JPL Java Coding Standard

JPL Institutional Coding Standard for the Java Programming Language

Version 1.0

Published March 31, 2014

JPL Java Coding Standard

Acknowledgements

This standard is based on the JPL Java Coding Standard developed by Klaus Havelund and Al Niessner. It was developed as a collaboration between Jet Propulsion Laboratory (JPL) and Semmle Limited. The following additional people at JPL have contributed to the standard via their comments: Eddie Benowitz, Dj Byrne, Bradley Clement, Thomas Crockett, Bob Deen, Marti DeMore, Dan Dvorak, Gerard Holzmann, Thomas Huang, Mark Indictor, Rajeev Joshi, Ara Kassabian, Cin-Young Lee, Ken Peters and David Wagner.

Introduction

This document presents 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 because of the richer resources available on ground software computers, and the often less time critical nature of ground applications. However, note that JPL ground software can indeed 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) the Java standard 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 standard is a collaboration between the Laboratory for Reliable Software (LaRS) at the Jet Propulsion Laboratory (JPL) and Semmle Limited. Semmle develops and sells a static analyzer that analyzes Java code and checks for adherence to the rules in this standard.

Terminology

Throughout this document, we use the following terms to discuss software quality:

- **Rule** – The coding standard consists of a set of rules. Each rule describes a coding convention that should be avoided or adhered to, to help avoid coding mistakes, avoid bad programming practice, or otherwise improve the quality of the software project. Rules are grouped by category (see table).
- **Violation** – Code that breaks a rule.
- **Defect –** A problem with the program, from coding mistakes through to user-reported problems concerning the behavior of the program.

Rules

The rules are grouped into three high-level categories:

- **Critical** - these rules must always be followed and violations of these rules must be corrected as soon as possible.
- **Important** - these rules should be followed and violations of these rules should be corrected where

practical.

- **Advisory** - these rules represent good practice. Violations of these rules are allowed but not recommended.

Critical

Rules in this category **must always** be followed and violations of these rules must be corrected as soon as possible after you identify them.

Rule types:

- Arithmetic
- Concurrency
- Declarations
- Encapsulation
- Equality
- Exceptions
- Expressions
- Extensibility
- Incomplete Code
- Java objects
- Logic Errors
- Naming
- Random
- Resource Leaks
- Strings
- Types

Arithmetic

- Avoid casting the result of integer multiplication to type 'long'
- Avoid implicit narrowing in compound assignment
- Avoid type mismatch in conditional expressions
- Avoid using octal literals
- Do not test floating point equality

Avoid casting the result of integer multiplication to type 'long'

Category: Critical > Arithmetic

Description: Casting the result of an integer multiplication to type 'long' instead of casting before the multiplication may cause overflow.

An integer multiplication that is assigned to a variable of type `long` or returned from a method with return type `long` may cause unexpected arithmetic overflow.

Recommendation

Casting to type `long` before multiplying reduces the risk of arithmetic overflow.

Example

In the following example, the multiplication expression assigned to `j` causes overflow and results in the value `-1651507200` instead of `4000000000000000000`.

```
1 int i = 2000000000;
2 long j = i*i; // causes overflow
```

In the following example, the assignment to `k` correctly avoids overflow by casting one of the operands to type `long`.

```
1 int i = 2000000000;
2 long k = i*(long)i; // avoids overflow
```

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 3. Addison-Wesley, 2005.
- The Java Language Specification: Multiplication Operator.
- Common Weakness Enumeration: CWE-190: Integer Overflow or Wraparound.
- The CERT Oracle Secure Coding Standard for Java: NUM00-J. Detect or prevent integer overflow.

Avoid implicit narrowing in compound assignment

Category: Critical > Arithmetic

Description: Compound assignment statements (for example 'intvar += longvar') that implicitly cast a value of a wider type to a narrower type may result in information loss and numeric errors such as overflows.

Compound assignment statements of the form `x += y` or `x *= y` perform an implicit narrowing conversion if the type of `x` is narrower than the type of `y`. For example, `x += y` is equivalent to `x = (T)(x + y)`, where `T` is the type of `x`. This can result in information loss and numeric errors such as overflows.

Recommendation

Ensure that the type of the left-hand side of the compound assignment statement is at least as wide as the type of the right-hand side.

Example

If `x` is of type `short` and `y` is of type `int`, the expression `x + y` is of type `int`. However, the expression `x += y` is equivalent to `x = (short) (x + y)`. The expression `x + y` is cast to the type of the left-hand side of the assignment: `short`, possibly leading to information loss.

To avoid implicitly narrowing the type of `x + y`, change the type of `x` to `int`. Then the types of `x` and `x + y` are both `int` and there is no need for an implicit cast.

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 9. Addison-Wesley, 2005.
- The Java Language Specification: Compound Assignment Operators, Narrowing Primitive Conversion.
- Common Weakness Enumeration: CWE-190: Integer Overflow or Wraparound.
- The CERT Oracle Secure Coding Standard for Java: NUM00-J. Detect or prevent integer overflow.

Avoid type mismatch in conditional expressions

Category: Critical > Arithmetic

Description: Using the '(p?e1:e2)' operator with different primitive types for the second and third operands may cause unexpected results.

Conditional expressions of the form `(p ? e1 : e2)` can yield unexpected results if `e1` and `e2` have distinct primitive types.

Example

The following example illustrates the most confusing case, which occurs when one branch has type `char` and the other branch does not have type `char`.

```
1 int i = 0;
2 System.out.print(true ? 'x' : 0); // prints "x"
3 System.out.print(true ? 'x' : i); // prints "120"
```

This unexpected result is due to binary numeric promotion of `'x'` from `char` to `int`. For details on the result type of the conditional operator, see the references.

Recommendation

When using the ternary conditional operator with numeric operands, the second and third operand should have the same numeric type. This avoids potentially unexpected results caused by binary numeric promotion.

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 8. Addison-Wesley, 2005.
- The Java Language Specification: Conditional Operator ?.

Avoid using octal literals

Category: Critical > Arithmetic

Description: An integer literal that starts with '0' may cause a problem. If the '0' is intentional, a programmer may misread the literal as a decimal literal. If the '0' is unintentional and a decimal literal is intended, the compiler treats the literal as an octal literal.

An integer literal consisting of a leading `0` digit followed by one or more digits in the range `0-7` is an octal literal. This can lead to two problems:

- An octal literal can be misread by a programmer as a decimal literal.
- A programmer might accidentally start a decimal literal with a zero, so that the compiler treats the decimal literal as an octal literal. For example, `010` is equal to `8`, not `10`.

Recommendation

To avoid these problems:

- Avoid using octal literals so that programmers do not confuse them with decimal literals. However, if you need to use octal literals, you should add a comment to each octal literal indicating the intention to use octal literals.
- When typing decimal literals, be careful not to begin them with a zero accidentally.

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 59. Addison-Wesley, 2005.
- The Java Language Specification: Integer Literals.

Do not test floating point equality

Category: Critical > Arithmetic

Description: Equality tests on floating point values may lead to unexpected results.

Equality tests on floating point values may lead to unexpected results because of arithmetic imprecision. For example, the expression `23.42f==23.42` evaluates to `false`.

Recommendation

Instead of testing for *exact equality* between floating point values, check that the difference between the values is within an appropriate error margin.

Alternatively, if you do not want any inaccuracy when testing for equality, use one of the following instead of floating point values:

- `BigDecimal` class. This can store decimal values with higher precision.
- `long` type. Because this is an integer type, you must convert any decimal values to whole values. For example, represent $1.43 as 143 cents.

Example

In the following example, `(0.1 + 0.2) == 0.3` evaluates to `false`, even though you would expect it to evaluate to `true`. This is because of the imprecision of floating point data types.

```
1 class NoComparisonOnFloats
2 {
3     public static void main(String[] args)
4     {
5         System.out.println((0.1 + 0.2) == 0.3);
6     }
7 }
```

In the following improved example, the test for equality is performed by calculating the difference between the two values, and checking if the difference is within the error margin, `EPSILON`.

```
1 class NoComparisonOnFloats
2 {
3     public static void main(String[] args)
4     {
5         final double EPSILON = 0.001;
6         System.out.println(Math.abs((0.1 + 0.2) - 0.3) < EPSILON);
7     }
8 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 48. Addison-Wesley, 2008.
- Numerical Computation Guide: (What Every Computer Scientist Should Know About Floating-Point Arithmetic).

Concurrency

- API Misuse
- Synchronization
- Thread Safety
- Waiting

API Misuse

- Avoid ineffective thread definitions
- Avoid setting thread priorities
- Avoid using 'notify'
- Do not call 'Thread.yield'
- Do not spin on field
- Do not start a thread in a constructor

Avoid ineffective thread definitions

Category: Critical > Concurrency > API Misuse

Description: Thread instances that neither get an argument of type 'Runnable' passed to their constructor nor override the 'Thread.run' method are likely to have no effect.

New threads can be defined using one of the following alternatives:

- By extending the `Thread` class and overriding its `run` method.
- By passing an argument of type `Runnable` to the constructor of the `Thread` class.

Thread instances that are defined using another approach are likely to have no effect.

Recommendation

To avoid empty thread instances, define new threads using one of the following alternatives:

- By extending the `Thread` class and overriding its `run` method.
- By passing an argument of type `Runnable` to the constructor of the `Thread` class.

Example

In the following example, class `Bad` shows the definition of a thread that has no effect.

```
1  class Bad{
2
3    public void runInThread(){
4      Thread thread = new Thread();
5      thread.start();
6    }
7
8  }
```

In the following example, class `GoodWithOverride` shows how to extend the `Thread` class and override its `run` method, and class `GoodWithRunnable` shows how to pass an argument of type `Runnable` to the constructor of the `Thread` class.

$body

```
1  class GoodWithOverride{
2
3    public void runInThread(){
4      Thread thread = new Thread(){
5        @Override
6        public void run(){
7          System.out.println("Doing something");
8        }
9      };
10      thread.start;
11    }
12
13 }
14
15 class GoodWithRunnable{
16
17   public void runInThread(){
18     Runnable thingToRun = new Runnable(){
19       @Override
20       public void run(){
21         System.out.println("Doing something");
22       }
23     };
24
25     Thread thread = new Thread(thingToRun());
```

```
26      thread.start();
27    }
28
29  }
```

References

- Java API Documentation: Thread.
- The Java Tutorials: Defining and Starting a Thread.

Avoid setting thread priorities

Category: Critical > Concurrency > API Misuse

Description: Setting thread priorities to control interactions between threads is not portable, and may not have the desired effect.

Specifying thread priorities using calls to `Thread.setPriority` and `Thread.getPriority` is not portable and may have adverse consequences such as starvation.

Recommendation

Avoid setting thread priorities to control interactions between threads. Using the default thread priority should be sufficient for most applications.

However, if you need to enforce a specific synchronization order, use one of the following alternatives:

- Waiting for a notification using the `wait` and `notifyAll` methods
- Using the `java.util.concurrent` library

In some cases, calls to `Thread.sleep` may be appropriate to temporarily stop execution (provided that there is no possibility for race conditions), but this is not generally recommended.

References

- J. Bloch, *Effective Java (second edition)*, Item 72. Addison-Wesley, 2008.
- Inform IT: Adding Multithreading Capability to Your Java Applications.

Avoid using 'notify'

Category: Critical > Concurrency > API Misuse

Description: Calling 'notify' instead of 'notifyAll' may fail to wake up the correct thread and cannot wake up multiple threads.

Calls to the `notify` method rather than `notifyAll` may fail to wake up the correct thread if an object's monitor (intrinsic lock) is used for multiple conditions. `notify` only wakes up a single arbitrary thread that is waiting on the object's monitor, whereas `notifyAll` wakes up all such threads.

Recommendation

Ensure that the call to `notify` instead of `notifyAll` is a correct and desirable optimization. If not, call `notifyAll` instead.

Example

In the following example, the methods `produce` and `consume` both use `notify` to tell any waiting threads that an object has been added or removed from the buffer. However, this means that only *one* thread is notified. The woken-up thread might not be able to proceed due to its condition being false, immediately going back to the waiting state. As a result no progress is made.

```
1  class ProducerConsumer {
2      private static final int MAX_SIZE=3;
3      private List<Object> buf = new ArrayList<Object>();
4
5      public synchronized void produce(Object o) {
6          while (buf.size()==MAX_SIZE) {
7              try {
8                  wait();
9              }
10              catch (InterruptedException e) {
11                  ...
12              }
13          }
14          buf.add(o);
15          notify(); // 'notify' is used
16      }
17
18      public synchronized Object consume() {
19
20          while (buf.size()==0) {
21              try {
22                  wait();
23              }
24              catch (InterruptedException e) {
25                  ...
26              }
27          }
28          Object o = buf.remove(0);
29          notify(); // 'notify' is used
30          return o;
31      }
32  }
```

When using `notifyAll` instead of `notify`, *all* threads are notified, and if there are any threads that could proceed, we can be sure that at least one of them will do so.

References

- J. Bloch. *Effective Java (second edition)*, p. 277. Addison-Wesley, 2008.
- Java API Documentation: notify(), notifyAll().

Do not call 'Thread.yield'

Category: Critical > Concurrency > API Misuse

Description: Calling 'Thread.yield' may have no effect, and is not a reliable way to prevent a thread from taking up too much execution time.

The method `Thread.yield` is a non-portable and underspecified operation. It may have no effect, and is not a reliable way to prevent a thread from taking up too much execution time.

Recommendation

Use alternative ways of preventing a thread from taking up too much execution time. Communication between threads should normally be implemented using some form of waiting for a notification using the `wait` and `notifyAll` methods or by using the `java.util.concurrent` library.

In some cases, calls to `Thread.sleep` may be appropriate to temporarily cease execution (provided there is no possibility for race conditions), but this is not generally recommended.

References

- J. Bloch, *Effective Java (second edition)*, Item 72. Addison-Wesley, 2008.
- Java API Documentation: Thread.yield(), Object.wait(), Object.notifyAll(), java.util.concurrent, Thread.sleep().

Do not spin on field

Category: Critical > Concurrency > API Misuse

Description: Repeatedly reading a non-volatile field within the condition of an empty loop may result in an infinite loop.

Repeatedly reading a non-volatile field within the condition of an empty loop statement may result in an infinite loop, since a compiler optimization may move this field access out of the loop.

Example

In the following example, the method `spin` repeatedly tests the field `done` in a loop. The method repeats the while-loop until the value of the field `done` is set by another thread. However, the compiler could optimize the code as shown in the second code snippet, because the field `done` is not marked as `volatile` and there are no statements in the body of the loop that could change the value of `done`. The optimized version of `spin` loops forever, even when another thread would set `done` to `true`.

