.

R09. obey the contract for equals()

When overriding the `equals()` method, one should ensure that it is:

- reflexive : x.equals(x) for all non-null objects x
- symmetric : x.equals(y) implies y.equals(x) for all non-null objects x and y
- transitive : x.equals(y) and y.equals(z) implies x.equals(z) for all non-null objects x, y and z
- null sensitive : x.equals(null) == false for all non-null objects x
- consistent : x.equals(y) should stay constant if no changes are made to x and y for all non-null objects x and y

Rationale

Various libraries, such as collections depend on the `equals()` method satisfying this contract. If it is violated, program behavior may violate functional requirements, or the program may crash.

Examples

The correct format for the definition of the `equals()` method is illustrated by the following code:

```
class Rover {
  private String name;

  public Rover(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  @Override public boolean equals(Object that) {
    if(that != null && that.getClass() == this.getClass()) {
      Rover thatRover = (Rover)that;
      return name.equals(thatRover.getName()); // ok
    } else {
      return false;
    }
  }
}
```

The argument type must be `Object`, the body must first test whether the argument is an instance of the right class, must then cast to the class in case it is, and finally perform the detailed test for equality.

The following `equals()` method is not symmetric:

```
class Rover {
  private String name;

  public Rover(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  @Override public boolean equals(Object that) {
    if(that != null && that.getClass() == this.getClass()) {
      Rover thatRover = (Rover)that;
      return name.equals(thatRover.getName().toLowerCase()); // not ok
    } else {
      return false;
    }
  }
}
```

For example, consider the following definitions:

```
Rover rover1 = new Rover("msl");
Rover rover2 = new Rover("MSL");
```

In this case:

```
rover1.equals(rover2) == true
```

but:

```
rover2.equals(rover1) == false
```

Literature

references (Effective 2008), Item 8 (Obey the general contract when overriding equals).
references (FindBugs) Rules BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS, EQ_ABSTRACT_SELF, EQ_CHECK_FOR_OPERAND_NOT_COMPATIBLE_WITH_THIS, EQ_SELF_NO_OBJECT, NP_EQUALS_SHOULD_HANDLE_NULL_ARGUMENT, and EQ_UNUSUAL.

R10. define both equals() and hashCode()

If the method `Object.equals` is overridden, so should be the method `Object.hashCode`, and vice-versa.

Elaboration

The following property should be satisfied when defining these two methods. For two given objects `o1` and `o2`:

`o1.equals(o2)` implies `o1.hashCode() == o2.hashCode()`

Similarly, if `compareTo` is overridden, then also `equals` (and consequently `hashCode`) should be. That is, the following contract should be satisfied:

`o1.compareTo(o2) == 0` if and only if `o1.equals(o2)`

The Eclipse programming environment can generate `equals` and `hashCode` for a class. Eclipse will only generate them together. Their definitions will depend on the contents of the class. To generate these methods, have the desired Java class open in the Eclipse editor, then click the Source menu, then choose the "Generate hashCode() and equals()" menu item. One will of course have to review the generated code.

Note that the correctness of the `equals`/`hashCode` contract can be dynamically checked in unit tests.

Rationale

Various libraries, such as collections depend on the equals() and hashCode() methods satisfying this contract. If it is violated, program behavior may violate functional requirements, or the program may crash.

Literature

references (Lea), Recommendations, Item 24.
references (Effective 2008), Item 9 (Always override hashCode when you override equals).
references (Puzzlers 2005), Rule 15.1 (Overriding equals without overriding hashCode can cause erratic behavior).
references (FindBugs), Rule HE_EQUALS_NO_HASHCODE and HE_HASHCODE_NO_EQUALS
references (PMD), Rule OverrideBothEqualsAndHashcode.

R11. define equals when adding fields

When adding fields to a class *C*, which extends a class *C0* (different from `Object`) in which `equals` is overridden (defined), then *C* should also override (define) `equals`.

Elaboration

The fact that class *C0* defines the `equals` method suggests that objects of that class as well of its sub-classes will be tested for equality using this method. Hence, it is an oversight if the `equals` method is not overridden in *C* if it introduces new fields.

Rationale

Violating this rule will result in the contract of the `equals()` method to be violated, see rule obey the contract for equals().

Literature

references (Semmle), Rule: Missing Definition of Equals.

R12. define equals with parameter type Object

The `equals` method should be defined with formal parameter type: `Object`.

Rationale

Violating this rule causes the `equals` method from the superclass (including class `Object`) not to be overridden, and calls of the `equals` method will consequently not have the intended effect.

Examples

The following example illustrates a class with an incorrect definition of the `equals` method (not having `Object` as formal parameter type, and missing the `@Override` annotation), followed by a correct definition of the `equals` method (having `Object` as formal parameter type):

```
class Rover {
  private String name;

  public String getName() {
    return name;
  }

  public Rover(String name) {
    this.name = name;
  }

  // not ok:

  public boolean equals(Rover r) { // type should be Object instead
      return name.equals(r.getName());
  }

  // ok:

  @Override public boolean equals(Object o) {
    if (!(o instanceof Rover)) {
      return false;
    } else {
      Rover rover = (Rover)o;
      return name.equals(rover.getName());
    }
  }
}
```

Note that in the latter correct version of the `equals` method, the method body starts with a test of whether the object `o` is of the right type, in this case `Rover`. This form of test is required for all `equals` methods.

Literature

references (Semmle), Rule: Covariant Equals.

R13. do not use finalizers

The `finalize()` method should not be used.

Elaboration

The method may be called on an object o by the JVM if o gets garbage collected. The method will normally execute various cleanup code. It is not predictable when and even if it is called, as it depends on the JVM and garbage collector used. Due to the lack of predictability of finalizer scheduling, overall performance can be decreased.

An uncaught exception thrown in a finalizer will just be swallowed up by the garbage collector, which will just continue, ignoring the exception.

Rationale

Finalizers are unpredictable, can hide exceptions, and may cause decreased performance.

Literature

references (CERT), OBJ02-J: Avoid using finalizers.
references (Effective 2008), Item 7 (Avoid finalizers).
references (Puzzlers 2005), Rule 7.5 (Finalizers are unpredictable, dangerous and slow).

R14. do not implement the Cloneable interface

The `clone()` method should never be overridden or even called.

Rationale

As stated in (Effective 2008): "*If implementing the Cloneable interface is to have any effect on a class, the class and all of its superclasses must obey a fairly complex, unenforceable, and thinly documented protocol*" ... "*it's safe to say that other interfaces should not extend it, and that classes designed for inheritance should not implement it. Because of the many shortcomings, some expert programmers simply choose never to override the clone method and never to invoke it except, perhaps, to copy arrays*".

Examples

The following example shows a `Sheep` class, and the sub class `Dolly` which defines two alternative ways of creating copies of a `Dolly` object, one using a *copy constructor* and one using a *copy factory*. Normally only one of these alternatives would be provided.

```
class Sheep {
  ...
}

class Dolly extends Sheep {
  // normal constructor:
  public Dolly() {
    ...
  }

  // copy constructor:
  public Dolly(Dolly dolly) {
    ...
  }

  // copy factory:
  public static Dolly newInstance(Dolly dolly) {
    ...
  }
}

class Cloning {
  Dolly dolly  = new Dolly();                // create initial object
  Dolly dolly1 = new Dolly(dolly);           // copy using copy constructor
  Dolly dolly2 = Dolly.newInstance(dolly);   // copy using copy factory
}
```

The `Dolly(Dolly dolly)` constructor creates a new `Dolly` object based on its argument, copying what needs to be copied. Likewise, the alternative static factory method `newInstance(Dolly dolly)` serves the same purpose.

Literature

references (Effective 2008), Item 11 (Override clone judiciously).
references (Puzzlers 2005), Rule 7.6 (Cloned objects can share internal state).

R15. do not call nonfinal methods in constructors

A nonfinal method should not be called from within a constructor.

Elaboration

A method is final if either it is declared within a final class (declared with a `final` modifier), or if the method is declared final with the `final` modifier, or if it is declared `private`. In these cases the method cannot be overridden by a subclass.

Rationale

If such a non-final, non-private method is called in the constructor of the superclass, the call will get dispatched to the sub class, which has not yet been initialized because classes are initialized top down from superclass to sub class, resulting in unpredictable behavior due to access to uninitialized state.

Examples

The following example illustrates the problem:

```
abstract class Car {
  private int speed;

  public Car() {
    speed = computeSpeed(); // not ok
  }

  abstract int computeSpeed(); // nonfinal

  ...
}

class Golf extends Car {
  private boolean gti;

  public Golf(boolean gti) {
    this.gti = gti;
  }

  @Override int computeSpeed() {
    return (gti ? 180 : 150); // called before constructor
  }

  ...
}
```

The `Car` constructor calls the nonfinal (abstract) method `computeSpeed`. This call is dispatched to the `Golf` subclass, where the overriding method refers to the `gti` field. However, this has the default initial value `false` since it has not yet been initialized by the `Golf` constructor. The result is that the `speed` value will always get the value 150.

Literature

references (Effective 2008), Item 17 (Design and document inheritance or else prohibit it).
references (Puzzlers 2005), Rule 7.2 (Invoking an overridden method from a constructor causes method to run before instance is initialized).
references (Elements 2001), Rule 81 (Do not call nonfinal methods from within a constructor).
references (FindBugs), Rule UR_UNINIT_READ_CALLED_FROM_SUPER_CONSTRUCTOR.

R16. select composition over inheritance

Composition should be used in favor of inheritance unless inheritance is the most natural solution. That is, inheritance should not be used as the default way to compose classes.

Elaboration

The terms *inheritance* and *composition* are defined as follows. Consider a class `B` being built using a class `A` as a building block. Using inheritance the definition of `B` might appear as follows (in principle):

```
// inheritance:

class B extends A {
  ...
}
```

Using composition on the other hand, the definition of `B` might appear as follows (in principle):

```
// composition:

class B {
  A a = new A(...);
  ...
}
```

In the latter case all features from `A` that need to be available in `B` have to be redefined in `B` and *delegated* to `A`. This will guarantee that `A` stays encapsulated.

Classes that are meant to be extended should document for each overridable method how it is used in other methods of the class (see example below for an explanation why).