```
1  class Spin {
2      public boolean done = false;
3
4      public void spin() {
5          while(!done){
6          }
7      }
8  }
9
10 class Spin { // optimized
11     public boolean done = false;
12
13     public void spin() {
14         boolean cond = done;
15         while(!cond){
16         }
17     }
18 }
```

Recommendation

Ensure that access to this field is properly synchronized. Alternatively, avoid spinning on the field and instead use the `wait` and `notifyAll` methods or the `java.util.concurrent` library to communicate between threads.

References

- The Java Language Specification: Threads and Locks.

Do not start a thread in a constructor

Category: Critical > Concurrency > API Misuse

Description: Starting a thread within a constructor may cause the thread to start before any subclass constructor has completed its initialization, causing unexpected results.

Starting a thread within a constructor may cause unexpected results. If the class is extended, the thread may start before the subclass constructor has completed its initialization, which may not be intended.

Recommendation

Avoid starting threads in constructors. Typically, the constructor of a class only *constructs* the thread object, and a separate `start` method should be provided to *start* the thread object created by the constructor.

Example

In the following example, because the `Test` constructor implicitly calls the `Super` constructor, the thread created in the `Super` constructor may start before `this.name` has been initialized. Therefore, the program may output "hello " followed by a null string.

```
1  class Super {
2      public Super() {
3          new Thread() {
4              public void run() {
5                  System.out.println(Super.this.toString());
6              }
7          }.start(); // BAD: The thread is started in the constructor of 'Super'.
8      }
9
10      public String toString() {
11          return "hello";
12      }
13 }
14
15 class Test extends Super {
16      private String name;
17      public Test(String nm) {
18          // The thread is started before
19          // this line is run
20          this.name = nm;
21      }
22
23      public String toString() {
24          return super.toString() + " " + name;
25      }
26
27      public static void main(String[] args) {
28          new Test("my friend");
29      }
30 }
```

In the following modified example, the thread created in the `Super` constructor is not started within the constructor; `main` starts the thread after `this.name` has been initialized. This results in the program outputting "hello my friend".

```
1  class Super {
2      Thread thread;
3      public Super() {
4          thread = new Thread() {
5              public void run() {
6                  System.out.println(Super.this.toString());
```

```
7             }
8         };
9     }
10
11     public void start() {  // good
12         thread.start();
13     }
14
15     public String toString() {
16         return "hello";
17     }
18 }
19
20 class Test extends Super {
21     private String name;
22     public Test(String nm) {
23         this.name = nm;
24     }
25
26     public String toString() {
27         return super.toString() + " " + name;
28     }
29
30     public static void main(String[] args) {
31         Test t = new Test("my friend");
32         t.start();
33     }
34 }
```

References

- IBM developerWorks: Don't start threads from within constructors.

Synchronization

- Avoid data races by accessing shared variables under synchronization
- Avoid empty synchronized blocks
- Avoid inconsistent synchronization for 'writeObject'
- Avoid inconsistent synchronization of overriding methods
- Avoid synchronizing 'set' but not 'get'
- Do not synchronize on a field and update it

Avoid data races by accessing shared variables under synchronization

Category: Critical > Concurrency > Synchronization

Description: If a field is mostly accessed in a synchronized context, but occasionally accessed in a non-synchronized way, the non-synchronized accesses may lead to race conditions.

If a field is mostly accessed in a synchronized context, but occasionally accessed in a non-synchronized way, the non-synchronized accesses may lead to race conditions.

Recommendation

Ensure that the non-synchronized field accesses are made synchronized, if required.

Example

In the following example, `counter` is accessed in a synchronized way in *all but one* cases. If `modifyCounter` is called by a large number of threads that are running concurrently, the value of `counter` at the end of each call may not be zero. This is because the non-synchronized statement could be interleaved with updates to the counter that are performed by the other threads.

```
1  class MultiThreadCounter {
2      public int counter = 0;
3
4      public void modifyCounter() {
5          synchronized(this) {
6              counter--;
7          }
8          synchronized(this) {
9              counter--;
10          }
11          synchronized(this) {
12              counter--;
13          }
14          counter = counter + 3;  // No synchronization
15      }
16 }
```

To correct this, the last statement of `modifyCounter` should be enclosed in a `synchronized` statement.

References

- The Java Language Specification: Synchronization.

Avoid empty synchronized blocks

Category: Critical > Concurrency > Synchronization

Description: Empty synchronized blocks may indicate the presence of incomplete code or incorrect synchronization, and may lead to concurrency problems.

Empty synchronized blocks suspend execution until a lock can be acquired, which is then released immediately. This is unlikely to achieve the desired effect and may indicate the presence of incomplete code or incorrect synchronization. It may also lead to concurrency problems.

Recommendation

Check which code needs to be synchronized. Any code that requires synchronization on the given lock should be placed within the synchronized block.

References

- The Java Language Specification: The synchronized Statement.
- The Java Tutorials: Synchronization.

Avoid inconsistent synchronization for 'writeObject'

Category: Critical > Concurrency > Synchronization

Description: Classes with a synchronized 'writeObject' method but no other synchronized methods usually lack a sufficient level of synchronization.

Classes with a synchronized `writeObject` method but no other synchronized methods usually lack a sufficient level of synchronization. If any mutable state of this class can be modified without proper synchronization, the serialization using the `writeObject` method may result in an inconsistent state.

Recommendation

See if synchronization is necessary on methods other than `writeOject` to make the class thread-safe. Any methods that access or modify the state of an object of this class should usually be synchronized as well.

References

- The Java Language Specification: Synchronization.

Avoid inconsistent synchronization of overriding methods

Category: Critical > Concurrency > Synchronization

Description: If a synchronized method is overridden in a subclass, and the overriding method is not synchronized, the thread-safety of the subclass may be broken.

If a synchronized method is overridden in a subclass, the compiler does not require the overriding method to be synchronized. However, if the overriding method is not synchronized, the thread-safety of the subclass may be broken.

Recommendation

Ensure that the overriding method is synchronized, if necessary.

References

- The Java Language Specification: Synchronization.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Avoid synchronizing 'set' but not 'get'

Category: Critical > Concurrency > Synchronization

Description: If a class has a synchronized 'set' method, and a similarly-named 'get' method is not also synchronized, calls to the 'get' method may not return a consistent state for the object.

If a class has a synchronized `set` method and a similarly-named `get` method is not also synchronized, calls to the `get` method may not return a consistent state for the object.

Recommendation

Synchronize read operations as well as write operations. You should usually synchronize the `get` method.

References

- The Java Language Specification: Synchronization.

Do not synchronize on a field and update it

Category: Critical > Concurrency > Synchronization

Description: Synchronizing on a field and updating that field while the lock is held is unlikely to provide the desired thread safety.

A block of code that synchronizes on a field and updates that field while the lock is held is unlikely to provide the desired thread safety. Such a synchronized block does not prevent multiple unsynchronized assignments to that field because it obtains a lock on the object stored *in* the field rather than the field itself.

Recommendation

Instead of synchronizing on the field itself, consider synchronizing on a separate lock object when you want to avoid simultaneous updates to the field. You can do this by declaring a synchronized method and using it for any field updates.

Example

In the following example, in class A, synchronization takes place on the field that is updated in the body of the `setField` method.

```
1  public class A {
2      private Object field;
3
4      public void setField(Object o){
5          synchronized (field){    // BAD: synchronize on the field to be updated
6              field = o;
7              // ... more code ...
8          }
9      }
10 }
```

In class B, the recommended approach is shown, where synchronization takes place on a separate lock object.

```
1  public class B {
2     private final Object lock = new Object();
3     private Object field;
4
5     public void setField(Object o){
6         synchronized (lock){      // GOOD: synchronize on a separate lock object
7             field = o;
8             // ... more code ...
9         }
10    }
11 }
```

References

- The Java Language Specification: The synchronized Statement, synchronized Methods.
- The Java Tutorials: Lock Objects.

Thread Safety

- Avoid lazy initialization of a static field
- Avoid static fields of type 'DateFormat' (or its descendants)
- Ensure that a method releases locks on exit

Avoid lazy initialization of a static field

Category: Critical > Concurrency > Thread Safety

Description: Initializing a static field without synchronization can be problematic in a multi-threaded context.

The tactic of initializing a static field the first time it is used, known as "lazy initialization", can be problematic in a multi-threaded context when used without proper synchronization. If a separate thread starts executing before the field is initialized, the thread may see an incompletely initialized object.

Recommendation

If lazy initialization is desirable for performance reasons, the best solution is usually to declare the enclosing method `synchronized`. Otherwise, avoid lazy initialization and initialize static fields using static initializers. A third possibility is to declare the field `volatile` and use the double-checked locking idiom as explained in the article referenced below. As the article points out, it is crucial to declare the field `volatile`: double-checked locking by itself is *not* correct under the Java memory model.

Example

In the following example, the static field `resource` is initialized without synchronization.

```
1 class Singleton {
2     private static Resource resource;
3 
4     public Resource getResource() {
5         if(resource == null)
6             resource = new Resource();  // Lazily initialize "resource"
7         return resource;
8     }
9 }
```

In the following modification of the above example, `Singleton` uses the recommended approach of using a static initializer to initialize `resource`.

```
1 class Singleton {
2     private static Resource resource;
3 
4     static {
5         resource = new Resource();  // Initialize "resource" only once
6     }
7 
8     public Resource getResource() {
9         return resource;
10     }
11 }
```

References

- University of Maryland Department of Computer Science: The "Double-Checked Locking is Broken" Declaration.

Avoid static fields of type 'DateFormat' (or its descendants)

Category: Critical > Concurrency > Thread Safety

Description: Static fields of type 'DateFormat' (or its descendants) should be avoided because the class 'DateFormat' is not thread-safe.

Static fields of type `java.text.DateFormat` or its descendants should be avoided because the class `DateFormat` is not thread-safe.

Recommendation

Use instance fields instead and synchronize access where necessary.

Example

In the following example, `DateFormattingThread` declares a static field `dateF` of type `DateFormat`. When instances of `DateFormattingThread` are created and run by `DateFormatThreadUnsafe`, erroneous results are output because `dateF` is shared by all instances of `DateFormattingThread`.

```
1  class DateFormattingThread implements Runnable {
2      private static DateFormat dateF = new SimpleDateFormat("yyyyMMdd");  // Static field declared
3
4      public void run() {
5          for(int i=0; i < 10; i++){
6              try {
7                  Date d = dateF.parse("20121221");
8                  System.out.println(d);
9              } catch (ParseException e) { }
10         }
11     }
12 }
13
14 public class DateFormatThreadUnsafe {
15
16     public static void main(String[] args) {
17         for(int i=0; i<100; i++){
18             new Thread(new DateFormattingThread()).start();
19         }
20     }
21
22 }
```

In the following modification of the above example, `DateFormattingThread` declares an *instance* field `dateF` of type `DateFormat`. When instances of `DateFormattingThread` are created and run by `DateFormatThreadUnsafeFix`, correct results are output because there is a separate instance of `dateF` for each instance of `DateFormattingThread`.

```
1  class DateFormattingThread implements Runnable {
2      private DateFormat dateF = new SimpleDateFormat("yyyyMMdd");  // Instance field declared
3
4      public void run() {
5          for(int i=0; i < 10; i++){
6              try {
7                  Date d = dateF.parse("20121221");
8                  System.out.println(d);
9              } catch (ParseException e) { }
10         }
11     }
12 }
13
14 public class DateFormatThreadUnsafeFix {
```

```
15
16      public static void main(String[] args) {
17          for(int i=0; i<100; i++){
18              new Thread(new DateFormattingThread()).start();
19          }
20      }
21
22  }
```

References

- Java API Documentation: java.text.DateFormat synchronization.

Ensure that a method releases locks on exit

Category: Critical > Concurrency > Thread Safety

Description: Methods that acquire a lock without releasing it on method exit may cause deadlock.

If a method acquires a lock and some of the exit paths from the method do not release the lock then this may cause deadlock.

Recommendation

Ensure that all exit paths of the method release the lock.

Example

In the following example, `LockingThread.run` acquires a lock but releases it only some of the time, dependent on the result of a random number generator. This means that, of the 10 threads that are started by `UnreleasedLock.main`, only the first few are likely to finish running. The first thread to acquire the lock but not release it prevents the next thread from completing execution.

```
1  class LockingThread implements Runnable {
2      private static ReentrantLock l = new ReentrantLock();
3
4      public void run() {
5          l.lock();  // Acquire lock
6          System.out.println("Got lock");
7          if(new Random().nextInt(2) == 0){
8              l.unlock();  // Release lock only some of the time
9          }
10      }
11 }
```

To avoid this problem, `LockingThread.run` should release the lock (using `l.unlock();`) each time that it is run.

References

- Java API Documentation: java.util.concurrent.Lock.

Waiting

- Avoid calling 'Object.wait' while two locks are held
- Avoid calling 'Thread.sleep' with a lock held
- Avoid calling 'wait' on a 'Condition' interface
- Avoid controlling thread interaction by using ineffective or wasteful methods
- Do not call 'wait' outside a loop

Avoid calling 'Object.wait' while two locks are held

Category: Critical > Concurrency > Waiting

Description: Calling 'Object.wait' while two locks are held may cause deadlock.

Calling `Object.wait` while two locks are held may cause deadlock, because only one lock is released by `wait`.

Recommendation

See if one of the locks should continue to be held while waiting for a condition on the other lock. If not, release one of the locks before calling `Object.wait`.

Example

In the following example of the problem, `printText` locks both `idLock` and `textLock` before it reads the value of `text`. It then calls `textLock.wait`, which releases the lock on `textLock`. However, `setText` needs to lock `idLock` but it cannot because `idLock` is still locked by `printText`. Thus, deadlock is caused.

```
1  class WaitWithTwoLocks {
2
3      private final Object idLock = new Object();
4      private int id = 0;
5
6      private final Object textLock = new Object();
7      private String text = null;
8
9      public void printText() {
10         synchronized (idLock) {
11             synchronized (textLock) {
12                 while(text == null)
13                     try {
14                         textLock.wait();  // The lock on "textLock" is released but not the
15                                           // lock on "idLock".
16                     }
17                     catch (InterruptedException e) { ... }
18                 System.out.println(id + ":" + text);
19                 text = null;
20                 textLock.notifyAll();
21             }
22         }
23     }
24
25     public void setText(String mesg) {
26         synchronized (idLock) { // "setText" needs a lock on "idLock" but "printText" already
27                                 // holds a lock on "idLock", leading to deadlock
28             synchronized (textLock) {
29                 id++;
30                 text = mesg;
31                 idLock.notifyAll();
32                 textLock.notifyAll();
33             }
34         }
35     }
36 }
```

In the following modification of the above example, `id` and `text` are included in the class `Message`. The method `printText` synchronizes on the field `message` before it reads the value of `message.text`. It then calls `message.wait`, which releases the lock on `message`. This enables `setText` to lock `message` so that it can proceed.

```
1  class WaitWithTwoLocksGood {
```

```
2
3     private static class Message {
4         public int id = 0;
5         public String text = null;
6     }
7
8     private final Message message = new Message();
9
10     public void printText() {
11         synchronized (message) {
12             while(message.txt == null)
13                 try {
14                     message.wait();
15                 }
16                 catch (InterruptedException e) { ... }
17             System.out.println(message.id + ":" + message.text);
18             message.text = null;
19             message.notifyAll();
20         }
21     }
22
23     public void setText(String mesg) {
24         synchronized (message) {
25             message.id++;
26             message.text = mesg;
27             message.notifyAll();
28         }
29     }
30 }
```

References

- Java API Documentation: Object.wait().

Avoid calling 'Thread.sleep' with a lock held

Category: Critical > Concurrency > Waiting

Description: Calling 'Thread.sleep' with a lock held may lead to very poor performance or even deadlock.

Calling `Thread.sleep` with a lock held may lead to very poor performance or even deadlock. This is because `Thread.sleep` does not cause a thread to release its locks.

Recommendation

`Thread.sleep` should be called only outside of a `synchronized` block. However, a better way for threads to yield execution time to other threads may be to use either of the following solutions:

- The `java.util.concurrent` library
- The `wait` and `notifyAll` methods

Example

In the following example of the problem, two threads, `StorageThread` and `OtherThread`, are started. Both threads output a message to show that they have started but then `StorageThread` locks `counter` and goes to sleep. The lock prevents `OtherThread` from locking `counter`, so it has to wait until `StorageThread` has woken up and unlocked `counter` before it can continue.

```
1  class StorageThread implements Runnable{
2      public static Integer counter = 0;
3      private static final Object LOCK = new Object();
4
5      public void run() {
6          System.out.println("StorageThread started.");
7          synchronized(LOCK) {  // "LOCK" is locked just before the thread goes to sleep
8              try {
9                  Thread.sleep(5000);
10              } catch (InterruptedException e) { ... }
11         }
12         System.out.println("StorageThread exited.");
13     }
14 }
15
16 class OtherThread implements Runnable{
17     public void run() {
18         System.out.println("OtherThread started.");
19         synchronized(StorageThread.LOCK) {
20             StorageThread.counter++;
21         }
22         System.out.println("OtherThread exited.");
23     }
24 }
25
26 public class SleepWithLock {
27     public static void main(String[] args) {
28         new Thread(new StorageThread()).start();
29         new Thread(new OtherThread()).start();
30     }
31 }
```

To avoid this problem, `StorageThread` should call `Thread.sleep` outside the `synchronized` block instead, so that `counter` is unlocked.

References

- Java API Documentation: Thread.sleep(), Object.wait(), Object.notifyAll(), java.util.concurrent.

Avoid calling 'wait' on a 'Condition' interface

Category: Critical > Concurrency > Waiting

Description: Calling 'wait' on a 'Condition' interface may result in unexpected behavior and is probably a typographical error.

Calling `wait` on an object of type `java.util.concurrent.locks.Condition` may result in unexpected behavior because `wait` is a method of the `Object` class, not the `Condition` interface itself. Such a call is probably a typographical error: typing "wait" instead of "await".

Recommendation

Instead of `Object.wait`, use one of the `Condition.await` methods.

References

- Java API Documentation: java.util.concurrent.Condition.

Avoid controlling thread interaction by using ineffective or wasteful methods

Category: Critical > Concurrency > Waiting

Description: Calling 'Thread.sleep' to control thread interaction is less effective than waiting for a notification and may also result in race conditions. Merely synchronizing over shared variables in a loop to control thread interaction may waste system resources and cause performance problems.

Trying to control thread interaction by periodically calling `Thread.sleep` within a loop while waiting for a condition to be satisfied is less effective than waiting for a notification. This is because the waiting thread may either sleep for an unnecessarily long time or wake up too frequently. This approach may also result in race conditions and, therefore, incorrect code.

Trying to control thread interaction by repeatedly checking a synchronized data structure without calling `Thread.sleep` or waiting for a notification may waste a lot of system resources and cause noticeable performance problems.

Recommendation

See if communication between threads can be improved by using either of the following solutions:

- The `java.util.concurrent` library, preferably
- The `Object.wait` and `Object.notifyAll` methods

If following one of these recommendations is not feasible, ensure that race conditions cannot occur and precise timing is not required for program correctness.

Example

In the following example, the `Receiver` thread sleeps for an unnecessarily long time (up to five seconds) until it has received the message.

```
1  class Message {
2      public String text = "";
3  }
4
5  class Receiver implements Runnable {
6      private Message message;
7      public Receiver(Message msg) {
8          this.message = msg;
9      }
10      public void run() {
11          while(message.text.isEmpty()) {
12              try {
13                  Thread.sleep(5000);  // Sleep while waiting for condition to be satisfied
14              } catch (InterruptedException e) { }
15          }
16          System.out.println("Message Received at " + (System.currentTimeMillis()/1000));
17          System.out.println(message.text);
18      }
19 }
20
21 class Sender implements Runnable {
22      private Message message;
23      public Sender(Message msg) {
24          this.message = msg;
25      }
26      public void run() {
27          System.out.println("Message sent at " + (System.currentTimeMillis()/1000));
28          message.text = "Hello World";
```

```
29      }
30  }
31
32  public class BusyWait {
33      public static void main(String[] args) {
34          Message msg = new Message();
35          new Thread(new Receiver(msg)).start();
36          new Thread(new Sender(msg)).start();
37      }
38  }
```

In the following modification of the above example, the `Receiver` thread uses the recommended approach of waiting for a notification that the message has been sent. This means that the thread can respond immediately instead of sleeping.

```
1   class Message {
2       public String text = "";
3   }
4
5   class Receiver implements Runnable {
6       private Message message;
7       public Receiver(Message msg) {
8           this.message = msg;
9       }
10       public void run() {
11          synchronized(message) {
12              while(message.text.isEmpty()) {
13                  try {
14                      message.wait();  // Wait for a notification
15                  } catch (InterruptedException e) { }
16              }
17          }
18          System.out.println("Message Received at " + (System.currentTimeMillis()/1000));
19          System.out.println(message.text);
20      }
21  }
22
23  class Sender implements Runnable {
24      private Message message;
25      public Sender(Message msg) {
26          this.message = msg;
27      }
28      public void run() {
29          System.out.println("Message sent at " + (System.currentTimeMillis()/1000));
30          synchronized(message) {
31              message.text = "Hello World";
32              message.notifyAll();  // Send notification
33          }
34      }
35  }
36
37  public class BusyWait {
38      public static void main(String[] args) {
39          Message msg = new Message();
40          new Thread(new Receiver(msg)).start();
41          new Thread(new Sender(msg)).start();
42      }
43  }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 72. Addison-Wesley, 2008.
- Java API Documentation: Object.wait(), Object.notifyAll(), java.util.concurrent.
- The Java Tutorials: Guarded Blocks, High Level Concurrency Objects.

Do not call 'wait' outside a loop

Category: Critical > Concurrency > Waiting

Description: Calling 'wait' outside a loop may result in the program continuing before the expected condition is met.

Calling `Object.wait` outside of a loop may cause problems because the thread does not go back to sleep after a spurious wake-up call. This results in the program continuing before the expected condition is met.

Recommendation

Ensure that `wait` is called within a loop that tests for the condition that the thread is waiting for. This ensures that the program only proceeds to execute when the relevant condition is true. Note that the thread that calls `wait` on an object must be the owner of that object's monitor.

Example

In the following example, `obj.wait` is called within a `while` loop until the condition is true, at which point the program continues with the next statement after the loop:

```
1 synchronized (obj) {
2     while (<condition is false>) obj.wait();
3     // condition is true, perform appropriate action ...
4 }
```

References

- J. Bloch, *Effective Java (second edition)*, p. 276. Addison-Wesley, 2008.
- Java API Documentation: Object.wait().
- The Java Tutorials: Guarded Blocks.

Declarations

- Avoid ambiguity when calling a method that is in both a superclass and an outer class
- Avoid confusing non-override of package-private method
- Avoid hiding a field in a super class
- Include 'break' in a 'case' statement

Avoid ambiguity when calling a method that is in both a superclass and an outer class

Category: Critical > Declarations

Description: An unqualified call to a method that exists with the same signature in both a superclass and an outer class is ambiguous.

If a call is made to a method from an inner class A, and a method of that name is defined in both a superclass of A and an outer class of A, it is not clear to a programmer which method is intended to be called.

Example

In the following example, it is not clear whether the call to `printMessage` calls the method that is defined in `Outer` or `Super`.

```
1  public class Outer
2  {
3      void printMessage() {
4          System.out.println("Outer");
5      }
6
7      class Inner extends Super
8      {
9          void ambiguous() {
10             printMessage();  // Ambiguous call
11         }
12     }
13
14     public static void main(String[] args) {
15         new Outer().new Inner().ambiguous();
16     }
17 }
18
19 class Super
20 {
21     void printMessage() {
22         System.out.println("Super");
23     }
24 }
```

Inherited methods take precedence over methods in outer classes, so the method in the superclass is called. However, such situations are a potential cause of confusion and defects.

Recommendation

Resolve the ambiguity by explicitly qualifying the method call:

- To specify the outer class, prefix the method with `Outer.this.`.
- To specify the superclass, prefix the method with `super.`.

In the above example, the call to `printMessage` could be replaced by either `Outer.this.printMessage` or `super.printMessage`, depending on which method you intend to call. To preserve the behavior in the example, use `super.printMessage`.

References

- Inner Classes Specification: What are top-level classes and inner classes?.

Avoid confusing non-override of package-private method

Category: Critical > Declarations

Description: A method that appears to override another method but does not, because the declaring classes are in different packages, is potentially confusing.

If a method is declared with default access (that is, not private, protected, nor public), it can only be overridden by methods in the same package. If a method of the same signature is defined in a subclass in a different package, it is a completely separate method and no overriding occurs.

Code like this can be confusing for other programmers, who have to understand that there is no overriding relation, check that the original programmer did not intend one method to override the other, and avoid mixing up the two methods by accident.

Recommendation

In cases where there is intentionally no overriding, the best solution is to rename one or both of the methods to clarify their different purposes.

If one method is supposed to override another method that is declared with default access in another package, the access of the method must be changed to `public` or `protected`. Alternatively, the classes must be moved to the same package.

Example

In the following example, `PhotoResizerWidget.width` does not override `Widget.width` because one method is in package `gui` and one method is in package `gui.extras`.

```
1  // File 1
2  package gui;
3
4  abstract class Widget
5  {
6      // ...
7
8      // Return the width (in pixels) of this widget
9      int width() {
10         // ...
11     }
12
13     // ...
14 }
15
16 // File 2
17 package gui.extras;
18
19 class PhotoResizerWidget extends Widget
20 {
21     // ...
22
23     // Return the new width (of the photo when resized)
24     public int width() {
25         // ...
26     }
27
28     // ...
29 }
```

Assuming that no overriding is intentional, one or both of the methods should be renamed. For example, `PhotoResizerWidget.width` would be better named `PhotoResizerWidget.newPhotoWidth`.

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Language Specification: 8.4.8.1 Overriding (by Instance Methods).

Avoid hiding a field in a super class

Category: Critical > Declarations

Description: Hiding a field in a superclass by redeclaring it in a subclass might be unintentional, especially if references to the hidden field are not qualified using 'super'.

A field that has the same name as a field in a superclass *hides* the field in the superclass. Such hiding might be unintentional, especially if there are no references to the hidden field using the `super` qualifier. In any case, it makes code more difficult to read.

Recommendation

Ensure that any hiding is intentional. For clarity, it may be better to rename the field in the subclass.

Example

In the following example, the programmer unintentionally added an `age` field to `Employee`, which hides the `age` field in `Person`. The constructor in `Person` sets the `age` field in `Person` to 20 but the `age` field in `Employee` is still 0. This means that the program outputs 0, which is probably not what was intended.

```
1  public class FieldMasksSuperField {
2      static class Person {
3          protected int age;
4          public Person(int age)
5          {
6              this.age = age;
7          }
8      }
9
10      static class Employee extends Person {
11          protected int age;  // This field hides 'Person.age'.
12          protected int numberOfYearsEmployed;
13          public Employee(int age, int numberOfYearsEmployed)
14          {
15              super(age);
16              this.numberOfYearsEmployed = numberOfYearsEmployed;
17          }
18      }
19
20      public static void main(String[] args) {
21          Employee e = new Employee(20, 2);
22          System.out.println(e.age);
23      }
24  }
```

To fix this, delete the declaration of `age` on line 11.