Rationale

Composition allows the used class to be treated as a black box whereas inheritance allows the developer to modify the behavior of the superclass. Extension through composition makes classes more robust to changes in the parent. Extension through inheritance make sharing protected state possible.

Examples

The following example illustrates a situation where method overriding has unexpected effects. The class `InstrumentedHashSet` is an extension of the `HashSet` class with a counter, which counts how many elements have been added to the hash set. The sub class overrides the `add` method and the `addAll` method. The problem arises when calling `addAll(c)`. This method will increase the counter according to how many elements `c` contains, and then call `super.addAll(c)`, which again calls `add(e)` for each element in `c`. Each such call will, however, be a call to the `add` method in the sub class, which will count **again**. Hence the counter will be incremented with 2 times the number of elements in `c`.

```
public class InstrumentedHashSet<E> extends HashSet<E> {
  private int addCount = 0;

  @Override public boolean add(E e) {
    addCount++;
    return super.add(e);
  }

  @Override public boolean addAll(Collection<? extends E> c) {
    addCount += c.size();
    return super.addAll(c);
  }

  public int getAddCount() {
    return addCount;
  }
}
```

The alternative would be to use *composition* where an instance of the `HashSet` class is created as a field inside the `InstrumentedHashSet` class. In this case all methods that should be visible in the `InstrumentedHashSet` class must of course be defined explicitly, yielding an extra burden, but perhaps a price worth paying for a safer class.

```
public class InstrumentedHashSet<E> {
  private HashSet<E> hashSet = new HashSet<E>();
  private int addCount = 0;

  public boolean add(E e) {
    addCount++;
    return hashSet.add(e);
  }

  public boolean addAll(Collection<? extends E> c) {
    addCount += c.size();
    return hashSet.addAll(c);
  }

  boolean contains(Object o) {
    return hashSet.contains(o);
  }

  public int getAddCount() {
    return addCount;
  }
}
```

Literature

references (CERT), OBJ01-J: Understand how a superclass can affect a subclass.
references (CERT), CON00-J: Do not invoke a superclass method or constructor from a synchronized region in the subclass.
references (CERT), CON04-J: Do not call overridable methods from synchronized regions.
references (Effective 2008), Item 16 (Favor composition over inheritance), and Item 17 (Design and document for inheritance or else prohibit it).

fields

R17. make fields private

Fields in a class should be made private with accessor member methods granting access to them to external entities.

Elaboration

The same advice applies to static fields, with the exception that one can expose constants via `static final` fields. However, such exposed constants should denote primitive values or references to immutable objects.

Rationale

The better encapsulated a software package is, the easier it is to learn to use it and the easier it is to maintain it. With good encapsulation implementations can be changed more easily. In particular, making fields public has several drawbacks. First, it gives up control over the internal class structure, making it difficult later to change this structure. Second, by making a field directly accessible from outside the class, it becomes impossible from within the class to control what values are stored in the field. This makes it impossible locally within the class to enforce an invariant on the field. Third, by making a field directly accessible from outside the class, it becomes difficult to program additional actions to be carried out when the field is accessed (without using some form of program transformation, for example using aspect oriented programming). Fourth, and finally, by only providing access to a field through accessor methods, it becomes easier to vary synchronization policies.

Examples

```
// ok since no fields except constant are made public:

public class Rover {
  public static final String DEVELOPER = "JPL";

  private String name;
  private Position position;

  String getName() {
    return name;
  }

  Position getPosition() {
    return position;
  }

  void setPosition(Position position) {
    this.position = position;
  }
}
```

Literature

references (AmbySoft), Section 2.2 Member Function Visibility.
references (AmbySoft), Section 3.2 Field Visibility.
references (AmbySoft), Section 3.4 The Use of Accessor Member Functions.
references (AmbySoft), Section 6.1.5 Minimize the Public and Protected Interface.
references (CERT), SEC05-J: Minimize accessibility of classes and their members.
references (CERT), SCP00-J: Use as minimal scope as possible for all variables.
references (CERT), OBJ00-J: Declare data members private.
references (GeoSoft), Rule 44 and Rule 48.
references (GeoSoft), Rule 46.
references (Lea), Recommendations, Item 8 and Item 14 and Item 15.
references (Macadamiam), Naming Conventions, Function Names, last paragraph.
references (SUN-code), page 16, Section 10.1 (Providing Access to Instance and Class Variables).
references (JPL-C-STD), Rule 13 (limited scope).
references (Effective 2008), Item 6 (Eliminate obsolete object references) and Item 45 (Minimize the scope of local variables).
references (Effective 2008), Item 13 (Minimize the accessibility of classes and members), and Item 14 (In public classes, use accessor methods, not public fields).
references (Elements 2001), Rule 71 (Make all fields private).
references (FindBugs), Rules MS_FINAL_PKGPROTECT, MS_MUTABLE_ARRAY, MS_MUTABLE_HASHTABLE, MS_OOI_PKGPROTECT, and MS_PKGPROTECT.
references (PMD), Rule SingularField.

R18. do not use static mutable fields

There should be no static mutable fields. A static field is mutable if it is not declared final, or if it is declared final but points to a mutable object.

Rationale

Static mutable fields act like globals in non-OO languages. They make methods more context-dependent, hide possible side effects, sometimes present synchronized access problems.

Exceptions to the rule

An exception is a singleton, where a class is instantiated to a single object, which is needed to coordinate actions across the system. In this case it would become cumbersome to pass this object around as parameter to constructors of classes referring to the object.

Another exception is a counter used to keep track of the use of a class, for example counting how many objects are created of the class.

Literature

references (Lea), Recommendations, Item 10.
references (CERT), OBJ31-J: Do not use public static non-final variables.
references (FindBugs), Rule MS_CANNOT_BE_FINAL.

R19. declare immutable fields final

The `final` keyword should be used to declare fields that never have their values changed after construction and are intended to be constant.

Elaboration

If a constant is not declared `static`, every instance of the class in which it is defined will retain its own copy of the constant. Hence, if the constant is the same across objects, it should be declared `static final`. If the field forms part of a set of related, and *alternative*, constants, an enumeration type should be used.

Rationale

Using the keyword `final` improves robustness and maintainability because the compiler, static analysis tools, and thread analysis tools can help enforce the explicitly enforced concept.

Exceptions to the rule

The typical exceptions are containers. Consider for example a vector field. This field will stay constant whereas of course its contents will change. See example. It would cause notational clutteredness to declare such containers `final` and would furthermore potentially cause a reader to wrongly believe that the container itself cannot change, which it can! That is, the `final` modifier only refers to the reference to the object, not the contents of the object, which still can be mutable.

Examples

```
public class Constants {
  static final double PI = 3.14159265;                  // ok
  final double thisPI = 3.14159265;                     // ok, but could also be static
  double maybePI = 3.14159265;                          // not ok, should be final and preferably also static
  List<Double> doubles1 = new Vector<Double>();         // ok, although doubles1 reference does not change value
  final List<Double> doubles2 = new Vector<Double>(); // ok, but not necessary, contents is mutable anyway

  void tryit() {
    doubles1.add(PI); // reference itself does not change, contents does
    doubles2.add(PI); // reference itself does not change, contents does
  }
}
```

Literature

references (Lea), Recommendations, Item 12.
references (CERT), DCL31-J: Qualify mathematical constants with the static and final modifiers.
references (CERT), OBJ03-J: Be aware that a final reference may not always refer to immutable data.
references (JPL-C-STD), Rule 13 (limited scope).
references (PMD), Rule ImmutableField.

R20. initialize fields before use

Fields should be explicitly initialized before used. Specifically, it should be ensured that static fields have sensible values even if no instances are ever created. Use static initializers (`static`) if necessary. All fields should be initialized in constructors (or object factories if used instead of constructors).

Rationale

This discipline helps preventing null pointer exceptions.

Literature

references (AmbySoft), Section 3.5 Always Initialize Static Fields.
references (Lea), Recommendations, Item 9.
references (Lea), Recommendations, Item 18.
references (FindBugs), Rules NP_UNWRITTEN_FIELD, UWF_UNWRITTEN_FIELD, and UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR.
references (PMD), Rule DataflowAnomalyAnalysis.

methods

R21. use assertions

Use assertions to check method boundary conditions.

Elaboration

Every class may supply a method, say '`boolean invariant()`', to perform a sanity check on the class that will verify if the class invariant holds.

Each non-trivial method in a class should have the following assertions.

Pre-condition - before performing its actions:

- parameter check: the method should verify that all the parameters passed to the method are valid.
- method specific class check: the method should verify that the class is in a proper state for this method to be called. For example, you can't pop an element from an empty stack.
- class invariant check: the method may call the invariant method to verify that the class invariant still holds.

Intermediate assertions - throughout method body:

- assertions should be used to perform basic sanity checks throughout the code, for example expressing loop invariants.

Post-condition - after performing its actions the following check should be performed:

- return value check: the method may verify that the value it returns is correct.
- method specific class check: the method may verify that the state of the class changed correctly.
- class invariant check: the method may call the invariant method to verify that the class invariant still holds.

Note that pre-condition checks can be interpreted more liberally to include checks on any form of input. This would include program parameters and information read from files. However, external input should be validated with if-statements (see below), not via assertions.

The expressions occurring in assertions should be side effect free.

Java provides an assertion feature, using one of two forms:

```
    assert Expression;

    assert Expression1 : Expression2;
```

Be Careful

In order for such assertions to be enabled the code has to be compiled with option `-ea` (enable assertions). This has the advantage that assertions can be disabled in case they are computationally expensive. The fact that assertions are not activated by default, however, has the disadvantage that unless care is taken, the assertions may not be checked.

Assertions cause exceptions to be thrown, and unless caught such assertions cause program termination. One can also define a home grown set of assertion methods, for example as follows:

```
  class Assert {
    public static final boolean CONTRACT = true;

    public static void preCondition(boolean b) {
      if (CONTRACT && !b) {
        // some reaction
      }
    }

    public static void postCondition(boolean b) {
      if (CONTRACT &&!b) {
        // some reaction
      }
    }
  }
```

They can then be called, for example conditionally on the value of the `CONTRACT` constant:

```
// import all static Contract fields and methods:
import static contract.Contract.*;

class MyMathClass {
  double sqrt(double x) {
    if (CONTRACT) preCondition(x >= 0);
    double result;
    ...
    result = ...;
    if (CONTRACT) postCondition(result >= 0 && Math.abs((result * result) - x) <= EPSILON);
    return result;
  }
}
```

The Java compiler will perform dead code elimination, and in case `CONTRACT` is `false`, the above contract pre- and post-condition calls will be eliminated by the compiler.