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- The Java Tutorials: Hiding Fields.

Include 'break' in a 'case' statement

Category: Critical > Declarations

Description: A 'case' statement that does not contain a 'break' statement allows execution to 'fall through' to the next 'case', which may not be intended.

In a `switch` statement, execution 'falls through' from one `case` to the next, unless the `case` ends with a `break` statement. A common programming error is to forget to insert a `break` at the end of a `case`.

Recommendation

End each `case` with a `break` statement or, if execution is supposed to fall through to the next `case`, comment the last line of the `case` with the following comment: `/* falls through */`

Such comments are not required for a completely empty `case` that is supposed to share the same implementation with the subsequent `case`.

Example

In the following example, the `PING` case is missing a `break` statement. As a result, after `reply` is assigned the value of `Message.PONG`, execution falls through to the `TIMEOUT` case. Then the value of `reply` is erroneously assigned the value of `Message.PING`. To fix this, insert `break;` at the end of the `PING` case.

```
1  class Server
2  {
3      public void respond(Event event)
4      {
5          Message reply = null;
6          switch (event) {
7          case PING:
8              reply = Message.PONG;
9              // Missing 'break' statement
10         case TIMEOUT:
11             reply = Message.PING;
12         case PONG:
13             // No reply needed
14         }
15         if (reply != null)
16             send(reply);
17     }
18
19     private void send(Message message) {
20         // ...
21     }
22 }
23
24 enum Event { PING, PONG, TIMEOUT }
25 enum Message { PING, PONG }
```

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 23. Addison-Wesley, 2005.
- Code Conventions for the Java Programming Language: 7.8 switch Statements.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Encapsulation

- Avoid casting from an abstract collection to a concrete implementation type
- Avoid declaring array constants
- Avoid defining an interface (or abstract class) only to hold constants

Avoid casting from an abstract collection to a concrete implementation type

Category: Critical > Encapsulation

Description: A cast from an abstract collection to a concrete implementation type makes the code brittle.

Most collections in the Java standard library are defined by an abstract interface (for example `java.util.List` or `java.util.Set`), which is implemented by a range of concrete classes and a range of wrappers. Normally, except when constructing an object, it is better to use the abstract types because this avoids assumptions about what the implementation is.

A cast from an abstract to a concrete collection makes the code brittle by ensuring it works only for one possible implementation class and not others. Usually, such casts are either an indication of over-reliance on concrete implementation types, or of the fact that the wrong abstract type was used.

Recommendation

It is usually best to use the abstract type consistently in variable, field and parameter declarations.

There may be individual exceptions. For example, it is common to declare variables as `LinkedHashSet` rather than `Set` when the iteration order matters and only the `LinkedHashSet` implementation provides the right behavior.

Example

The following example illustrates a situation where the wrong abstract type is used. The `List` interface does not provide a `poll` method, so the original code casts `queue` down to the concrete type `LinkedList`, which does. To avoid this downcasting, simply use the correct abstract type for this method, namely `Queue`. This documents the intent of the programmer and allows for various implementations of queues to be used by clients of this method.

```
1  Customer getNext(List<Customer> queue) {
2      if (queue == null)
3          return null;
4      LinkedList<Customer> myQueue = (LinkedList<Customer>)queue;  // AVOID: Cast to concrete type.
5      return myQueue.poll();
6  }
7
8  Customer getNext(Queue<Customer> queue) {
9      if (queue == null)
10         return null;
11      return queue.poll();  // GOOD: Use abstract type.
12 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 52. Addison-Wesley, 2008.
- Java 6 API Specification: Collection.

Avoid declaring array constants

Category: Critical > Encapsulation

Description: Array constants are mutable and can be changed by malicious code or by accident.

Constant values are typically represented by public, static, final fields. When defining several related constants, it is sometimes tempting to define a public, static, final field with an array type, and initialize it with a list of all the different constant values.

However, the `final` keyword applies only to the field itself (that is, the array reference), and not to the contents of the array. This means that the field always refers to the same array instance, but each element of the array may be modified freely. This possibly invalidates important assumptions of client code.

Recommendation

Where possible, avoid declaring array constants. If there are only a few constant values, consider using a named constant for each one, or defining them in an `enum` type.

If you genuinely need to refer to a long list of constants with the same name and an index, consider replacing the array constant with a constant of type `List` to which you assign an unmodifiable collection. See the example for ways of achieving this.

Example

In the following example, `public static final` applies only to `RGB` itself, not the constants that it contains.

```
1  public class Display {
2      // AVOID: Array constant is vulnerable to mutation.
3      public static final String[] RGB = {
4          "FF0000", "00FF00", "0000FF"
5      };
6
7      void f() {
8          // Re-assigning the "constant" is legal.
9          RGB[0] = "00FFFF";
10      }
11 }
```

The following example shows examples of ways to declare constants that avoid this problem.

```
1  // Solution 1: Extract to individual constants
2  public class Display {
3      public static final String RED = "FF0000";
4      public static final String GREEN = "00FF00";
5      public static final String BLUE = "0000FF";
6  }
7
8  // Solution 2: Define constants using in an enum type
9  public enum Display
10  {
11      RED ("FF0000"), GREEN ("00FF00"), BLUE ("0000FF");
12
13      private String rgb;
14      private Display(int rgb) {
15          this.rgb = rgb;
16      }
17      public String getRGB(){
18          return rgb;
19      }
```

```
}

// Solution 3: Use an unmodifiable collection
public class Display {
    public static final List<String> RGB =
            Collections.unmodifiableList(
                    Arrays.asList("FF0000",
                            "00FF00",
                            "0000FF"));
}

// Solution 4: Use a utility method
public class Utils {
    public static <T> List<T> constList(T... values) {
        return Collections.unmodifiableList(
                Arrays.asList(values));
    }
}

public class Display {
    public static final List<String> RGB =
            Utils.constList("FF0000", "00FF00", "0000FF");
}
```

References

- J. Bloch, *Effective Java (second edition)*, p. 70. Addison-Wesley, 2008.
- Java Language Specification: 4.12.4 final Variables.

Avoid defining an interface (or abstract class) only to hold constants

Category: Critical > Encapsulation

Description: Defining an interface (or abstract class) only to hold a number of constant definitions is bad practice.

Definitions of constants (meaning static, final fields) should be placed in an appropriate class where they belong logically. It is usually bad practice to define an interface (or abstract class) only to hold a number of constant definitions.

This often arises when a developer tries to put the constant definitions into scope by just implementing the interface (or extending the abstract class) that defines them.

Recommendation

The preferred way of putting the constant definitions into scope is to use the `import static` directive, which allows a compilation unit to put any visible static members from other classes into scope.

This issue is discussed in [Bloch]:

> *That a class uses some constants internally is an implementation detail. Implementing a constant interface causes this implementation detail to leak into the classes exported API. It is of no consequence to the users of a class that the class implements a constant interface. In fact, it may even confuse them. Worse, it represents a commitment: if in a future release the class is modified so that it no longer needs to use the constants, it still must implement the interface to ensure binary compatibility.*

To prevent this pollution of a class's binary interface, it is best to move the constant definitions to whatever concrete class uses them most frequently. Users of the definitions could use `import static` to access the relevant fields.

Example

In the following example, the interface `MathConstants` has been defined only to hold a constant.

```
1  public class NoConstantsOnly {
2      static interface MathConstants
3      {
4          public static final Double Pi = 3.14;
5      }
6
7      static class Circle implements MathConstants
8      {
9          public double radius;
10          public double area()
11          {
12              return Math.pow(radius, 2) * Pi;
13          }
14      }
15 }
```

Instead, the constant should be moved to the `Circle` class or another class that uses the constant frequently.

References

- J. Bloch, *Effective Java (second edition)*, Item 19. Addison-Wesley, 2008.

Equality

- Avoid comparing arrays using 'Object.equals'
- Avoid comparing object identity of boxed types
- Avoid comparing object identity of strings
- Avoid hashed instances that do not define 'hashCode'
- Avoid overriding 'compareTo' but not 'equals'
- Avoid overriding only one of 'equals' and 'hashCode'
- Avoid possible inconsistency due to 'instanceof' in 'equals'
- Avoid reference comparisons with operands of type 'Object'
- Avoid unintentionally overloading 'Object.equals'
- Do not make calls of the form 'x.equals(y)' with incomparable types
- Ensure that an implementaton of 'equals' inspects its argument type

Avoid comparing arrays using 'Object.equals'

Category: Critical > Equality

Description: Comparing arrays using the 'Object.equals' method checks only reference equality, which is unlikely to be what is intended.

Code that compares arrays using the `Object.equals` method checks only reference equality. This is unlikely to be what is intended.

Recommendation

To compare the lengths of the arrays and the corresponding pairs of elements in the arrays, use one of the comparison methods from `java.util.Arrays`:

- The method `Arrays.equals` performs a shallow comparison. That is, array elements are compared using `equals`.
- The method `Arrays.deepEquals` performs a deep comparison, which is appropriate for comparisons of nested arrays.

Example

In the following example, the two arrays are first compared using the `Object.equals` method. Because this checks only reference equality and the two arrays are different objects, `Object.equals` returns `false`. The two arrays are then compared using the `Arrays.equals` method. Because this compares the length and contents of the arrays, `Arrays.equals` returns `true`.

```
1  public void arrayExample(){
2      String[] array1 = new String[]{"a", "b", "c"};
3      String[] array2 = new String[]{"a", "b", "c"};
4
5      // Reference equality tested: prints 'false'
6      System.out.println(array1.equals(array2));
7
8      // Equality of array elements tested: prints 'true'
9      System.out.println(Arrays.equals(array1, array2));
10 }
```

References

- Java API Documentation: Arrays.equals(), Arrays.deepEquals(), Object.equals().

Avoid comparing object identity of boxed types

Category: Critical > Equality

Description: Comparing two boxed primitive values using the == or != operator compares object identity, which may not be intended.

Comparing two boxed primitive values using `==` or `!=` compares object identity, which may not be intended.

Recommendation

Usually, you should compare non-primitive objects, for example boxed primitive values, by using their `equals` methods.

Example

With the following definition, the method call `refEq(new Integer(2), new Integer(2))` returns `false` because the objects are not identical.

```
1 boolean refEq(Integer i, Integer j) {
2     return i == j;
3 }
```

With the following definition, the method call `realEq(new Integer(2), new Integer(2))` returns `true` because the objects contain equal values.

```
1 boolean realEq(Integer i, Integer j) {
2     return i.equals(j);
3 }
```

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 32. Addison-Wesley, 2005.
- Java API Documentation: Object.equals(), Integer.equals().

Avoid comparing object identity of strings

Category: Critical > Equality

Description: Comparing two strings using the == or != operator compares object identity, which may not be intended.

Comparing two `String` objects using `==` or `!=` compares object identity, which may not be intended. The same sequence of characters can be represented by two distinct `String` objects.

Recommendation

To see if two `String` objects represent the same sequence of characters, you should usually compare the objects by using their `equals` methods.

Example

With the following definition, the method call `refEq("Hello World", new String("Hello World"))` returns `false` because the objects are not identical.

```
1 boolean refEq(String s1, String s2) {
2     return s1 == s2;
3 }
```

With the following definition, the method call `realEq("Hello World", new String("Hello World"))` returns `true` because the objects contain equal values.

```
1 boolean realEq(String s1, String s2) {
2     return s1.equals(s2);
3 }
```

References

- Java API Documentation: String.equals(), String.intern().
- The Java Language Specification: 15.21.3, 3.10.5, 15.28.

Avoid hashed instances that do not define 'hashCode'

Category: Critical > Equality

Description: Classes that define an 'equals' method but no 'hashCode' method, and whose instances are stored in a hashing data structure, can lead to unexpected results.

Classes that define an `equals` method but no `hashCode` method can lead to unexpected results if instances of those classes are stored in a hashing data structure. Hashing data structures expect that hash codes fulfill the contract that two objects that `equals` considers equal should have the same hash code. This contract is likely to be violated by such classes.

Recommendation

Every class that implements a custom `equals` method should also provide an implementation of `hashCode`.

Example

In the following example, class `Point` has no implementation of `hashCode`. Calling `hashCode` on two distinct `Point` objects with the same coordinates would probably result in different hash codes. This would violate the contract of the `hashCode` method, in which case objects of type `Point` should not be stored in hashing data structures.

```
1  class Point {
2      int x;
3      int y;
4
5      Point(int x, int y) {
6          this.x = x;
7          this.y = y;
8      }
9
10      public boolean equals(Object o) {
11          if (!(o instanceof Point)) return false;
12          Point q = (Point)o;
13          return x == q.x && y == q.y;
14      }
15 }
```

In the modification of the above example, the implementation of `hashCode` for class `Point` is suitable because the hash code is computed from exactly the same fields that are considered in the `equals` method. Therefore, the contract of the `hashCode` method is fulfilled.

```
1  class Point {
2      int x;
3      int y;
4
5      Point(int x, int y) {
6          this.x = x;
7          this.y = y;
8      }
9
10      public boolean equals(Object o) {
11          if (!(o instanceof Point)) return false;
12          Point q = (Point)o;
13          return x == q.x && y == q.y;
14      }
15
16      // Implement hashCode so that equivalent points (with the same values of x and y) have the
17      // same hash code
```

```
18      public int hashCode() {
19          int hash = 7;
20          hash = 31*hash + x;
21          hash = 31*hash + y;
22          return hash;
23      }
24  }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 9. Addison-Wesley, 2008.
- Java API Documentation: Object.equals, Object.hashCode.
- IBM developerWorks: Java theory and practice: Hashing it out.

Avoid overriding 'compareTo' but not 'equals'

Category: Critical > Equality

Description: If a class overrides 'compareTo' but not 'equals', it may mean that 'compareTo' and 'equals' are inconsistent.

A class that overrides `compareTo` but not `equals` may not implement a natural ordering that is consistent with `equals`.

Recommendation

Although this consistency is not strictly required by the `compareTo` contract, usually both methods should be overridden to ensure that they are consistent, that is, that `x.compareTo(y)==0` is `true` if and only if `x.equals(y)` is `true`, for any non-null `x` and `y`.

Example

In the following example, the class `InconsistentCompareTo` overrides `compareTo` but not `equals`.

```
1  public class InconsistentCompareTo implements Comparable<InconsistentCompareTo> {
2      private int i = 0;
3      public InconsistentCompareTo(int i) {
4          this.i = i;
5      }
6
7      public int compareTo(InconsistentCompareTo rhs) {
8          return i - rhs.i;
9      }
10 }
```

In the following example, the class `InconsistentCompareToFix` overrides both `compareTo` and `equals`.

```
1  public class InconsistentCompareToFix implements Comparable<InconsistentCompareToFix> {
2      private int i = 0;
3      public InconsistentCompareToFix(int i) {
4          this.i = i;
5      }
6
7      public int compareTo(InconsistentCompareToFix rhs) {
8          return i - rhs.i;
9      }
10
11      public boolean equals(InconsistentCompareToFix rhs) {
12          return i == rhs.i;
13      }
14 }
```

If you require a natural ordering that is inconsistent with `equals`, you should document it clearly.

References

- J. Bloch, *Effective Java (second edition)*, Item 12. Addison-Wesley, 2008.
- Java API Documentation: Comparable.compareTo, Comparable, Object.equals.

Avoid overriding only one of 'equals' and 'hashCode'

Category: Critical > Equality

Description: If a class overrides only one of 'equals' and 'hashCode', it may mean that 'equals' and 'hashCode' are inconsistent.

A class that overrides only one of `equals` and `hashCode` is likely to violate the contract of the `hashCode` method. The contract requires that `hashCode` gives the same integer result for any two equal objects. Not enforcing this property may cause unexpected results when storing and retrieving objects of such a class in a hashing data structure.

Recommendation

Usually, both methods should be overridden to ensure that they are consistent.

Example

In the following example, the class `InconsistentEqualsHashCode` overrides `hashCode` but not `equals`.

```
1  public class InconsistentEqualsHashCode {
2      private int i = 0;
3      public InconsistentEqualsHashCode(int i) {
4          this.i = i;
5      }
6
7      public int hashCode() {
8          return i;
9      }
10 }
```

In the following example, the class `InconsistentEqualsHashCodeFix` overrides both `hashCode` and `equals`.

```
1  public class InconsistentEqualsHashCodeFix {
2      private int i = 0;
3      public InconsistentEqualsHashCodeFix(int i) {
4          this.i = i;
5      }
6
7      @Override
8      public int hashCode() {
9          return i;
10      }
11
12      @Override
13      public boolean equals(Object obj) {
14          if (obj == null)
15              return false;
16          if (getClass() != obj.getClass())
17              return false;
18          InconsistentEqualsHashCodeFix that = (InconsistentEqualsHashCodeFix) obj;
19          return this.i == that.i;
20      }
21 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 9. Addison-Wesley, 2008.
- Java API Documentation: Object.equals, Object.hashCode.
- IBM developerWorks: Java theory and practice: Hashing it out.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Avoid possible inconsistency due to 'instanceof' in 'equals'

Category: Critical > Equality

Description: Implementations of 'equals' that use 'instanceof' to test the type of the argument and are further overridden in a subclass are likely to violate the 'equals' contract.

Implementations of `equals` that use `instanceof` to check the type of their argument are likely to lead to non-symmetric definitions of `equals`, if they are further overridden in subclasses that add fields and redefine `equals`. A definition of the `equals` method should be reflexive, symmetric, and transitive, and a violation of the `equals` contract may lead to unexpected behavior.

Recommendation

Consider using one of the following options:

- Check the type of the argument using `getClass` instead of `instanceof`.
- Declare the class or the `equals` method `final`. This prevents the creation of subclasses that would otherwise violate the `equals` contract.
- Replace inheritance by composition. Instead of a class `B` extending a class `A`, class `B` can declare a field of type `A` in addition to any other fields.

The first option has the disadvantage of violating the substitution principle of object-oriented languages, which says that an instance of a subclass of `A` can be provided whenever an instance of class `A` is required.

Example

The first option is illustrated in the following example:

```
1  class BadPoint {
2      int x;
3      int y;
4
5      BadPoint(int x, int y) {
6          this.x = x;
7          this.y = y;
8      }
9
10      public boolean equals(Object o) {
11          if(!(o instanceof BadPoint))
12              return false;
13          BadPoint q = (BadPoint)o;
14          return x == q.x && y == q.y;
15      }
16 }
17
18 class BadPointExt extends BadPoint {
19      String s;
20
21      BadPointExt(int x, int y, String s) {
22          super(x, y);
23          this.s = s;
24      }
25
26      // violates symmetry of equals contract
27      public boolean equals(Object o) {
28          if(!(o instanceof BadPointExt)) return false;
29          BadPointExt q = (BadPointExt)o;
30          return super.equals(o) && (q.s==null ? s==null : q.s.equals(s));
```

```
31      }
32  }
33
34  class GoodPoint {
35      int x;
36      int y;
37
38      GoodPoint(int x, int y) {
39          this.x = x;
40          this.y = y;
41      }
42
43      public boolean equals(Object o) {
44          if (o != null && getClass() == o.getClass()) {
45              GoodPoint q = (GoodPoint)o;
46              return x == q.x && y == q.y;
47          }
48          return false;
49      }
50  }
51
52  class GoodPointExt extends GoodPoint {
53      String s;
54
55      GoodPointExt(int x, int y, String s) {
56          super(x, y);
57          this.s = s;
58      }
59
60      public boolean equals(Object o) {
61          if (o != null && getClass() == o.getClass()) {
62              GoodPointExt q = (GoodPointExt)o;
63              return super.equals(o) && (q.s==null ? s==null : q.s.equals(s));
64          }
65          return false;
66      }
67  }
68
69  BadPoint p = new BadPoint(1, 2);
70  BadPointExt q = new BadPointExt(1, 2, "info");
```

Given the definitions in the example, `p.equals(q)` returns `true` whereas `q.equals(p)` returns `false`, which violates the symmetry requirement of the `equals` contract.

Attempting to enforce symmetry by modifying the `BadPointExt.equals` method to ignore the field `s` when its parameter is an instance of type `BadPoint` results in violating the transitivity requirement of the `equals` contract.

The classes `GoodPoint` and `GoodPointExt` avoid violating the `equals` contract by using `getClass` rather than `instanceof`.

References

- J. Bloch, *Effective Java (second edition)*, Items 8 and 16. Addison-Wesley, 2008.
- Java API Documentation: Object.equals().
- The Java Language Specification: Type Comparison Operator instanceof.
- Artima Developer: How to Write an Equality Method in Java.

Avoid reference comparisons with operands of type 'Object'

Category: Critical > Equality

Description: Reference comparisons (== or !=) with operands where the static type is 'Object' may not work as intended.

Reference comparisons (`==` or `!=`) with operands where the static type is `Object` may not work as intended. Reference comparisons check if two objects are *identical*. To check if two objects are *equivalent*, use `Object.equals` instead.

Recommendation

Use `Object.equals` instead of `==` or `!=`, and override the default behavior of the method in a subclass, so that it uses the appropriate notion of equality.

References

- Java API Documentation: Object.equals().

Avoid unintentionally overloading 'Object.equals'

Category: Critical > Equality

Description: Defining 'Object.equals', where the parameter of 'equals' is not of the appropriate type, overloads 'equals' instead of overriding it.

Classes that define an `equals` method whose parameter type is not `Object` *overload* the `Object.equals` method instead of *overriding* it. This may not be intended.

Recommendation

To *override* the `Object.equals` method, the parameter of the `equals` method must have type `Object`.

Example

In the following example, the definition of class `BadPoint` does not override the `Object.equals` method. This means that `p.equals(q)` resolves to the default definition of `Object.equals` and returns `false`. Class `GoodPoint` correctly overrides `Object.equals`, so that `r.equals(s)` returns `true`.

```
1  class BadPoint {
2      int x;
3      int y;
4
5      BadPoint(int x, int y) {
6          this.x = x;
7          this.y = y;
8      }
9
10      // overloaded equals method -- should be avoided
11      public boolean equals(BadPoint q) {
12          return x == q.x && y == q.y;
13      }
14 }
15
16 BadPoint p = new BadPoint(1, 2);
17 Object q = new BadPoint(1, 2);
18 boolean badEquals = p.equals(q); // evaluates to false
19
20 class GoodPoint {
21      int x;
22      int y;
23
24      GoodPoint(int x, int y) {
25          this.x = x;
26          this.y = y;
27      }
28
29      // correctly overrides Object.equals(Object)
30      public boolean equals(Object obj) {
31          if (obj != null && getClass() == obj.getClass()) {
32              GoodPoint q = (GoodPoint)obj;
33              return x == q.x && y == q.y;
34          }
35          return false;
36      }
37 }
```

```
38
39 GoodPoint r = new GoodPoint(1, 2);
40 Object s = new GoodPoint(1, 2);
41 boolean goodEquals = r.equals(s); // evaluates to true
```

References

- J. Bloch, *Effective Java (second edition)*, Item 8. Addison-Wesley, 2008.
- The Java Language Specification: Overriding (by Instance Methods), Overloading.
- The Java Tutorials: Overriding and Hiding Methods.

Do not make calls of the form 'x.equals(y)' with incomparable types

Category: Critical > Equality

Description: Calls of the form 'x.equals(y)', where the types of 'x' and 'y' are incomparable, should always return 'false'.

Calls of the form `x.equals(y)`, where `x` and `y` have incomparable types, should always return `false` because the runtime types of `x` and `y` will be different. Two types are incomparable if they are distinct and do not have a common subtype.

Recommendation

Ensure that such comparisons use comparable types.

Example

In the following example, the call to `equals` on line 5 refers to the whole array by mistake, instead of a specific element. Therefore, "Value not found" is returned.

```
1  String[] anArray = new String[]{"a","b","c"}
2  String valueToFind = "b";
3
4  for(int i=0; i<anArray.length; i++){
5    if(anArray.equals(valueToFind){    // anArray[i].equals(valueToFind) was intended
6      return "Found value at index " + i;
7    }
8  }
9
10 return "Value not found";
```

References

- Java API Documentation: Object.equals().

Ensure that an implementaton of 'equals' inspects its argument type

Category: Critical > Equality

Description: An implementation of 'equals' that does not check the type of its argument may lead to failing casts.

An implementation of `equals` must be able to handle an argument of any type, to avoid failing casts. Therefore, the implementation should inspect the type of its argument to see if the argument can be safely cast to the class in which the `equals` method is declared.

Recommendation

Usually, an implementation of `equals` should check the type of its argument using `instanceof`, following the general pattern below.

```
1  class A {
2      // ...
3      public final boolean equals(Object obj) {
4          if (!(obj instanceof A)) {
5              return false;
6          }
7          A a = (A)obj;
8          // ...further checks...
9      }
10      // ...
11 }
```

Using `instanceof` in this way has the added benefit that it includes a guard against null pointer exceptions: if `obj` is `null`, the check fails and `false` is returned. Therefore, after the check, it is guaranteed that `obj` is not `null`, and its fields can be safely accessed.

Whenever you use `instanceof` to check the type of the argument, you should declare the `equals` method `final`, so that subclasses are unable to cause a violation of the symmetry requirement of the `equals` contract by further overriding `equals`.

If you want subclasses to redefine the notion of equality by overriding `equals`, use `getClass` instead of `instanceof` to check the type of the argument. However, note that the use of `getClass` prevents any equality relationship between instances of a class and its subclasses, even when no additional state is added in a subclass.

References

- J. Bloch, *Effective Java (second edition)*, Item 8. Addison-Wesley, 2008.
- Java API Documentation: Object.equals().
- The Java Language Specification: Type Comparison Operator instanceof.

Exceptions

- Avoid catching 'Throwable' or 'Exception'
- Do not dereference a variable that is 'null'
- Ensure that 'finally' blocks complete normally

Avoid catching 'Throwable' or 'Exception'

Category: Critical > Exceptions

Description: Catching 'Throwable' or 'Exception' is dangerous because these can include 'Error' or 'RuntimeException'.

Catching `Throwable` or `Exception` is dangerous because these can include an `Error` such as `OutOfMemoryError` or a `RuntimeException` such as `ArrayIndexOutOfBoundsException`. These should normally be propagated to the outermost level because they generally indicate a program state from which normal operation cannot be recovered.

Recommendation

It is usually best to ensure that exceptions that are caught in a `catch` clause are as specific as possible to avoid inadvertently suppressing more serious problems.

Example

In the following example, the `catch` clause in the first `try` block catches `Throwable`. However, when performing read operations on a `FileInputStream` within a `try` block, the corresponding `catch` clause should normally catch `IOException` instead. This is shown in the second, modified `try` block.