A sub class method that overrides a superclass method should preserve the pre- and post conditions of the superclass method. This can be done by making the superclass method final and let it call a protected nonfinal method, which is then overridden by the sub class. The final superclass method will then check the pre- and post conditions.

Rationale

Checking pre-conditions, and reacting locally to violations, make functions able to handle any context in which they are applied. Checking post-conditions and intermediate assertions help ensure functional correctness of the methods. Liberal use of assertions increase the chance of detecting bugs during testing.

Literature

references (CERT), EXP31-J: Avoid side effects in assertions.
references (CERT), SDV00-J: Always validate user input.
references (CERT), SDV15-J: Library methods should validate their parameters.
references (Macmadiam), Testing/Debug Support, Class Invariant and Assertions and Defensive Programming
references (JPL-C-STD), Rule 15 (checking parameter values).
references (JPL-C-STD), Rule 16 (use of assertions).
references (Effective 2008) Item 38 (Check parameters for validity).
references (Elements 2001), Rule 89 (Program by contracts), Rule 90 (Use dead code elimination to implement assertions), Rule 91 (Use assertions to catch logic errors in your code), and Rule 92 (Use assertions to test pre- and postconditions of a method).
references (FindBugs), Rule NP_ARGUMENT_MIGHT_BE_NULL.

R22. use annotations

Annotations should be used to indicate intentions that can then be checked by the compiler or by static analyzers.

Elaboration

Specifically, a method that overrides a method in a superclass should be defined with an `@Override` annotation to indicate the intent.

Method return types and method/constructor argument types can in addition be annotated with `@Null` to indicate the possibility of a null value to be returned, respectively passed as an argument. The `@Null` annotation has to be user-defined as follows:

```
@interface Null{}
```

The `@Null` annotation, however, is somewhat controversial since it is not a standard annotation supported by the Java compiler. FindBugs (FindBugs) has an annotation for non-nullness, hence the opposite, and checks accordingly. It might be better to conform with FindBugs. However, recent research (SpeC# http://research.microsoft.com/en-us/projects/specsharp at Microsoft Research) suggests that assuming null free types as a default is the most effective approach, an approach that has now also been adopted for the Eiffel programming language (http://www.eiffel.com).

Rationale

Making explicit the intent to override a parent method leads to early detection of typos. When overriding a method in a superclass it is very easy to either misspell the method name or indicate wrong argument types. In such a case the method in the superclass will mistakenly not be overridden, but instead a new method will be defined in the sub class. The Java compiler may not detect this.

`@Null` annotations make explicit what input/output values can be null. It allows the developer to devote the time and effort to defensively protect against generating a `NullPointerException` for those items that can be null. It also allows static analyzers to provide early warning of violations to prevent difficult to find and understand runtime errors.

Examples

The following code illustrates the use of `@Override` annotations:

```
// ok:

@Override public int hashCode() {
  return location.getX() + location.getY();
}

// ok, except for the miss-spelling of hashcode,
// but this can now be caught by the compiler:

@Override public int hashcode() { // should be hashCode
  return location.getX() + location.getY();
}

// not ok, @Override is missing, and
// miss-spelling of hashcode will not be caught:

public int hashcode() {
  return location.getX() + location.getY();
}
```

The following code illustrates the use of `@Null` annotations:

```
// ok : annotations used to indicate potential for null values

@interface Null{}

class NullAnnotation {
  public @Null Result computeResult(@Null Data data) {
    if (data == null) {
      return null;
    } else {
      return new Result(data);
    }
  }
}
```

Literature

referecnes (Effective 2008), Item 36 (Consistently use the Override annotation).
references (Puzzlers 2005), Rule 9.1 (It is easy to overload when you intend to override).
references (FindBugs), Rules NP_BOOLEAN_RETURN_NULL, NP_TOSTRING_COULD_RETURN_NULL, and NP_NONNULL_RETURN_VIOLATION.
references (PMD), Rules FinalizeOverloaded, SuspiciousHashcodeMethodName and SuspiciousEqualsMethodName.

R23. restrict method overloading

Method overloading on argument types should be restricted to cases where the argument types are *radically different*.

Elaboration

Two types are radically different if it is impossible to cast or auto-box/auto-unbox a value of one type to the other. Overloading on arity is unproblematic, as in having a one-argument version versus a two-argument version. Note that a variable argument method can be considered as an infinite set of overloaded methods, each with a finite number of arguments, and each of which should satisfy the above rule about radically different arguments with respect to other methods with the same number of arguments.

Rationale

Java method resolution is static, based on the listed types, not the actual types of arguments. Consequently, the results can be counterintuitive.

Examples

As an example, the following program violates this rule. It outputs "`object`", contrary to what one might think.

```
class Classifier {
  String identify(Object x)  { return "object"; }
  String identify(Integer x) { return "integer"; } // overloading violates rule
}

class Relay {
  String relay(Object obj) { return (new Classifier()).identify(obj); } // here we loose the fact that obj is an Integer
}

public class Overloading {
  public static void main(String[] args) {
    Relay relayer = new Relay();
    Integer i = new Integer(17); // it is indeed an Integer
    System.out.println(relayer.relay(i));
  }
}
```

The problem arises because the `identify` method in class `Classifier` is overloaded with two types that are not *radically different* since instances of the `Integer` type can be cast into instances of the `Object` type (casting is not even required, forming a trivial case of casting).

Literature

references (CERT), DCL02-J: Do not overload variable argument methods.
references (Lea), Recommendations, Item 21.
references (Effective 2008), Item 41 (Use overloading judiciously).
references (Puzzlers 2005), Rule 9.2 (Overload-resolution rules are not obvious).

R24. do not assign to parameters

One should not reassign values to parameters. Use local variables instead.

Rationale

Assignments to parameters will only have local effect, but may give the impression of having global effect. Avoid the confusion by not assigning to parameters. Parameters are meant to be inputs to the method and are not meant to function as local variables.

Literature

references (FindBugs), Rule IP_PARAMETER_IS_DEAD_BUT_OVERWRITTEN.
references (PMD), Rule AvoidReassigningParameters.

R25. do not return null arrays or collections

One should not return the `null` value from a method that (type-wise) returns an array or a collection. Instead return an empty array or collection.

Rationale

Returning null puts an obligation on the caller of the method to check that the returned value is not null before any operations can be applied to it. Such checks are easily omitted causing difficult to find and understand runtime failures.

Comments

A stronger rule would be that a method should never return `null`. The current rule limits the restriction to only methods returning arrays or collections.

Literature

references (Effective 2008), Item 43 (Return empty arrays or collections, not null).
references (Puzzlers 2005), Rule 8.7 (Returning null instead of a zero-length array or collection is error prone).
references (FindBugs), Rule PZLA_PREFER_ZERO_LENGTH_ARRAYS.

R26. do not call System.exit

One should not call `System.exit`.

Rationale

Calling `System.exit` in a thread in a multi-threaded application can have as consequence that some other thread doing some important job will abruptly be killed. This may have as a result that the application may leave an externally persistent resource in a bad state.

Literature

references (FindBugs), Rule DM_EXIT.

declarations and statements

R27. have one concept per line

There should be no more than one statement or variable declaration per line.

Rationale

Very long declarations introducing several variables may be hard to read, especially if some of them have array brackets [] associated. Furthermore, multi declarations may lead to confusion if they contain initializations. Finally, it becomes impossible to comment each variable declaration. The rule has likely originated from C/C++ standards, where pointers can be incorrectly used, and where it perhaps the rule has a stronger motivation than in Java. The rule is, however, included here since having one declaration per line simply is easier to read.

Examples

```
// not ok - all declarations on one line:
class MultipleDecls {
  int i, j = 1, x[], y = 2, z;
  ...
}

//ok - declarations on individual lines:
class SingleDecls {
  int i;
  int j = 1;
  int[] x;
  int y = 2;
  int z;
  ...
}
```

Note the declaration of `x` in the `SingleDecls` class: the array specifier `[]` is associated with the type `int` rather than with the identifier `x`. This is a recommended style, although not a rule in itself due to its low priority.

Literature

references (AmbySoft), Section 2.4.6 Write Short, Single Command Lines.
references (AmbySoft), Section 4.2 Declaring and Documenting Local Variables, Item 1.
references (CERT), DCL04-J: Do not declare more than one variable per declaration.
references (SUN-code), page 6, Section 6.1 (Number Per Line).
references (SUB-code), page 11, Section 7.1 (Simple Statements).
references (JPL-C-STD), Rule 24.

R28. use braces in control structures

Braces should be used to group statements that are part of a control structure, such as an `if-else` or `for` statement.

Rationale

This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces. It also removes the risk of *dangling else* problems (determining what `if`-construct an `else`-construct belongs to).

Examples

The following example illustrates the basic principle:

```
// not ok:

for(int i = 0; i < 100; i++)
  for (int j = 0; j < 100; j++)
    process(i,j);

// ok:

for(int i = 0; i < 100; i++) {
  for (int j = 0; j < 100; j++) {
    process(i,j);
  }
}
```

The following example illustrates the *dangling else* problem. In the first code snippet a wrong indentation gives the impression that the `else`-construct belongs to the outermost `if`-construct. With braces such as misunderstanding is not possible. Note that the two code snippets do not have the same semantics.

```
// not ok:

if (x >= 0)
  if (x > 0)
    positive(x);
else // matches most recent if
  negative(x); // will be called if x==0

// ok:

if (x >= 0) {
  if (x > 0) {
    positive(x);
  }
} else {
  negative(x);
}
```

Literature

references (GeoSoft), Rule 72 (argues against this rule).
references (SUN-code), page 7, Section 7.2 (Compound Statements), third bullet.
references (JPL-C-STD), MISRA-C Rules 14.8 and 14.9 (Control flow).
references (Elements 2001), Rule 76 (Use block statements ...).
references (PMD), Rules IfStmtsMustUseBraces, WhileLoopsMustUseBraces, IfElseStmtsMustUseBraces, ForLoopsMustUseBraces.