```
1  FileInputStream fis = ...
2  try {
3      fis.read();
4  } catch (Throwable e) {  // BAD: The exception is too general.
5      // Handle this exception
6  }
7
8  FileInputStream fis = ...
9  try {
10      fis.read();
11 } catch (IOException e) {  // GOOD: The exception is specific.
12     // Handle this exception
13 }
```

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 44. Addison-Wesley, 2005.
- Java Platform, Standard Edition 6, API Specification: Throwable, Error, Exception, RuntimeException.

Do not dereference a variable that is 'null'

Category: Critical > Exceptions

Description: Dereferencing a variable whose value is 'null' causes a 'NullPointerException'.

If a variable is dereferenced, and the variable has a `null` value on all possible execution paths leading to the dereferencing, the dereferencing is guaranteed to result in a `NullPointerException`.

Recommendation

Ensure that the variable does not have a `null` value when it is dereferenced.

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Ensure that 'finally' blocks complete normally

Category: Critical > Exceptions

Description: A 'finally' block that runs because an exception has been thrown, and that does not complete normally, causes the exception to disappear silently.

A `finally` block that does not complete normally suppresses any exceptions that may have been thrown in the corresponding `try` block. This can happen if the `finally` block contains any `return` or `throw` statements, or if it contains any `break` or `continue` statements whose jump target lies outside of the `finally` block.

Recommendation

To avoid suppressing exceptions that are thrown in a `try` block, design the code so that the corresponding `finally` block always completes normally. Remove any of the following statements that may cause it to terminate abnormally:

- `return`
- `throw`
- `break`
- `continue`

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 36. Addison-Wesley, 2005.
- The Java Language Specification: Execution of try-finally and try-catch-finally.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Expressions

- Avoid accidentally assigning to a local variable in a 'return' statement
- Avoid accidentally using a bitwise logical operator instead of a conditional operator

Avoid accidentally assigning to a local variable in a 'return' statement

Category: Critical > Expressions

Description: Assigning to a local variable in a 'return' statement has no effect.

An assignment is an expression. The value of an assignment expression is the value assigned to the variable. This can be useful, for example, when initializing two or more variables at once (for example, `a = b = 0;`). However, assigning to a local variable in the expression of a return statement is redundant because that value can never be read.

Recommendation

Remove the redundant assignment from the `return` statement, leaving just the right-hand side of the assignment.

Example

In the following example, consider the second assignment to `ret`. The variable goes out of scope when the method returns, and the value assigned to it is never read. Therefore, the assignment is redundant. Instead, the last line of the method can be changed to `return Math.max(ret, c);`

```
1 public class Utilities
2 {
3     public static int max(int a, int b, int c) {
4         int ret = Math.max(a, b)
5         return ret = Math.max(ret, c);  // Redundant assignment
6     }
7 }
```

References

- Java Language Specification: 14.17 The return Statement, 15.26 Assignment Operators.

Avoid accidentally using a bitwise logical operator instead of a conditional operator

Category: Critical > Expressions

Description: Using a bitwise logical operator on a Boolean where a conditional-and or conditional-or operator is intended is likely to give the wrong result and may cause an exception.

Using a bitwise logical operator (`&` or `|`) on a Boolean where a conditional-and or conditional-or operator (`&&` or `||`) is intended is likely to give the wrong result and may cause an exception. This is especially true if the left-hand operand is a guard for the right-hand operand.

Typically, as in the example below, this kind of defect is introduced by simply mistyping the intended logical operator rather than any conceptual mistake by the programmer.

Recommendation

If the right-hand side of an expression is only intended to be evaluated if the left-hand side evaluates to `true`, use a conditional-and.

Similarly, if the right-hand side of an expression is only intended to be evaluated if the left-hand side evaluates to `false`, use a conditional-or.

Example

In the following example, the `hasForename` method is implemented correctly. For a forename to be valid it must be a non-null string with a non-zero length. The method has two expressions (`forename != null` and `forename.length() > 0`) to check these two properties. The second check is executed only if the first succeeds, because they are combined using a conditional-and operator (`&&`).

In contrast, although `hasSurname` looks almost the same, it contains a defect. Again there are two tests (`surname != null` and `surname.length() > 0`), but they are linked by a bitwise logical operator (`&`). Both sides of a bitwise logical operator are *always* evaluated, so if `surname` is `null` the `hasSurname` method throws a `NullPointerException`. To fix the defect, change `&` to `&&`.

```
1  public class Person
2  {
3      private String forename;
4      private String surname;
5
6      public boolean hasForename() {
7          return forename != null && forename.length() > 0;  // GOOD: Conditional-and operator
8      }
9
10      public boolean hasSurname() {
11          return surname != null & surname.length() > 0;  // BAD: Bitwise AND operator
12      }
13
14      // ...
15 }
```

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 42. Addison-Wesley, 2005.
- Java Language Specification: 15.22.2 Boolean Logical Operators &, ^, and |, 15.23 Conditional-And Operator &&, 15.24 Conditional-Or Operator ||.

Extensibility

- Avoid calling 'getClass().getResource()'
- Avoid forcible termination of the JVM

Avoid calling 'getClass().getResource()'

Category: Critical > Extensibility

Description: Calling 'this.getClass().getResource()' may yield unexpected results if called from a subclass in another package.

Using the `Class.getResource` method is a common way of including some non-code resources with an application.

There are problems when this is called using `x.getClass().getResource()`, for some variable `x`. This is not a safe way to retrieve a resource. The method `getClass` returns the *run-time* class of `x` (that is, its actual, "most derived" class, rather than its declared type), which causes two potential problems:

- If the run-time type of the receiving object is a subclass of the declared type and is in a different package, the resource path may be interpreted differently. According to its contract, `Class.getResource` qualifies non-absolute paths with the current package name, thus potentially returning a different resource or failing to find the requested resource.
- `Class.getResource` delegates finding the resource to the class loader that loaded the class. At run time, there is no guarantee that all subclasses of a particular type are loaded by the same class loader, resulting in resource lookup failures that are difficult to diagnose.

Recommendation

Rather than using the `getClass` method, which relies on dynamic dispatch and run-time types, use `class` literals instead. For example, instead of calling `getClass().getResource()` on an object of type `Foo`, call `Foo.class.getResource()`. Class literals always refer to the declared type they are used on, removing the dependency on run-time types.

Example

In the following example, the calls to `getPostalCodes` return different results, depending on which class the call is made on: the class `Address` is in the package `framework` and the class `UKAddress` is in the package `client`.

```
1  package framework;
2  class Address {
3      public URL getPostalCodes() {
4          // AVOID: The call is made on the run-time type of 'this'.
5          return this.getClass().getResource("postal-codes.csv");
6      }
7  }
8
9  package client;
10  class UKAddress extends Address {
11      public void convert() {
12          // Looks up "framework/postal-codes.csv"
13          new Address().getPostalCodes();
14          // Looks up "client/postal-codes.csv"
15          new UKAddress().getPostalCodes();
16      }
17  }
```

In the following corrected example, the implementation of `getPostalCodes` is changed so that it always calls `getResource` on the same class.

```
1  package framework;
2  class Address {
3      public URL getPostalCodes() {
4          // GOOD: The call is always made on an object of the same type.
```

```
5          return Address.class.getResource("postal-codes.csv");
6      }
7  }
8
9  package client;
10  class UKAddress extends Address {
11      public void convert() {
12          // Looks up "framework/postal-codes.csv"
13          new Address().getPostalCodes();
14          // Looks up "framework/postal-codes.csv"
15          new UKAddress().getPostalCodes();
16      }
17  }
```

References

- Java Platform, Standard Edition 7, API Specification: class.getResource().

Avoid forcible termination of the JVM

Category: Critical > Extensibility

Description: Calling 'System.exit', 'Runtime.halt', or 'Runtime.exit' may make code harder to reuse and prevent important cleanup steps from running.

Calling one of the methods `System.exit`, `Runtime.halt`, and `Runtime.exit` immediately terminates the Java Virtual Machine (JVM), effectively killing all threads without giving any of them a chance to perform cleanup actions or recover. As such, it is a dangerous thing to do: firstly, it can terminate the entire program inadvertently, and secondly, it can prevent important resources from being released or program state from being written to disk consistently.

It is sometimes considered acceptable to call `System.exit` from a program's `main` method in order to indicate the overall exit status of the program. Such calls are an exception to this rule.

Recommendation

It is usually preferable to use a different mechanism for reporting failure conditions. Consider returning a special value (perhaps `null`) that users of the current method check for and recover from appropriately. Alternatively, throw a suitable exception, which unwinds the stack and allows properly written code to clean up after itself, while leaving other threads undisturbed.

Example

In the following example, problem 1 shows that `FileOutput.write` tries to write some data to disk and terminates the JVM if this fails. This leaves the partially-written file on disk without any cleanup code running. It would be better to either return `false` to indicate the failure, or let the `IOException` propagate upwards and be handled by a method that knows how to recover.

Problem 2 is more subtle. In this example, there is just one entry point to the program (the `main` method), which constructs an `Action` and performs it. `Action.run` calls `System.exit` to indicate successful completion. Consider, however, how this code might be integrated in an application server that constructs `Action` instances and calls `run` on them without going through `main`. The fact that `run` terminates the JVM instead of returning its exit code as an integer makes that use-case impossible.

```
1  // Problem 1: Miss out cleanup code
2  class FileOutput {
3      boolean write(String[] s) {
4          try {
5              output.write(s.getBytes());
6          } catch (IOException e) {
7              System.exit(1);
8          }
9          return true;
10     }
11 }
12
13 // Problem 2: Make code reuse difficult
14 class Action {
15     public void run() {
16         // ...
17         // Perform tasks ...
18         // ...
19         System.exit(0);
20     }
21     public static void main(String[] args) {
```

```
22          new Action(args).run();
23      }
24  }
```

References

- J. Bloch, *Effective Java (second edition)*, p. 232. Addison-Wesley, 2008.
- Java Platform, Standard Edition 7, API Specification: System.exit(int), Runtime.halt(int), Runtime.exit(int).

Incomplete Code

- Avoid empty blocks or statements
- Avoid empty statements
- Ensure that a 'switch' includes cases for all 'enum' constants

Avoid empty blocks or statements

Category: Critical > Incomplete Code

Description: An undocumented empty block or statement hinders readability. It may also indicate incomplete code.

An unexplained empty block or statement makes the code less readable. It might also indicate missing code, a misplaced semicolon, or a misplaced brace. For these reasons, it should be avoided.

Recommendation

If a block is empty because some code is missing, add the code.

If an `if` statement has an empty `then` branch and a non-empty `else` branch, it may be possible to negate the condition and move the statements of the `else` branch into the `then` branch.

If a block is deliberately empty, add a comment to explain why.

Example

In the following example, the `while` loop has intentionally been left empty. The purpose of the loop is to scan a `String` for the first occurrence of the character `'='`. A programmer reading the code might not understand the reason for the empty loop body, and think that something is missing, or perhaps even that the loop is useless. Therefore it is a good practice to add a comment to an empty block explaining why it is empty.

```
1  public class Parser
2  {
3      public void parse(String input) {
4          int pos = 0;
5          // ...
6          // AVOID: Empty block
7          while (input.charAt(pos++) != '=') { }
8          // ...
9      }
10 }
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Language Specification: 14.2 Blocks, 14.6 The Empty Statement, 14.9 The if Statement, 14.12 The while Statement, 14.13 The do Statement, 14.14 The for Statement.

Avoid empty statements

Category: Critical > Incomplete Code

Description: An empty statement hinders readability.

An *empty statement* is a single semicolon `;` that does not terminate another statement. Such a statement hinders readability and has no effect on its own.

Recommendation

Avoid empty statements. If a loop is intended to have an empty body, it is better to mark that fact explicitly by using a pair of braces `{}` containing an explanatory comment for the body, rather than a single semicolon.

Example

In the following example, there is an empty statement on line 3, where an additional semicolon is used. On line 6, the `for` statement has an empty body because the condition is immediately followed by a semicolon. In this case, it is better to include a pair of braces `{}` containing an explanatory comment for the body instead.

$body

```
1 public class Cart {
2     // AVOID: Empty statement
3     List<Item> items = new ArrayList<Cart>();;
4     public void applyDiscount(float discount) {
5         // AVOID: Empty statement as loop body
6         for (int i = 0; i < items.size(); items.get(i++).applyDiscount(discount));
7     }
8 }
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Ensure that a 'switch' includes cases for all 'enum' constants

Category: Critical > Incomplete Code

Description: A 'switch' statement that is based on an 'enum' type and does not have cases for all the 'enum' constants is usually a coding mistake.

A `switch` statement that is based on a variable with an `enum` type should either have a default case or handle all possible constants of that `enum` type. Handling all but one or two `enum` constants is usually a coding mistake.

Recommendation

If there are only a handful of missing cases, add them to the end of the `switch` statement. If there are many cases that do not need to be handled individually, add a default case to handle them.

If there are some `enum` constants that should never occur in this particular part of the code, then program defensively by adding cases for those constants and explicitly throwing an exception (rather than just having no cases for those constants).

Example

In the following example, the case for 'YES' is missing. Therefore, if `answer` is 'YES', an exception is thrown at run time. To fix this, a case for 'YES' should be added.

```
1  enum Answer { YES, NO, MAYBE }
2
3  class Optimist
4  {
5      Answer interpet(Answer answer) {
6          switch (answer) {
7              case MAYBE:
8                  return Answer.YES;
9              case NO:
10                  return Answer.MAYBE;
11              // Missing case for 'YES'
12          }
13          throw new RuntimeException("uncaught case: " + answer);
14      }
15 }
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Language Specification: 8.9 Enums, 14.11 The switch Statement.

Java objects

- Cloning
- Garbage collection
- Serialization

Cloning

- Ensure that a subclass 'clone' method calls 'super.clone'

Ensure that a subclass 'clone' method calls 'super.clone'

Category: Critical > Java objects > Cloning

Description: A 'clone' method that is overridden in a subclass, and that does not itself call 'super.clone', causes calls to the subclass's 'clone' method to return an object of the wrong type.

A `clone` method that is overridden in a subclass should call `super.clone`. Not doing so causes the subclass `clone` to return an object of the wrong type, which violates the contract for `Cloneable`.

The Java API documentation states that, for an object `x`, the general intent of the `clone` method is for it to satisfy the following three properties:

- `x.clone() != x` (the cloned object is a different object instance)
- `x.clone().getClass() == x.getClass()` (the cloned object is the same type as the source object)
- `x.clone().equals(x)` (the cloned object has the same 'contents' as the source object)

For the cloned object to be of the same type as the source object, non-final classes must call `super.clone` and that call must eventually reach `Object.clone`, which creates an instance of the right type. If it were to create a new object using a constructor, a subclass that does not implement the `clone` method returns an object of the wrong type. In addition, all of the class's supertypes that also override `clone` must call `super.clone`. Otherwise, it never reaches `Object.clone` and creates an object of the incorrect type.

However, as `Object.clone` only does a shallow copy of the fields of an object, any `Cloneable` objects that have a "deep structure" (for example, objects that use an array or `Collection`) must take the clone that results from the call to `super.clone` and assign explicitly created copies of the structure to the clone's fields. This means that the cloned instance does not share its internal state with the source object. If it *did* share its internal state, any changes made in the cloned object would also affect the internal state of the source object, probably causing unintended behavior.

One added complication is that `clone` cannot modify values in final fields, which would be already set by the call to `super.clone`. Some fields must be made non-final to correctly implement the `clone` method.

Recommendation

Every clone method should always use `super.clone` to construct the cloned object. This ensures that the cloned object is ultimately constructed by `Object.clone`, which uses reflection to ensure that an object of the correct runtime type is created.

Example

In the following example, the attempt to clone `WrongEmployee` fails because `super.clone` is implemented incorrectly in its superclass `WrongPerson`.

```
1  class WrongPerson implements Cloneable {
2      private String name;
3      public WrongPerson(String name) { this.name = name; }
4      // BAD: 'clone' does not call 'super.clone'.
5      public WrongPerson clone() {
6          return new WrongPerson(this.name);
7      }
8  }
9
10 class WrongEmployee extends WrongPerson {
11     public WrongEmployee(String name) {
12         super(name);
13     }
14     // ALMOST RIGHT: 'clone' correctly calls 'super.clone',
```

```
    // but 'super.clone' is implemented incorrectly.
    public WrongEmployee clone() {
        return (WrongEmployee)super.clone();
    }
}

public class MissingCallToSuperClone {
    public static void main(String[] args) {
        WrongEmployee e = new WrongEmployee("John Doe");
        WrongEmployee eclone = e.clone(); // Causes a ClassCastException
    }
}
```

However, in the following modified example, the attempt to clone `Employee` succeeds because `super.clone` is implemented correctly in its superclass `Person`.

```
class Person implements Cloneable {
    private String name;
    public Person(String name) { this.name = name; }
    // GOOD: 'clone' correctly calls 'super.clone'
    public Person clone() {
        try {
            return (Person)super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError("Should never happen");
        }
    }
}

class Employee extends Person {
    public Employee(String name) {
        super(name);
    }
    // GOOD: 'clone' correctly calls 'super.clone'
    public Employee clone() {
        return (Employee)super.clone();
    }
}

public class MissingCallToSuperClone {
    public static void main(String[] args) {
        Employee e2 = new Employee("Jane Doe");
        Employee e2clone = e2.clone(); // 'clone' correctly returns an object of type 'Employee'
    }
}
```

References

- J. Bloch, *Effective Java (second edition)*, Item 11. Addison-Wesley, 2008.
- Java 6 API Specification: Object.clone().

Garbage collection

- Do not call 'System.runFinalizersOnExit' or 'Runtime.runFinalizersOnExit'

Do not call 'System.runFinalizersOnExit' or 'Runtime.runFinalizersOnExit'

Category: Critical > Java objects > Garbage collection

Description: Calling 'System.runFinalizersOnExit' or 'Runtime.runFinalizersOnExit' may cause finalizers to be run on live objects, leading to erratic behavior or deadlock.

Avoid calling `System.runFinalizersOnExit` or `Runtime.runFinalizersOnExit`, which are considered to be dangerous methods.

The Java Development Kit documentation for `System.runFinalizersOnExit` states:

> *This method is inherently unsafe. It may result in finalizers being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock.*

Object finalizers are normally only called when the object is about to be collected by the garbage collector. Using `runFinalizersOnExit` sets a Java Virtual Machine-wide flag that executes finalizers *on all objects with a `finalize` method* before the runtime exits. This would require all objects with finalizers to defend against the possibility of `finalize` being called when the object is still in use, which is not practical for most applications.

Recommendation

Ensure that the code does not rely on the execution of finalizers. If the code is dependent on the garbage collection behavior of the Java Virtual Machine, there is no guarantee that finalizers will be executed in a timely manner, or at all. This may become a problem if finalizers are used to dispose of limited system resources, such as file handles.

Instead of finalizers, use explicit `dispose` methods in `finally` blocks, to make sure that an object's resources are released.

Example

The following example shows a program that calls `runFinalizersOnExit`, which is not recommended.

```
1 void main() {
2     // ...
3     // BAD: Call to 'runFinalizersOnExit' forces execution of all finalizers on termination of
4     // the runtime, which can cause live objects to transition to an invalid state.
5     // Avoid using this method (and finalizers in general).
6     System.runFinalizersOnExit(true);
7     // ...
8 }
```

The following example shows the recommended approach: a program that calls a `dispose` method in a `finally` block.

```
1 // Instead of using finalizers, define explicit termination methods
2 // and call them in 'finally' blocks.
3 class LocalCache {
4     private Collection<File> cacheFiles = ...;
5
6     // Explicit method to close all cacheFiles
7     public void dispose() {
8         for (File cacheFile : cacheFiles) {
9             disposeCacheFile(cacheFile);
10         }
```

```
11      }
12  }
13
14  void main() {
15      LocalCache cache = new LocalCache();
16      try {
17          // Use the cache
18      } finally {
19          // Call the termination method in a 'finally' block, to ensure that
20          // the cache's resources are freed.
21          cache.dispose();
22      }
23  }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 7. Addison-Wesley, 2008.
- Java 6 API Documentation: System.runFinalizersOnExit(), Object.finalize().
- Java SE Documentation: Java Thread Primitive Deprecation.

Serialization

- Ensure that a 'serialVersionUID' field that is declared in a serializable class is of the correct type
- Ensure that a class that implements 'Comparator' and is used to construct a sorted collection is serializable
- Ensure that a non-serializable, immediate superclass of a serializable class declares a default constructor
- Ensure that a non-static, serializable nested class is enclosed in a serializable class

Ensure that a 'serialVersionUID' field that is declared in a serializable class is of the correct type

Category: Critical > Java objects > Serialization

Description: A 'serialVersionUID' field that is declared in a serializable class but is of the wrong type cannot be used by the serialization framework.

A serializable class that uses the `serialVersionUID` field to act as an object version number must declare the field to be `final`, `static`, and of type `long` for it to be used by the Java serialization framework.

Recommendation

Make sure that the `serialVersionUID` field in a serialized class is final, static, and of type `long`.

Example

In the following example, `WrongNote` defines `serialVersionUID` using the wrong type, so that it is not used by the Java serialization framework. However, `Note` defines it correctly so that it is used by the framework.

```
1  class WrongNote implements Serializable {
2      // BAD: serialVersionUID must be static, final, and 'long'
3      private static final int serialVersionUID = 1;
4
5      //...
6  }
7
8  class Note implements Serializable {
9      // GOOD: serialVersionUID is of the correct type
10      private static final long serialVersionUID = 1L;
11 }
```

References

- Java API Documentation: Serializable.
- JavaWorld: Ensure proper version control for serialized objects.

Ensure that a class that implements 'Comparator' and is used to construct a sorted collection is serializable

Category: Critical > Java objects > Serialization

Description: A comparator that is passed to an ordered collection (for example, a treemap) must be serializable, otherwise the collection fails to serialize at run-time.

A class that implements `java.util.Comparator` and is used to construct a sorted collection needs to be serializable. An ordered collection (such as a `java.util.TreeMap`) that is constructed using a comparator serializes successfully only if the comparator is serializable.

The `Collection`s in the Java Standard Library that require a comparator (`TreeSet`, `TreeMap`, `PriorityQueue`) all call `ObjectOutputStream.defaultWriteObject`, which tries to serialize every non-static, non-transient field in the class. As the comparator is stored in a field in these collections, the attempt to serialize a non-serializable comparator throws a `java.io.NotSerializableException`.

Recommendation

Comparators should be serializable if they are used in sorted collections that may be serialized. In most cases, simply changing the comparator so it also implements `Serializable` is enough. Comparators that have internal state may require additional changes (for example, custom `writeObject` and `readObject` methods). In these cases, it is best to follow general best practices for serializable objects (see references below).

Example

In the following example, `WrongComparator` is not serializable because it does not implement `Serializable`. However, `StringComparator` is serializable because it does implement `Serializable`.

$body

```
1  // BAD: This is not serializable, and throws a 'java.io.NotSerializableException'
2  // when used in a serializable sorted collection.
3  class WrongComparator implements Comparator<String> {
4      public int compare(String o1, String o2) {
5          return o1.compareTo(o2);
6      }
7  }
8
9  // GOOD: This is serializable, and can be used in collections that are meant to be serialized.
10  class StringComparator implements Comparator<String>, Serializable {
11      private static final long serialVersionUID = -5972458403679726498L;
12
13      public int compare(String arg0, String arg1) {
14          return arg0.compareTo(arg1);
15      }
16  }
```

References

- Java API Documentation: Comparator, ObjectOutputStream, Serializable.

Ensure that a non-serializable, immediate superclass of a serializable class declares a default constructor

Category: Critical > Java objects > Serialization

Description: A non-serializable, immediate superclass of a serializable class that does not itself declare an accessible, no-argument constructor causes deserialization to fail.

A serializable class that is a subclass of a non-serializable class cannot be deserialized if its superclass does not declare a no-argument constructor. The Java serialization framework uses the no-argument constructor when it initializes the object instance that is created during deserialization. Deserialization fails with an `InvalidClassException` if its superclass does not declare a no-argument constructor.

The Java Development Kit API documentation states:

> *To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields. The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state. It is an error to declare a class `Serializable` if this is not the case. The error will be detected at runtime.*

Recommendation

Make sure that every non-serializable class that is extended by a serializable class has a no-argument constructor.

Example

In the following example, the class `WrongSubItem` cannot be deserialized because its superclass `WrongItem` does not declare a no-argument constructor. However, the class `SubItem` *can* be serialized because it declares a no-argument constructor.

```
1  class WrongItem {
2      private String name;
3
4      // BAD: This class does not have a no-argument constructor, and throws an
5      // 'InvalidClassException' at runtime.
6
7      public WrongItem(String name) {
8          this.name = name;
9      }
10 }
11
12 class WrongSubItem extends WrongItem implements Serializable {
13     public WrongSubItem() {
14         super(null);
15     }
16
17     public WrongSubItem(String name) {
18         super(name);
19     }
20 }
21
22 class Item {
23     private String name;
24
25     // GOOD: This class declares a no-argument constructor, which allows serializable
```

```
26      // subclasses to be deserialized without error.
27      public Item() {}
28
29      public Item(String name) {
30          this.name = name;
31      }
32  }
33
34  class SubItem extends Item implements Serializable {
35      public SubItem() {
36          super(null);
37      }
38
39      public SubItem(String name) {
40          super(name);
41      }
42  }
```

References

- Java API Documentation: Serializable.
- J. Bloch, *Effective Java (second edition)*, Item 74. Addison-Wesley, 2008.

Ensure that a non-static, serializable nested class is enclosed in a serializable class

Category: Critical > Java objects > Serialization

Description: A class that is serializable with an enclosing class that is not serializable causes serialization to fail.

Non-static nested classes that implement `Serializable` must be defined in an enclosing class that is also serializable. Non-static nested classes retain an implicit reference to an instance of their enclosing class. If the enclosing class is not serializable, the Java serialization mechanism fails with a `java.io.NotSerializableException`.

Recommendation

To avoid causing a `NotSerializableException`, do one of the following:

- **Declare the nested class as `static` :** If the nested class does not use any of the non-static fields or methods of the enclosing class, it is best to declare it `static`. This removes the implicit reference to an instance of the enclosing class, and has the additional effect of breaking an unnecessary dependency between the two classes. A similar solution is to turn the nested class into a separate top-level class.
- **Make the enclosing class implement `Serializable` :** However, this is not recommended because the implementation of inner classes may be compiler-specific, and serializing an inner class can result in non-portability across compilers. The Java Serialization Specification states:

> *Serialization of inner classes (i.e., nested classes that are not static member classes), including local and anonymous classes, is strongly discouraged for several reasons. Because inner classes declared in non-static contexts contain implicit non-transient references to enclosing class instances, serializing such an inner class instance will result in serialization of its associated outer class instance as well. Synthetic fields generated by javac (or other Java(TM) compilers) to implement inner classes are implementation dependent and may vary between compilers; differences in such fields can disrupt compatibility as well as result in conflicting default serialVersionUID values. The names assigned to local and anonymous inner classes are also implementation dependent and may differ between compilers.*

Example

In the following example, the class `WrongSession` cannot be serialized without causing a `NotSerializableException`, because it is enclosed by a non-serializable class. However, the class `Session` can be serialized because it is declared as `static`.

```
1  class NonSerializableServer {
2
3      // BAD: The following class is serializable, but the enclosing class
4      // 'NonSerializableServer' is not. Serializing an instance of 'WrongSession'
5      // causes a 'java.io.NotSerializableException'.
6      class WrongSession implements Serializable {
7          private static final long serialVersionUID = 8970783971992397218L;
8          private int id;
9          private String user;
10
11          WrongSession(int id, String user) { /*...*/ }
12      }
13
```

```
14      public WrongSession getNewSession(String user) {
15          return new WrongSession(newId(), user);
16      }
17  }
18
19  class Server {
20
21      // GOOD: The following class can be correctly serialized because it is static.
22      static class Session implements Serializable {
23          private static final long serialVersionUID = 1065454318648105638L;
24          private int id;
25          private String user;
26
27          Session(int id, String user) { /*...*/ }
28      }
29
30      public Session getNewSession(String user) {
31          return new Session(newId(), user);
32      }
33  }
```

References

- Java 6 Object Serialization Specification: 1.10 The Serializable Interface, 2.1 The ObjectOutputStream Class.

Logic Errors

- Annotate annotations with a 'RUNTIME' retention policy
- Avoid array downcasts
- Avoid type mismatch when calling 'Collection.contains'
- Avoid type mismatch when calling 'Collection.remove'
- Do not call a non-final method from a constructor
- Do not perform self-assignment
- Include braces for control structures

Annotate annotations with a 'RUNTIME' retention policy

Category: Critical > Logic Errors

Description: If an annotation has not been annotated with a 'RUNTIME' retention policy, checking for its presence at runtime is not possible.

To be able to use the `isAnnotationPresent` method on an `AnnotatedElement` at runtime, an annotation must be explicitly annotated with a `RUNTIME` retention policy. Otherwise, the annotation is not retained at runtime and cannot be observed using reflection.

Recommendation

Explicitly annotate annotations with a `RUNTIME` retention policy if you want to observe their presence using `AnnotatedElement.isAnnotationPresent` at runtime.

Example

In the following example, the call to `isAnnotationPresent` returns `false` because the annotation cannot be observed using reflection.