R29. do not have empty blocks

Empty code blocks '{}' should be avoided.

Rationale

An empty block can indicate that the programmer has forgotten to fill it out. In some cases the programmer may have taken a conscious decision to leave it empty, but this is not recommended.

Exceptions to the rule

There are exceptions to the rule as described in the following. In each case a comment of the form:

```
// @empty-block
```

should be inserted inside the block indicating the intent.

One exception is the body of a method that is intended to be overridden in a subclass. Note, however, that if the method is intended to always be overridden by any subclass, the method should be defined as `abstract`, or be part of an interface to force a compiler check.

Another exception is a block that remains to be implemented. In this case it might be preferable to indicate the intended program structure (and introduce an empty block), and fill out the details later, rather than defer laying out this structure.

A `catch` block should never be empty. It is not enough to just write a comment.

Examples

```
// ok:

try { ... }
catch {
  // @empty-block
  // this is to be filled out when I know what to do
}

// not ok:

try { ... }
catch {
  // this is to be filled out when I know what to do
}

// not ok:

try { ... }
catch { }
```

Literature

references (Effective 2008), Item 65 (Don't ignore exceptions).
references (Elements 2001), Rule 66 (Label empty statements) and Rule 87 (Do not silently absorb a run-time or error exception).
references (FindBugs), Rules UCF_USELESS_CONTROL_FLOW and UCF_USELESS_CONTROL_FLOW_NEXT_LINE.
references (PMD) Rules EmptyCatchBlock, EmptyIfStmt, EmptyWhileStmt, EmptyTryBlock, EmptyFinallyBlock, EmptySwitchStatements, UncommentedEmptyMethod, UncommentedEmptyConstructor, and EmptyFinalizer.

R30. use breaks in switch statements

An unconditional `break` statement should terminate every non-empty `switch` clause.

Rationale

First of all, not adding a `break` statement to the end of a `case` block can indicate an error. Second, consider a `case` block without a terminating `break`. If a user later adds a new `case` after that case, suddenly an unintended fall-through error may be introduced. An exception is empty `case` blocks since it there is obvious that a fall-through is intended.

Examples

```
switch(alarm) {
  case ALARM1:
    // ok, no break needed since case is empty
  case ALARM2:
    handleAlarm1and2();
    // not ok, a break is needed
  case ALARM3:
    handleAlarm3();
    break; // ok
  case ALARM4:
    handleAlarm4();
    break; // ok
  default:
    handleError();
}
```

Literature

references (GeoSoft), Rule 70.
references (SUN-code) page 8, Section 7.8 (switch Statements): suggests that one at least writes a comment.
references (JPL-C-STD), MISRA-C Rule 15.2 (Switch statements).
references (Puzzlers 2005), Rule 5.1 (Missing break in switch case causes fall-through).
references (Elements 2001) Rule 65 (Add a fall-through comment ...) and Rule 78 (Always code a break statement ...).
references (FindBugs), Rules SF_DEAD_STORE_DUE_TO_SWITCH_FALLTHROUGH, SF_DEAD_STORE_DUE_TO_SWITCH_FALLTHROUGH_TO_THROW, and SF_SWITCH_FALLTHROUGH.
references (PMD), Rule MissingBreakInSwitch.

R31. end switch statements with default

The final clause of a `switch` statement should be the `default` clause. In general it is recommended to only use `switch` statements over enumerated types.

Elaboration

The use of enumerated types does not automatically produce warnings or errors in the situation a case is missing. However, in certain programming environments such a check can be turned on. For example in Eclipse on can set `` `enum not covered in switch' `` to `Warning` or `Error` in `preferences->java->compiler->errors/warnings`. This will cause missing cases to be detected, makin the `default` unnecessary.

Rationale

By introducing a `default` clause the programmer is forced to consider what should happen in case not all cases are covered by the `switch` statement. Without a `default` statement, the `switch` statement may silently fail to match a case. No warnings or errors will be issued at compile time or runtime.

Examples

The following example introduces three different `switch` statements, each of which is missing a case, and each of which is missing a default statement (commented out). Neither the Java compiler, nor the Eclipse environment will emit any warnings about this code. Running this program will produce no output at all and cause no warnings or errors to be issued.

```
public class SwitchDefault {
  enum CoinSide {Heads, Tails, Edge;
    void switchInternalEnum() {
      switch (this) {
        case Heads : System.out.println("Heads"); break;
        case Tails : System.out.println("Tails"); break;
        // default : System.out.println("must be Edge"); break;
      }
    }
  };

  void switchExternalEnum(CoinSide coinSide) {
    switch(coinSide) {
      case Heads : System.out.println("Heads"); break;
      case Tails : System.out.println("Tails"); break;
      // default : System.out.println("must be Edge"); break;
    }
  }

  void switchInt(int coinSide) {
    switch(coinSide) {
      case 1: System.out.println("Heads"); break;
      case 2: System.out.println("Tails"); break;
      // default : System.out.println("not 1 or 2"); break;
    }
  }

  public static void main(String[] args) {
    CoinSide.Edge.switchInternalEnum();
    SwitchDefault s = new SwitchDefault();
    s.switchExternalEnum(CoinSide.Edge);
    s.switchInt(3);
  }
}
```

Literature

references (SUN-code) page 8, Section 7.8 (switch Statements).
references (JPL-C-STD), MISRA-C Rule 15.3 (Switch statements).
references (FindBugs), Rule SF_SWITCH_NO_DEFAULT.
references (PMD), Rule SwitchStmtsShouldHaveDefault.

R32. terminate if-else-if with else

All `if-else` constructs should be terminated with an `else` clause.

Rationale

By introducing an `else` clause, the programmer is forced to consider what should happen in case not all previous alternatives are chosen. A missing `else` clause might indicate a missing case handling.

Examples

The following program violates the rule:

```
public class IfElse {
  static final int LIMIT1 = 25;
  static final int LIMIT2 = 50;

  public static void printRange(int x) {
    if (x < LIMIT1) {
      System.out.println("below " + LIMIT1);
    } else
      if (x <= LIMIT2) {
        System.out.println("between " + LIMIT1 + " and " + LIMIT2);
      }
      // not ok, should not have been commented out:
      //else {
      //  System.out.println("above " + LIMIT2);
      //}
  }

  public static void main(String[] args) {
    printRange(75);
  }
}
```

Running this program yields no output.

Literature

references (JPL-C-STD), MISRA-C Rule 14.10 (Control flow).

expressions

R33. restrict side effects in expressions

Side effects should be limited as follows:

1. the assignment operator =, its derived compound assignment forms (such as +=), and the increment ++ and decrement -- operators should only occur in *statement expressions* (as statements).
2. side effects should not occur in:
    a. boolean conditional expressions as part of conditional statements (`if`, `while`, `do`), in expressions in the control-part of `for` loops, or in the expression in `switch` statements.
    b. sub-expressions of composite expressions built by applying Java's built in operators (`&&`, `||`, `+`, ...).
    c. actual argument expressions in method and constructor calls.

Rationale

As a general rule, side effects in contexts where several things happen at the same time makes code difficult to read and debug. As a special case, occurrences of = in expressions can be typos. The Java compiler will specifically not catch those cases where the variable assigned to is a `boolean`.

Examples

The following example illustrates the first three kinds of violations mentioned above:

```
public class LimitSideEffects {
  private int field = 0;

  // Method violates the rules 1, 2.1 and 2.2 above:
  // - an if-statement has a conditional with side effects
  // - a composite expression {{(field += x) <= 100}} has a sub-expression {{(field += x)}} with side effects
  // - an assignment operator (+=) is used in an expression that is not a statement expression

  public void add(int x) {
    if ((field += x) <= 100) { // the three violations
      System.out.println("field " +  field);
    }
  }

  public static void main(String[] args) {
    LimitSideEffects limit = new LimitSideEffects();
    limit.add(10);
    }
  }
```

Literature

references (AmbySoft), Section 2.5.2 Place Constants on the Left Side of Comparisons.
references (CERT), EXP30-J: Do not depend on operator precedence while using expressions containing side effects.
references (GeoSoft), Rule 56.
references (Lea), Recommendations, Item 36.
references (SUN-code), page 17, Section 10.4 (Variable Assignments).
references (JPL-C-STD), Rule 19.
references (JPL-C-STD), MISRA-C Rule 12.13 (Expressions).
references (Puzzlers 2005), Rule 4.2 (Operands of operators are evaluated left to right) - this rule just recommends not to have multiple assignments to the same variable in the same expression.
references (PMD), Rule AssignmentInOperand.

R34. use named constants for non-trivial literals

Literals (specifically integers, floating point numbers, and strings) should as a general rule not be coded directly, except for the simple integer values - `1`, `0`, and `1`, which can appear in a for loop as counter values. Instead such literals should be defined as constants (`static final`).

Rationale

Extensive use of literals within a program can lead to two problems: first, the meaning of the literal is often obscured or unclear from the context (magic numbers), and second, changing a frequently-used literal requires the entire program source code to be searched for occurrences of that literal, creating possible error sources if some of the occurrences are overlooked.

Examples

```
final int LENGTH = 173;
int[] elements1 = new int[LENGTH]; // ok

int[] elements2 = new int[173];    // not ok
```

Literature

references (CERT), DCL03-J: Use meaningful symbolic constants to represent literal values in program logic.
references (CERT), EXP07-J: Do not diminish the benefits of constants by assuming their values in expressions.
references (GeoSoft), Rule 57.
references (Macadamiam), Programming Conventions, Constants and Enumerations.
references (SUN-code), page 17, Section 10.3 (Constants).
references (Puzzlers 2005), Rule 4.1 (Mixed-type computations are confusing), specifically: prefer constant variables to inline magic numbers.

R35. make operator precedence explicit

In compound expressions with multiple sub-expressions the intended grouping of expressions should be made explicit with parentheses. Operator precedence should not be relied upon as commonly mastered by all programmers.

Rationale

The expression will be easier to read.

Examples

```
a + b * c + d   // not ok
a + (b * c) + d // ok
```

Literature

references (AmbySoft), Section 2.4.7 Specify the Order of Operations.
references (CERT), EXP09-J: Use parentheses for precedence of operation.
references (SUN-code), page 18, Section 10.5.1 (Parentheses)
references (JPL-C-STD), Rule 18.
references (JPL-C-STD), MISRA-C Rule 12.1 (Expressions).
references (Puzzlers 2005), Rule 4.3 (Operator precedence is not always obvious).

references (Elements 2001), Rule 77 (Clarify the order of operations with parentheses).

R36. do not use reference equality

One should normally use the `equals` method instead of the == and `!=` operators when comparing objects. This includes `String` objects, and all boxed primitives such as `Integer` and `Long`.

Rationale

Equality == refers to the same object whereas `equals()` refers to the same attributes. Using == with dynamically allocated objects is usually suspect.

Exceptions to the rule

The `equals` method should not be used to compare arrays (just as == should not be used), since it will not perform the expected element-wise comparison. That is, for two arrays `a` and `b`: `a.equals(b)` has the same value as `a == b`. Instead use `java.util.Arrays.equals`.

There are cases where one wants to use == to check that the object references are the same. One case is the null check for some object `o`: `o == null`, which of course is allowed. There are other cases where object identity per pointer is essential, or where object identity is ensured, for example when the `intern` method has been used on strings to canonize string objects (two strings which are equal according to the `equals()` method become represented by the same one object). Another exception is singleton classes. The use of static factory methods over constructors facilitates instance control which in turn limits the effective number of instances of an immutable class to one. As a result, for two objects a and b, a.equals(b) is true when a==b.

Examples

```
String s1 = "Java";
String s2 = "Java";
if (s1 == s2) { // not ok
  ...
}

if (s1.equals(s2)) { // ok
  ...
}
```

Literature

references (CERT), EXP03-J: Do not compare String objects using equality or relational operators.
references (CERT), EXP32-J: Do not use the equal and not equal operators to compare boxed primitives.
references (Lea), Recommendations, Item 30.
references (Puzzlers 2005), Rule 4.4 (Operators == and `!=` perform reference comparisons on boxed primitive types).
references (Elements 2001), Rule 79 (Use equals(), not == ...).
references (FindBugs), Rules ES_COMPARING_PARAMETER_STRING_WITH_EQ, RC_REF_COMPARISON_BAD_PRACTICE, RC_REF_COMPARISON_BAD_PRACTICE_BOOLEAN, and RC_REF_COMPARISON.
references (PMD), Rule CompareObjectsWithEquals and UseEqualsToCompareStrings.

R37. use only short-circuit logic operators

The Boolean non-short-circuit operators `|` and `&` should not be used. Note that the corresponding Integer operators are not covered by this rule (they are allowed).

Rationale

Non-short-circuit logic causes both sides of the expression to be evaluated even when the result can be inferred from knowing the value of the left-hand side. If the right hand side has a side effect it will get executed no matter what the left hand side evaluates to. The programmer's intension may have been different. The rule's intent is to avoid this form of misunderstandings.

Examples

```
// not ok: the & operator causes wheel to be de-referenced
// even when wheel is null:

if (wheel != null & wheel.unlocked()) {
  wheel.turn();
}
```

Literature

references (CERT), EXP06-J: Be aware of the short-circuit behavior of the conditional AND and OR operators. This rule in fact argues that it is the short-curcuit operators `&&` and `||` that are dangerous, hence the contra view.
references (FindBugs), Rule NS_DANGEROUS_NON_SHORT_CIRCUIT.

R38. do not use octal values

Integer literals should not start with '`0`'. An integer starting with `0` is interpreted as an Octal value.

Rationale

The fact that integer literals starting with '`0`' are interpreted as Octal numbers is extremely confusing.

Examples

```
int x =  10;
int y = 010;       // not ok, Octal 10 is interpreted as Decimal 8
assert x+y == 20; // this therefore fails
```

Literature

references (CERT), DCL06-J: Beware of integer literals beginning with '0'.
references (JPL-C-STD), MISRA-C Rule 7.1 (Constants).
references (Puzzlers 2005), Rule 1.3 (Octal literals look like decimal literals).
references (PMD), Rule AvoidUsingOctalValues.

R39. do not use floating point equality

Expressions of type `float` and `double` should not be tested with `==` or `!=`.

Rationale

Floating point numbers often do not have an exact representation. Therefore, although numbers may be sufficiently close, they may not have the same representation.

references (JPL-C-STD), MISRA-C Rule 13.3 (Control statement expressions).
references (Puzzlers 2005), Rule 3.1 (Floating-point arithmetic is inexact).
references (FindBugs), Rule FE_FLOATING_POINT_EQUALITY.
references (PMD), Rule BadComparison.

R40. use one result type in conditional expressions

When using the `(p?e1:e2)` operator with numeric operands, one should use the same numeric type for both the second and third operands.

Rationale

The rules for determining the result type of a conditional expression are very complex and may result in surprises.

Examples

The following code snippet prints `X88` and not `XX` as one might have expected:

```
int i = 0;
System.out.print(true ? 'X' : 0); // not ok
System.out.print(true ? 'X' : i); // not ok
```

This is due to the algorithm used for determining the result type of the conditional expression. In this case, the first conditional expression has an integer constant as third argument, causing the result type to be that of the second operand, a `char`. The second conditional expression has an integer variable as third argument, causing the result type to be the largest type containing both `char` and `int`, which is `int`.

The following code snippet prints `XX` as expected:

```
  int i = 0;
  System.out.print(true ? 'X' : (char)0); // ok
  System.out.print(true ? 'X' : (char)i); // ok
```

Literature

references (CERT), EXP00-J: Use the same type for the second and third operands in conditional expressions.
references (Puzzlers 2005), Rule 4.1 (Mixed-type computations are confusing).

R41. do not use string concatenation operator in loops

The String concatenation operator + should not be used in a loop statement. Instead one should use a `java.lang.StringBuffer` or `java.lang.StringBuilder` for efficiency.

Rationale

The String concatenation operator + creates a new String object, which is costly. If performed in a loop with many iterations, a new String object is created for each + operator, for each loop iteration. This at first hand may just seem as a performance issue, but it can become a correctness issue in case the application response time is affected severely.

Examples

The following program contains two loops, each building a String object to be printed on standard output. The first loop correctly uses `java.lang.StringBuffer`, while the second loop incorrectly uses the String concatenation operator +. If one runs this program one will notice the remarkable difference in execution time of the two loops.

```
public class ConcatenationInWhile {
  static final int MAX = 40000;

  public static void main(String[] args) {
    // ok:

    StringBuffer message1 = new StringBuffer("");
    for (int i=0;i<MAX;i++) {
      message1.append("line ").append(i).append("\n");
    }
    System.out.println(message1);

    // not ok:

    String message2 = "";
    for (int i=0;i<MAX;i++) {
      message2 = message2 + "line " + i + "\n"; // uses + operator in loop
    }
    System.out.println(message2);
  }
}
```

Literature

references (Semmle), Rule: String Concatenation in Loop.

exceptions

R42. do not drop exceptions

Exceptions should not be dropped or ignored. Either exceptions should be caught and handled, or they should be thrown to the outermost level. Note that *checked exceptions* typically should be handled, whereas *unchecked exceptions* typically can be thrown to the outermost level.

Rationale

Dropping and ignoring an exception means that a user will not observe the occurrence of an error. This may result in errors not being detected when they appear the first time.

Note that it is dangerous to catch exceptions of type `Exception` since this includes exceptions of type `RuntimeException`, and such exceptions should normally flow to the outermost level since they indicate non-recoverable error.

Examples

The following example illustrates a typical way in which exceptions can be dropped, by having an empty `catch` block:

```
// not ok:

try {
  throw new BadIdeaException();
} catch (BadIdeaException e) {
  // let's ignore it
}
```

The rule is related to the rule: do not have empty blocks.

Literature

references (FindBugs) Rules DE_MIGHT_DROP, DE_MIGHT_IGNORE, and REC_CATCH_EXCEPTION.

R43. do not abruptly exit a finally block

A `finally` block should not contain a `return` statement and should not throw any exceptions that are not caught within the `finally` block.

Rationale

A `finally` block is entered when the `try` block finishes, reguardless of whether it finishes normally or abnormally by throwing an exception. The case of concern is where the `try` code throws an exception giving rise to two distinct situations where the exception is lost. First, if the `finally` block contains a return statement, then the original exception will be discarded. Second, if there is an uncaught exception thrown in the `finally` block, then the original exception will be discarded.

Examples

The following method returns `0.0` when applied to `0`. The division by `0` throws an exception, but it is ignored when the `return 0` statement in the `finally` construct is executed.

```
// not ok:

public double fraction(int x) {
  double value = 0.0;
  try {
    value = 1/x; // integer division by 0, throws exception
  } finally { // this is always executed
    if (x < 0.01 ) {
      return 0; // not ok, exception is now ignored
    }
  }
  return value;
}

// not ok:

public double fraction (int x) throws FixableProblem {
  double value = 0.0;
  try {
    value = 1/x; // integer division by 0, throws exception
  } finally { // this is always executed
    if (x < 0.01 ) {
      throws FixableProblem(); // not ok, original exception is now ignored
    }
  }
  return value;
}
```

Literature

references (Puzzlers 2005), Rule 5.3 (Abrupt completion of a finally block masks pending transfer of control).
references (PMD), Rule ReturnFromFinallyBlock and DoNotThrowExceptionInFinally.

types

R44. use generics

Java's generic type system should be used to its fullest extent. In other words, provide type arguments to generic interfaces and classes where possible.

Elaboration

Note that due to backwards compatibility Java allows *raw types* (interface and class instantiations without explicitly providing type arguments), although warnings are issued when used. New code should completely avoid raw types.

Rationale

This makes it possible for the compiler to detect when elements of the wrong type are added to the collection. It also means that explicit type casting can be avoided when taking elements out of the collection. This reduces the potential for casting errors. Finally, it functions as documentation. Before generics were introduced, good programming style was to write the element type as a comment.

Examples

```
List          elements1 = new ArrayList();          // not ok, yields a warning
List<Integer> elements2 = new ArrayList();          // not ok, yields a warning
List<Integer> elements3 = new ArrayList<Integer>(); // ok
```

Exceptions to the Rule

Generics were introduced in Java 1.5. Java legacy applications developed in earlier versions of Java do not use generics. That is, they do not provide explicit type parameters to interface and class instantiations. Even legacy code developed in Java 1.5 and Java 1.6 may not use generics due to lack of attention to the type checking benefits provided by this concept. When writing code that interacts with such legacy code, full type checking therefore cannot be enforced at compile time. When adding elements of a wrong type to a raw type collection for example, no error message is given, an error message may only result later when the collection is traversed, and only if it at that moment is expected that the collection elements have a particular type.

However, it is possible, for example, to transform a raw type list to a list which performs this check already when elements are added. This is done with the following `java.util.Collections` method:

```
public static <E> List<E> checkedList(List<E> list, Class<E> type)
```

This use of this method is illustrated by the following example where new code using a `List<Integer>` interacts with legacy code (the class `Legacy`) which only uses the raw style `List`. The example contains an incorrect interaction followed by a correct interaction.

```
class Legacy {
  public void add42(List list) {
    list.add(42);
  }
}

public class GenericType {
  private List<String> list = new ArrayList<String>();
  Legacy legacy = new Legacy();

  public void interactWithLegacy() {
    list.add("Ok"); // ok
    legacy.add42(list); // not ok and will not be caught here
    legacy.add42(Collections.checkedList(list, String.class)); // not ok, but will be caught here
    for (String s : list) {
      System.out.println(s); // if we get this far a ClassCastException will be thrown
    }
  }

  public static void main(String[] args) {
    new GenericType().interactWithLegacy();
  }
}
```

Literature

references (CERT), OBJ35-J: Use checked collections against external code.
references (GeoSoft), Rule 85.
references (Lea), Recommendations, Item 39.
references (Effective 2008), Item 23 (Don't use raw types in new code).
references (Puzzlers 2005), Rule 8.6 (Mixing raw and parameterized types weakens type checking).
references (FindBugs), Rule BC_UNCONFIRMED_CAST.

R45. use interfaces as types when available

One should avoid using implementation types (i.e., `HashSet`) when defining the type of a variable or parameter; use the interface (i.e, `Set`) instead in case it is defined.

Rationale

Loose coupling improves encapsulation. It makes the using software dependent on the definition (interface) and not the implementation (class) allowing for change of implementations without changing using software.

Example

```
HashSet<Integer> elements1 = new HashSet<Integer>(); // not ok
Set<Integer> elements2 = new HashSet<Integer>(); // ok
```

Literature

references (AmbySoft), Section 5.2 Documenting Parameters, Tip : Use Interfaces for Parameter Types.
references (PMD), Rule LooseCoupling.

R46. use primitive types

One should use primitive types (`byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`) in preference to boxed primitives (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`, `Character`) whenever there is a choice.

Rationale

The use of boxed primitives (objects) instead of primitive values generates three kinds of problems: (1) one cannot use == to compare values, although a programmer may make the mistake to do so, (2) the use of boxed primitives together with primitives in expressions may result in null pointer exceptions, and (3) repeated auto-boxing and auto-unboxing in a program may lead to unnecessary object creations.

Examples

The following definition fails to work due to the comparison: `first == second`. This comparison compares pointers, rather than the integers contained in the two `Integer` objects.

```
// this is not ok:

public int compare(Integer first, Integer second) {
  return first < second ? -1 : (first == second ? 0 : 1); // compares objects with ==
}
```

The use of `Integer` in this case is not necessary and `int` should be used instead.

The following code shows a case were the use of boxed primitives (in this case `Integer`) is necessary in order to check for `null`. This is needed due to the use of the generic `Map` type.

```
// this is ok:

public class AutoBoxing {
  public static void main(String[] args) {
    Map<String, Integer> m = new TreeMap<String, Integer>();
    for (String word : args) {
      Integer freq = m.get(word);                    // necessary use Integer
      m.put(word, (freq == null ? 1 : freq + 1)); // to allow check for null
    }
    System.out.println(m);
  }
}
```

Literature

references (Effective 2008) Item 49 (Prefer primitive types to boxed primitives).
references (Puzzlers 2005), Rule 4.4 (Operators == and `!=` perform reference comparisons on boxed primitive types).

R47. do not remove literals from collections

When removing a number (integral or floating point number) from a collection, one should be explicit about the type in order to ensure that auto-boxing converts to a boxed primitive object of the right type, and thereby causes the element to actually be removed. That is, ensure that the same boxed primitives are used when adding and when removing.

Rationale

Numeric auto-boxing, while removing elements from a collection, can make operations on the collection fail silently. Auto-boxing converts a value of primitive type to a corresponding boxed primitive. For example, auto-boxing will convert the `short` variable `s` to a `Short` object, and the integer `10` to an `Integer` object. The problem occurs when adding and removing numbers from a collection, where the numbers due to auto-boxing may be added as objects of one type and attempted removed as objects another type. In this case removal operations may fail to remove.

Examples

The following code defines a `HashSet` of `Short` objects. Adding a `short` variable causes it to auto-boxed to a `Short` object before being added. Removing a `10` (which is an `int` expression in Java) causes `10` to be auto-boxed to an `Integer` object, which is then attempted removed. Recall that the remove method has the type `boolean remove(Object o)`. This removal operation fails since there is no `Integer` object in the set, only a `Short` object. To get the `Short(10)` object removed, it is necessary with a call for example of the form `set.remove((short)10)` to ensure that the argument to the `HashSet.remove` method is auto-boxed to a `Short` object.

```
public class ShortSet {
  public static void main(String[] args) {
    HashSet<Short> set = new HashSet<Short>();
    short s = 10;
    set.add(s);                // adds Short(10)
    set.remove(10);            // not ok, attempts to remove Integer(10), not Short(10)
    System.out.println(set); // unexpectedly prints [10]
    set.remove((short)10);     // ok, now removes Short(10)
    System.out.println(set); // prints [] as expected
  }
}
```

Literature

references (CERT), EXP05-J: Be careful of auto-boxing when removing elements from a Collection.

R48. restrict numeric conversions

It should be verified (with whatever means, including code inspection) that unexpected values do not arise when performing numeric computations. Unexpected values result from conversions between different types resulting in loss of information or precision, via overflow and underflow, or application of operations outside their domain (for example division by zero).

Elaboration

More specifically, the following problems can arise:

- *information loss* : can occur when converting a number from a wider type to a narrower type. An example is a conversion from `int` to `byte`. Such conversions may cause loss of information. Java generally requires explicit casts to be inserted by the programmer from wider to narrower types, hence forcing the programmer to be aware of the issue. However, note that a compound assignment statement `v += e` is equivalent to `v = (T)(v + e)` where `T` is the type of `v`. This introduces an implicit cast, which could potentially cast from a wider type (of the right hand side) to a narrower type (T).

- *precision loss* : can occur when converting from `int` or `long` to `float`, or from `long` to `double`. Such conversions may cause loss of precision. When the expressions are not `strictfp`, conversions from `float` to `double` may lose precision about the overall magnitude of the converted value.

- *overflow/underflow* : occurs when operations compute values that are too big or too small to be represented in a word corresponding to the computation being performed. Overflow should generally be avoided by preferring `long` to `int`, and `double` to `float`. However, overflows/underflows may still happen. Consider as an example the computation:

  ```
  long l = Integer.MAX_VALUE + 1;
  ```

  `Integer.MAX_VALUE` (2147483647) is the biggest value representable in an `int` location. The expression `Integer.MAX_value+1` will be interpreted as an `int` computation (since the arguments are interpreted as `int`) and all intermediate results will therefore be stored in an `int` storage cell. Since this expression produces a value bigger than what is representable in an `int` cell, an overflow occurs, and the value -`2147483648` is stored in `l` instead of `2147483648`. This happens even though the result is stored in a `long` variable, wide enough to hold the result. In this case one of the arguments should explicitly be converted to the larger type, for example as in: `(long)Integer.MAX_VALUE + 1`, or `Integer.MAX_VALUE + 1L`. Generally, however, when using binary operators with mixed operand sizes, the narrower operands will be automatically promoted to a wider type, to match the type of the other operand. The Java Language Specification (JLS) section 5.6 "Numeric Promotions" describes numeric promotion as:

  - If any of the operands is of a reference type, unboxing conversion is performed. Then:
  - If either operand is of type double, the other is converted to double.
  - Otherwise, if either operand is of type float, the other is converted to float.
  - Otherwise, if either operand is of type long, the other is converted to long.
  - Otherwise, both operands are converted to type int.

- *truncation* : occurs for example during integer division (dividing an integer by an integer), in which case the remainder of the division is ignored. In order to avoid this, floating point division must be enforced by explicitly converting at least one of the arguments to a floating point number, for example by a cast.

- *pre-condition violations* : certain operators have pre-conditions that must be satisfied, such as the division operator that fails if the second operand is zero.

- *out of memory* : auto-boxing from a primitive numeric types to an object, for example from `int` to `Integer` may cause an `OutOfMemoryError` exception to be thrown if memory otherwise is limited at that moment.

- *null pointer exceptions* : integer operators can throw a `NullPointerException` if unboxing conversion of a null reference is required.

- *exceptional values* : Floating point expressions can evaluate to two special values in addition to normal floating point values: `infinite` (resulting from an overflow) and `NaN` (resulting from an erroneous computation). Care should be taken to ensure that these values are not computed unexpectedly. To check whether a variable is `infinite` or `NaN` one should use the following methods from `java.lang.Float` and `java.lang.Double` (and not compare with the == equality operator, which will not give the expected results):

```
class Float{
  boolean isInfinite();
  static boolean isInfinite(float v);
  boolean isNaN();
  static boolean isNaN(float v);
  ...
}

class Double {
  boolean isInfinite();
  static boolean isInfinite(double v);
  boolean isNaN();
  static boolean isNaN(double v);
  ...
}
```

- *platform dependence* : `strictfp` is a class/interface/method modifier used to restrict floating-point calculations to ensure portability. The modifier was introduced into the Java programming language with the Java virtual machine version 1.2. In older JVMs, floating-point calculations were always strict floating-point, meaning all values used during floating-point calculations are made in the IEEE-standard float or double sizes. This could sometimes result in a numeric overflow or underflow in the middle of a calculation, even if the end result would be a valid number. Since version 1.2 of the JVM, floating-point calculations do not require that all numbers used in computations are themselves limited to the standard float or double precision. However, for some applications, a programmer might require every platform to have precisely the same floating-point behavior, even if some platforms could handle more precision. In that case, the programmer can use the modifier `strictfp` to ensure that calculations are performed as in the earlier versions. That is, with `strictfp` results are portable, without it they are more likely to be accurate, causing less overflows/underflows.

Many of the above mentioned problems can be avoided by using the big number classes `BigInteger` and `BigDecimal`, which, however, are less efficient wrt. memory and time consumption.

Rationale

Unexpected and undesired errors can easily occur during numeric conversions and computations.

Examples

```
strictfp class NumericComputation {
  // strictfp modifier makes floating point computations follow IEEE standard

  public static void main(String[] args) {
    int i;
    long l;
```

```
        float f;

        // information loss:

        l = Long.MAX_VALUE;
        i = (int)l; // not ok
        System.out.println("info loss : " + l + " --> " + i);
        // PRINTS info loss : 9223372036854775807 --> -1

        // precision loss:

        i = Integer.MAX_VALUE;
        f = i; // perhaps ok, perhaps not, depends on precision wanted
        System.out.println("prec. loss : " + i + " --> " + f);
        // PRINTS prec. loss : 2147483647 --> 2.14748365E9

        // overflow:

        i = Integer.MAX_VALUE;
        l = i + 1;        // not ok: rhs evaluated as integer expression
        System.out.println("overflow :" + (i+1L) + " --> " + l);
        // PRINTS overflow :2147483648 --> -2147483648
        l = (long)i + 1; // ok : rhs converted before computation
        System.out.println("no overflow :" + (i+1L) + " --> " + l);
        //PRINTS no overflow :2147483648 --> 2147483648
        l = i + 1L;      // ok : rhs converted before computation

        // truncation:

        f = 1/2; // probably not ok : rhs computed as integer expression
        System.out.println("truncation :" + (1/2f) + " --> " +  f);
        // PRINTS truncation :0.5 --> 0.0
        f = 1/2f; // ok : rhs converted to float before computation

        // pre-condition violation, division by zero:

        f = 1/(1/2); // `1/2' is zero due to integer truncation.
        // throws ArithmeticException (/ by zero) exception

        // out of memory:

        Integer k = new Integer(0);
        k = k + 1; // ok, but can cause OutOfMemoryError due to auto-boxing of result

        // null pointer exceptions:

        k = null;
        k = k + 1; // not ok, throws a NullPointerException

        // exceptional values:

        f = Float.MAX_VALUE;
        System.out.println(f+1 == f+2); // prints `true' due to loss of precision
        f = f * f;
        System.out.println("that " + f + " is Infinity is " + Float.isInfinite(f));
        // PRINTS that Infinity is Infinity is true
        f = (float)Math.sqrt(-1);
        System.out.println("that " + f + " is NaN is " + Float.isNaN(f));
        // PRINTS that NaN is NaN is true
    }
}
```

Literature

references (CERT), EXP04-J: Be wary of invisible implicit casts when using compound assignment operators.
references (CERT), EXP08-J: Be aware of integer promotions in binary operators.
references (CERT), INT30-J: Range check before casting integers to narrower types.
references (CERT), INT33-J: Do not cast numeric types to wider floating-point types without range checking.
references (CERT), INT34-J: Perform explicit range checking to ensure integer operations do not overflow.
references (CERT), INT35-J: Do not attempt to store signed values in the char integral type.
references (CERT), FLP02-J: Do not attempt comparisons with NaN.
references (CERT), FLP03-J: Use the strictfp modifier for floating point calculation consistency.
references (CERT), FLP04-J: Check floating point inputs for exceptional values.
references (CERT), FLP31-J: Convert integers to floating point for floating point operations.
references (CERT), FLP32-J: Range check before casting floating point numbers to narrower types.
references (GeoSoft), Rules 42, 58 and 59.
references (Lea), Recommendations, Item 11.
references (JPL-C-STD), MISRA-C Rules 10.1, 10.2, 10.3, 10.4 (Arithmetic type conversions), and Rule 12.11 (Expressions).
references (Puzzlers 2005), Rule 2.2 (Integer arithmetic overflows silently).
references (Puzzlers 2005), Rule 2.4 (Compound assignment operators can cause silent narrowing cast).
references (Puzzlers 2005), Rule 3.1 (Floating-point arithmetic is inexact).
references (Puzzlers 2005), Rule 5.2 (It is difficult to terminate an int-indexed loop at Integer.MAX_VALUE).
references (Puzzlers 2005), Rule 10.6 (Values of type char are silently converted to int, not String).

references (FindBugs), Rules ICAST_IDIV_CAST_TO_DOUBLE and ICAST_INTEGER_MULTIPLY_CAST_TO_LONG.
references (PMD), Rule StringBufferInstantiationWithChar.

concurrency

As a general recommendation: use Java's `java.util.concurrent` package (new with Java 1.5) whenever appropriate. This package is meant to make programming with threads easier. This is recommended in the following literature:

references (Effective 2008), Item 68 (Prefer executors and tasks to threads) and Item 69 (Prefer concurrency utilities to wait and notify).
references (Puzzlers 2005), Introduction to Section 12 (Threads) in Appendix A: Avoid using low-level multithreaded programming where possible.
references (Concurrency 2006)

R49. program against data races

If a non-final field is to be accessed by two or more threads, then any code accessing the field should be protected in a `synchronized` block or method, or protected by locks using the `java.util.concurrent` package.

Rationale

Synchronization not only guarantees *mutual exclusion*, it also enforces *publication* of the current value of the data object to be made visible to other threads. Note that Java's memory model does not guarantee publication as a default. Even in the case where mutual exclusion is not needed (for example when writing to and reading from a field of primitive type occupying at most 32 bits, that is: different from `long` and `double` - both 64 bits), it is still necessary to *enforce* publication. Writes and reads of `volatile` fields are always atomic and always get published right away. Hence they provide an alternative to using synchronization. Note, however, that declaring a reference variable (pointing to an object) `volatile` only covers the reference, not the contents of the object pointed to. For example, declaring an array volatile and then updating an array element does not enforce the publication of that array element.

Examples

The following example illustrates access to a field `stop` by two threads `task1` and `task2`. Since the access is not synchronized, the assignment to the `stop` field in `requestStop()` called by `task2` cannot be guaranteed to be visible by `task1` due to the memory model.

```
// not ok since access to stop is not synchronized

class Task1 extends Thread {
  private boolean stop = false;

  public void requestStop() {
    stop = true;
  }

  boolean stopRequested() {
    return stop;
  }

  public void run() {
    int i = 0;
    while (!stopRequested()) {
      System.out.println(i);
      i++;
    }
  }
}

class Task2 extends Thread {
  Task1 task1;

  public Task2(Task1 task1) {
    this.task1 = task1;
  }

  public void run() {
    task1.requestStop();
  }
}

public class Synchronization {
  public static void main(String[] args) {
    Task1 task1 = new Task1();
    Task2 task2 = new Task2(task1);
    task1.start();
    task2.start();
  }
}
```

A solution is to make the two methods `requestStop` and `stopRequested` synchronized. In this particular case mutual exclusion is actually not necessary since `stop` is a `boolean`, and since it is only assigned to or written to. The effect of the synchronization is in this case only to ensure publication. In this case the alternative solution would be to declare the `stop` field volatile.

If the assignment had been a toggle of the form: `stop = !stop`, requiring a read of the variable and then a write-back, mutual exclusion would be needed, and the use of `volatile` would not work. In this case `synchronized` should be used.

Literature

references (CERT), CON03-J: Do not assume that elements of an array declared volatile are volatile.
references (JPL-C-STD), Rule 8 (access to shared data).
references (Effective 2008), Item 66 (Synchronize access to shared mutable data).
references (Puzzlers 2005), Rule 12.5 (Failure to synchronize when sharing mutable state can result in failure to observe state changes).
references (Elements 2001), Rule 97 (Avoid unnecessary synchronization ...) argues against synchronization for basic types (except long and double and references) for efficiency reasons, but ignores the delayed data transfer problem of the Java memory model mentioned above.
references (FindBugs), Rules IS_FIELD_NOT_GUARDED and VO_VOLATILE_REFERENCE_TO_ARRAY.

R50. program against deadlocks

One should protect against deadlocks caused by threads waiting for each other in a cyclic manner. More generally, one should avoid any one thread waiting for a lock indefinitely or even for an unnecessarily prolonged time period.

Elaboration

More specifically:

- locks should be taken in a predetermined, and documented, order.
- if two threads take the same lock, one thread should not wait for the other thread's termination while holding the lock.
- In the case where Java's built-in synchronization concept is replaced with explicit library lock and unlock methods (for example when using Java's `java.util.concurrent.locks.ReentrantLock`), unlock operations should always appear within the body of the same method that performs the matching lock operation.
- a thread should not call `Thread.sleep()` while holding a lock.
- a thread should not call `wait` while holding two or more locks.

Rationale

The classical example is that of a thread *T1* taking a lock *A* and then nested (while holding *A*) a lock *B*, while another thread *T2* takes a lock *B* and then nested (while holding *B*) the lock *A*. A deadlock for example occurs if thread *T1* takes *A* and immediately thereafter thread *T2* takes *B*. Now *T1* cannot get *B* and *T2* cannot get *A*. The rule that unlock operations should occur in the same method body as the matching lock operation makes it clearer that every lock operation is followed by an unlock operation. This reduces the risks for deadlocks.

Literature

references (CERT), CON00-J: Do not invoke a superclass method or constructor from a synchronized region in the subclass.
references (CERT), CON07-J: Do not defer a thread that is holding a lock.
references (JPL-C-STD), Rule 9 (semaphores and locking).
references (Effective 2008), Item 67 (Avoid excessive synchronization).
references (Puzzlers 2005), Rule 12.6 (Invoking alien method from within a synchronized block can cause deadlock).
references (FindBugs), Rules UL_UNRELEASED_LOCK, UL_UNRELEASED_LOCK_EXCEPTION_PATH, SWL_SLEEP_WITH_LOCK_HELD, and TLW_TWO_LOCK_WAIT.

R51. do not rely on the scheduler for synchronization

Thread synchronization and data protection should not be performed by relying on the scheduler by using thread delays, calling the `Thread.yield()` method, and using thread priorities.

Rationale

The use of a thread delay for thread synchronization and/or data protection requires a guess of how long certain actions will take. If the guess is wrong, bad computations can be the result. Calls of the `Thread.yield()` method and priorities are occasionally used by programmers in an attempt to control scheduling. However, neither of these concepts are portable. Their semantics are unspecified and they may work differently on different JVMs. Some JVMS even regard `Thread.yield()` as a no-op.

Examples

The following flawed code attempts to achieve an interaction between a producer and consumer, where their action should alternate according to the following regular expression: (produce consume)+. Note that the code is safe with respect to data races (obeys the rule for program against data races) and deadlocks (obeys the rule for program against deadlocks). The correct solution to this problem is to use the `notifyAll` and `wait` methods to ensure that the consumer sees every piece of data produced by the consumer exactly once.

```
// The following code is not ok

class SharedData {
  int x = 0;

  synchronized void set(int x) {
     this.x = x;
  }

  synchronized int get() {
    return x;
  }
}

class Producer extends Thread {
  private SharedData data;

  ...

  public void run() {
    try {
      while(true) {
        data.set(newData());
        sleep(2000);
      }
    } catch(InterruptedException e) { ... }
  }
}

class Consumer extends Thread {
  private SharedData data;

  ...

  public void run() {
    try {
      sleep(1000); // wait a bit to get in between producer activity
      while(true) {
        useData(data.get());
        sleep(2000);
      }
    } catch(InterruptedException e) { ... }
  }
}
```

Literature

references (JPL-C-STD), Rule 7 (thread safety).
references (Effective 2008), Item 72 (Don't depend on the thread scheduler).
references (Puzzlers 2005), Rule 12.8 (Depending on the thread scheduler may result in erratic and platform-dependent behavior).

R52. wait and notify safely

A `wait` statement should be in a while loop that re-waits if the condition being waited for does not hold. The while-loop itself should be within a synchronized block. One should never use the `notify()` method, instead one should use the `notifyAll()` method.

Rationale

When a thread is notified after having called `wait()`, the thread does not necessarily immediately get the lock. It is instead competing with other threads to acquire the lock. This has as consequence that another thread might acquire the lock, and violate the condition waited for. Therefore a repeated test on the condition is needed.

A call of `notify()` will only wake up one waiting thread. `notifyAll()` will wake up all waiting threads. This is regarded safer in the case where only a subset of the waiting threads may be ready to execute on the current object. Waking them all up increases the chances that one of them gets to run. Strictly speaking, if all threads wait for the same condition, and only one thread can benefit from continuing execution, `notify()` can be used to optimize the implementation. If they wait for different conditions `notifyAll()` should be used. However, errors can be introduced trying to determine whether more than one condition is waited for by different threads. It is therefore always safe to use `notifyAll()`.

Examples

The following example illustrates a correct use of a `while` loop around a `wait` statement.

```
// ok:

void put(Message message) {
  synchronized (this) {
    while (isFull()) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
  buffer.add(message);
}
```

The following example illustrates a one-place buffer offering a synchronized `int get()` method and a synchronized `put(int value)` method. The two methods wait for *different* conditions to become true (`full` respectively `!full`), hence their use of `notify()` instead of `notify()` is flawed.

```
class Buffer {
  private boolean full = false;
  private int value = 0;

  public synchronized int get() {
    while(!full) {
      try{
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    notify(); // not ok, should be notifyAll()
    this.full = false;
    return this.value;
  }

  public synchronized void put(int value) {
    while(full) {
      try{
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    notify(); // not ok, should be notifyAll()
    this.full = true;
    this.value = value;
  }
}
```

Consider for example a scenario with one producer and two consumers. A consumer may take out an element from the buffer and notify the other consumer instead of the producer.

Literature

references (Lea), Recommendations, Item 31.
references (Effective 2008), Item 69 (Prefer concurrency utilities to wait and notify).
references (Puzzlers 2005), Rule 12.7 (Invoking wait outside of a while loop causes unpredictable behavior).
references (Elements 2001), Rule 98 (Consider using notify() instead of notifyAll()) in fact recommends to use notify() for efficiency reasons if all waiting threads wait for the same condition.
references (FindBugs), Rules UW_UNCOND_WAIT and WA_NOT_IN_LOOP.
references (FindBugs), Rule NO_NOTIFY_NOT_NOTIFYALL.
references (PMD), Rule UseNotifyAllInsteadOfNotify.

complexity

R53. reduce code complexity

Code should be divided into small blocks with simple APIs for ease of use. Specifically:

- a class should contain no more than 10 fields
- a class should contain no more than 20 methods
- a method should contain no more than 75 lines of code
- a method should have no more than 7 parameters
- a method body should a cyclomatic complexity of no more than 10. More precisely, the cyclomatic complexity is the number of branching statements (`if`, `while`, `do`, `for`, `switch`, `case`, `catch`) plus the number of branching expressions (`?:`, `&&` and `||`) plus one. Methods with a high cyclomatic complexity (> 10) are hard to test and maintain, given their large number of possible execution paths. One may, however, have comprehensible control flow despite high numbers. For example, one large `switch` statement can be clear to understand, but can dramatically increase the count.
- an expression should contain no more than 5 operators.

Rationale

The above rules (1) limits on the size and complexity of classes and methods, (2) limits dependencies of one class on other classes, and (3) limits the complexity of an API (number of methods, number of parameters per method). It is generally acknowledged that oversized code blocks are hard to read and maintain. Putting numbers on what limits are reasonable is difficult. The numbers are meant to focus attention on code block size, rather than be perfect numbers.

Literature

references (AmbySoft), Section 2.4.5 Follow The Thirty-Second Rule.
references (GeoSoft), Rule 53.
references (JPL-C-STD), Rule 25.
references (SwartzComplexity)
references (Effective 2008), Item 40 (Design method signatures carefully).
references (Puzzlers 2005), Rule 4.3 (Operator precedence is not always obvious).
references (Elements 2001), Rule 69 (Define small classes and small methods).
references (Elements 2001), Rule 69 (Define small classes and small methods).
references (PMD), Rules ExcessiveImports and UnusedImports.
references (PMD), Rules ExcessiveClassLength, NcssTypeCount, TooManyFields, TooManyMethods, and ExcessivePublicCount.
references (PMD), Rules ExcessiveMethodLength, NcssConstructorCount and ExcessiveParameterList.
references (PMD), Rules CyclomaticComplexity, NPathComplexity, AvoidDeeplyNestedIfStmts, and SwitchDensity.

references

coding guidelines for Java

- (AmbySoft) AmbySoft's coding standard: http://www.ambysoft.com/essays/javaCodingStandards.html
- (CERT) CERT Secure Coding Standards: https://www.securecoding.cert.org
- (ESA) ESA Java Coding Standards: ftp://ftp.estec.esa.nl/pub/wm/wme/bssc/Java-Coding-Standards-20050303-releaseA.pdf
- (GeoSoft) GeoSoft's coding style: http://geosoft.no/development/javastyle.html
- (Lea) Doug Lea: Draft Java Coding Standard: http://g.oswego.edu/dl/html/javaCodingStd.html
- (Macadamiam) Macadamiam's coding standard: http://www.macadamian.com/insight/best_practices_detail/P1
- (SUN-code) SUN's code conventions: http://java.sun.com/docs/codeconv
- (SUN-doc) SUN's JavaDoc conventions: http://java.sun.com/j2se/javadoc/writingdoccomments/index.html
- (SwartzComplexity) Fred Swartz: Java Complexity Measurement: http://leepoint.net/notes-java/principles_and_practices/complexity/complexity_measurement.html

coding standards for other languages

- (JPL-C-STD) JPL Coding standard for C http://lars/LOC.html
- (MISRA-C) MISRA-C 2004 Guidelines for the use of the C language in critical systems http://lars/m2004.html

books on best practices

- (Puzzlers 2005) Java Puzzlers (Joshua Bloch and Neal Gafter); Addison-Wesley, 2005, (http://www.javapuzzlers.com). All references in this standard refer to Appendix A.
- (Effective 2008) Effective Java (Joshua Bloch); second edition, the Java series (SUN), Addison-Wesley, 2008, (http://java.sun.com/docs/books/effective/).
- (Concurrency 2006) Java Concurrency in Practice (Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea); Addison-Wesley, 2006, (http://www.javaconcurrencyinpractice.com).
- (Elements 2001) The elements of Java style (Allan Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur, and Patrick Thompson); Rogue Wave Software, Cambridge University Press, 2001, (http://www.ambysoft.com/books/elementsJavaStyle.html). This

book should include the material in the (AmbySoft) standard above.

static analyzers

- (CheckStyle) http://checkstyle.sourceforge.net
- (FindBugs) http://findbugs.sourceforge.net
- (PMD) http://pmd.sourceforge.net
- (Semmle) http://semmle.com

other Java tools

- (JavaDoc) The JavaDoc Tool: http://java.sun.com/j2se/javadoc.
- (JUnit) http://www.junit.org
- (JML) The Java Modeling Language: http://www.cs.ucf.edu/~leavens/JML.

Java

- (JLS) The Java Language Specification, Third Edition, James Gosling, Bill Joy, Guy Steele, Gilad Bracha, ADDISON-WESLEY, 2005: http://java.sun.com/docs/books/jls/.
- (JavaAPI) JavaTM 2 Platform Standard Edition 5.0 API Specification: http://java.sun.com/j2se/1.5.0/docs/api.

other programming languages

The following programming languages have been extended with assertion and annotation constructs:

- Eiffel : http://www.eiffel.com.
- SpeC# : http://research.microsoft.com/en-us/projects/specsharp.

Children (1) Hide Children | View in Hierarchy | Add Child Page

Rejected Rules

3 Comments | Add Comment