```
1  public class AnnotationPresentCheck {
2      public static @interface UntrustedData { }
3
4      @UntrustedData
5      public static String getUserData() {
6          Scanner scanner = new Scanner(System.in);
7          return scanner.nextLine();
8      }
9
10      public static void main(String[] args) throws NoSuchMethodException, SecurityException {
11          String data = getUserData();
12          Method m = AnnotationPresentCheck.class.getMethod("getUserData");
13          if(m.isAnnotationPresent(UntrustedData.class)) {  // Returns 'false'
14              System.out.println("Not trusting data from user.");
15          }
16      }
17  }
```

To correct this, the annotation is annotated with a `RUNTIME` retention policy.

```
1  public class AnnotationPresentCheckFix {
2      @Retention(RetentionPolicy.RUNTIME)  // Annotate the annotation
3      public static @interface UntrustedData { }
4
5      @UntrustedData
6      public static String getUserData() {
7          Scanner scanner = new Scanner(System.in);
8          return scanner.nextLine();
9      }
10
11      public static void main(String[] args) throws NoSuchMethodException, SecurityException {
12          String data = getUserData();
13          Method m = AnnotationPresentCheckFix.class.getMethod("getUserData");
14          if(m.isAnnotationPresent(UntrustedData.class)) {  // Returns 'true'
15              System.out.println("Not trusting data from user.");
16          }
17      }
18  }
```

References

- Java API Documentation: Annotation Type Retention, RetentionPolicy.RUNTIME, AnnotatedElement.isAnnotationPresent().

Avoid array downcasts

Category: Critical > Logic Errors

Description: Trying to cast an array of a particular type as an array of a subtype causes a 'ClassCastException' at runtime.

Some downcasts on arrays will fail at runtime. An object `a` with dynamic type `A[]` cannot be cast to `B[]`, where `B` is a subtype of `A`, even if all the elements of `a` can be cast to `B`.

Recommendation

Ensure that the array creation expression constructs an array object of the right type.

Example

The following example shows an assignment that throws a `ClassCastException` at runtime.

```
1  String[] strs = (String[])new Object[]{ "hello", "world" };
```

To avoid the exception, a `String` array should be created instead.

```
1  String[] strs = new String[]{ "hello", "world" };
```

References

- The Java Language Specification: Checked Casts at Run Time, Reference Type Casting, Subtyping among Array Types.

Avoid type mismatch when calling 'Collection.contains'

Category: Critical > Logic Errors

Description: Calling 'Collection.contains' with an object of a different type than that of the collection is unlikely to return 'true'.

The `contains` method of the `Collection` interface has an argument of type `Object`. Therefore, you can try to check if an object of any type is a member of a collection, regardless of the collection's element type. However, although you can call `contains` with an argument of a different type than that of the collection, it is unlikely that the collection actually contains an object of this type.

Recommendation

Ensure that you use the correct argument with a call to `contains`.

Example

In the following example, although the argument to `contains` is an integer, the code does not result in a type error because the argument does not have to match the type of the elements of `list`. However, the argument is unlikely to be found (and the body of the `if` statement is therefore not executed), so it is probably a typographical error: the argument should be enclosed in quotation marks.

```
1 void m(List<String> list) {
2     if (list.contains(123)) {  // Call 'contains' with non-string argument (without quotation marks)
3         // ...
4     }
5 }
```

Note that you must take particular care when working with collections over boxed types, as illustrated in the following example. The first call to `contains` returns `false` because you cannot compare two boxed numeric primitives of different types, in this case `Short(1)` (in `set`) and `Integer(1)` (the argument). The second call to `contains` returns `true` because you can compare `Short(1)` and `Short(1)`.

```
1 HashSet<Short> set = new HashSet<Short>();
2 short s = 1;
3 set.add(s);
4 // Following statement prints 'false', because the argument is a literal int, which is auto-boxed
5 // to an Integer
6 System.out.println(set.contains(1));
7 // Following statement prints 'true', because the argument is a literal int that is cast to a short,
8 // which is auto-boxed to a Short
9 System.out.println(set.contains((short)1));
```

References

- Java API Documentation: Collection.contains.

Avoid type mismatch when calling 'Collection.remove'

Category: Critical > Logic Errors

Description: Calling 'Collection.remove' with an object of a different type than that of the collection is unlikely to have any effect.

The `remove` method of the `Collection` interface has an argument of type `Object`. Therefore, you can try to remove an object of any type from a collection, regardless of the collection's element type. However, although you can call `remove` with an argument of a different type than that of the collection, it is unlikely that the collection actually contains an object of this type.

Recommendation

Ensure that you use the correct argument with a call to `remove`.

Example

In the following example, although the argument to `contains` is an integer, the code does not result in a type error because the argument to `remove` does not have to match the type of the elements of `list`. However, the argument is unlikely to be found and removed (and the body of the `if` statement is therefore not executed), so it is probably a typographical error: the argument should be enclosed in quotation marks.

```
1 void m(List<String> list) {
2     if (list.remove(123)) {  // Call 'remove' with non-string argument (without quotation marks)
3         // ...
4     }
5 }
```

Note that you must take particular care when working with collections over boxed types, as illustrated in the following example. The first call to `remove` fails because you cannot compare two boxed numeric primitives of different types, in this case `Short(1)` (in `set`) and `Integer(1)` (the argument). Therefore, `remove` cannot find the item to remove. The second call to `remove` succeeds because you can compare `Short(1)` and `Short(1)`. Therefore, `remove` can find the item to remove.

```
1 HashSet<Short> set = new HashSet<Short>();
2 short s = 1;
3 set.add(s);
4 // Following statement fails, because the argument is a literal int, which is auto-boxed
5 // to an Integer
6 set.remove(1);
7 System.out.println(set); // Prints [1]
8 // Following statement succeeds, because the argument is a literal int that is cast to a short,
9 // which is auto-boxed to a Short
10 set.remove((short)1);
11 System.out.println(set); // Prints []
```

References

- Java API Documentation: Collection.remove.

Do not call a non-final method from a constructor

Category: Critical > Logic Errors

Description: If a constructor calls a method that is overridden in a subclass, the result can be unpredictable.

If a constructor calls a method that is overridden in a subclass, it can cause the overriding method in the subclass to be called before the subclass has been initialized. This can lead to unexpected results.

Recommendation

Do not call a non-final method from within a constructor if that method could be overridden in a subclass.

Example

In the following example, executing `new Sub("test")` results in a `NullPointerException`. This is because the subclass constructor implicitly calls the superclass constructor, which in turn calls the overridden `init` method before the field `s` is initialized in the subclass constructor.

```
1  public class Super {
2      public Super() {
3          init();
4      }
5
6      public void init() {
7      }
8  }
9
10  public class Sub extends Super {
11      String s;
12      int length;
13
14      public Sub(String s) {
15          this.s = s==null ? "" : s;
16      }
17
18      @Override
19      public void init() {
20          length = s.length();
21      }
22  }
```

To avoid this problem:

- The `init` method in the super constructor should be made `final` or `private`.
- The initialization that is performed in the overridden `init` method in the subclass can be moved to the subclass constructor itself, or delegated to a separate final or private method that is called from within the subclass constructor.

References

- J. Bloch, *Effective Java (second edition)*, pp. 89–90. Addison-Wesley, 2008.
- The Java Tutorials: Writing Final Classes and Methods.

Do not perform self-assignment

Category: Critical > Logic Errors

Description: Assigning a variable to itself has no effect.

Assigning a variable to itself does not have any effect. Therefore, such an assignment is either completely unnecessary, or it indicates a typo or a similar mistake.

Recommendation

If the assignment is unnecessary, remove it. If the assignment indicates a typo or a similar mistake, correct the mistake.

Example

The following example shows part of a method that is intended to make a copy of an existing `MotionEvent` without preserving its history. On line 8, `o.mFlags` is assigned to itself. Given that the statement is surrounded by statements that transfer information from the fields of `o` to the fields of the new event, `ev`, the statement is clearly a mistake. To correct this, the `mFlags` value should be assigned to `ev.mFlags` instead, as shown in the corrected method.

```
1  static public MotionEvent obtainNoHistory(MotionEvent o) {
2      MotionEvent ev = obtain(o.mNumPointers, 1);
3      ev.mDeviceId = o.mDeviceId;
4      o.mFlags = o.mFlags;  // Variable is assigned to itself
5      ...
6  }
7
8  static public MotionEvent obtainNoHistory(MotionEvent o) {
9      MotionEvent ev = obtain(o.mNumPointers, 1);
10      ev.mDeviceId = o.mDeviceId;
11      ev.mFlags = o.mFlags;  // Variable is assigned correctly
12      ...
13 }
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Include braces for control structures

Category: Critical > Logic Errors

Description: If a control structure does not use braces, misleading indentation makes it difficult to see which statements are within its scope.

A control structure (`if` statements and loops) has a body that is either a block of statements or a single statement. The second option may be indicated by omitting the braces: `{` and `}`.

However, omitting the braces can lead to confusion, especially if the indentation of the code suggests that multiple statements are within the body of a control structure when in fact they are not.

Recommendation

It is usually considered good practice to include braces for all control structures in Java. This is because it makes it easier to maintain the code later. For example, it's easy to see at a glance which part of the code is in the scope of an `if` statement, and adding more statements to the body of the `if` statement is less error-prone.

You should also ensure that the indentation of the code is consistent with the actual flow of control, so that it does not confuse programmers.

Example

In the example below, the original version of `Cart` is missing braces. This means that the code triggers a `NullPointerException` at runtime if `i` is `null`. The corrected version of `Cart` does include braces, so that the code executes as the indentation suggests.

```
1  class Cart {
2      Map<Integer, Integer> items = ...
3      public void addItem(Item i) {
4          // No braces and misleading indentation.
5          if (i != null)
6              log("Adding item: " + i);
7              Integer curQuantity = items.get(i.getID());  // Indentation suggests that this statement
8                                                           // is in the body of the 'if'
9              if (curQuantity == null) curQuantity = 0;
10              items.put(i.getID(), curQuantity+1);
11     }
12 }
13
14 class Cart {
15     Map<Integer, Integer> items = ...
16     public void addItem(Item i) {
17         // Braces included.
18         if (i != null) {
19             log("Adding item: " + i);
20             Integer curQuantity = items.get(i.getID());
21             if (curQuantity == null) curQuantity = 0;
22             items.put(i.getID(), curQuantity+1);
23         }
24     }
25 }
```

References

- Java SE Documentation: Compound Statements.

Naming

- Avoid declaring a method with the same name as its declaring type
- Avoid naming a method with the same name as a superclass method but with different capitalization

Avoid declaring a method with the same name as its declaring type

Category: Critical > Naming

Description: A method that has the same name as its declaring type may have been intended to be a constructor.

A method that has the same name as its declaring type may be intended to be a constructor, not a method.

Example

The following example shows how the singleton design pattern is often misimplemented. The programmer intends the constructor of `MasterSingleton` to be protected so that it cannot be instantiated (because the singleton instance should be retrieved using `getInstance`). However, the programmer accidentally wrote `void` in front of the constructor name, which makes it a method rather than a constructor.

```
1  class MasterSingleton
2  {
3      // ...
4
5      private static MasterSingleton singleton = new MasterSingleton();
6      public static MasterSingleton getInstance() { return singleton; }
7
8      // Make the constructor 'protected' to prevent this class from being instantiated.
9      protected void MasterSingleton() { }
10 }
```

Recommendation

Ensure that methods that have the same name as their declaring type are intended to be methods. Even if they are intended to be methods, it may be better to rename them to avoid confusion.

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 63. Addison-Wesley, 2005.
- E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Objection-Oriented Software*, §3. Addison-Wesley Longman Publishing Co. Inc., 1995.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Language Specification: 8.4 Method Declarations, 8.8 Constructor Declarations.

Avoid naming a method with the same name as a superclass method but with different capitalization

Category: Critical > Naming

Description: A method that would override another method but does not, because the name is capitalized differently, is confusing and may be a mistake.

If a method that would override another method but does not because the name is capitalized differently, there are two possibilities:

- The programmer intends the method to override the other method, and the difference in capitalization is a typographical error.
- The programmer does not intend the method to override the other method, in which case the similarity of the names is very confusing.

Recommendation

If overriding *is* intended, make the capitalization of the two methods the same.

If overriding is *not* intended, consider naming the methods to make the distinction between them clear.

Example

In the following example, `toString` has been wrongly capitalized as `tostring`. This means that objects of type `Customer` do not print correctly.

```
1  public class Customer
2  {
3      private String title;
4      private String forename;
5      private String surname;
6
7      // ...
8
9      public String tostring() {  // Incorrect capitalization of 'toString'
10         return title + " " + forename + " " + surname;
11     }
12 }
```

References

- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, §17.N4. Prentice Hall, 2008.

Random

- Avoid using 'Math.abs' to generate a non-negative random integer

Avoid using 'Math.abs' to generate a non-negative random integer

Category: Critical > Random

Description: Calling 'Math.abs' to find the absolute value of a randomly generated integer is not guaranteed to return a non-negative integer.

Using `Math.abs` on the result of a call to `Random.nextInt()` (or `Random.nextLong()`) is not guaranteed to return a non-negative number. `Random.nextInt()` can return `Integer.MIN_VALUE`, which when passed to `Math.abs` results in the same value, `Integer.MIN_VALUE`. (Because of the two's-complement representation of integers in Java, the positive equivalent of `Integer.MIN_VALUE` cannot be represented in the same number of bits.) The case for `Random.nextLong()` is similar.

Recommendation

If a non-negative random integer is required, use `Random.nextInt(int)` instead, and use `Integer.MAX_VALUE` as its parameter. The values that might be returned do not include `Integer.MAX_VALUE` itself, but this solution is likely to be sufficient for most purposes.

Another solution is to increment the value of `Random.nextInt()` by one, if it is negative, before passing the result to `Math.abs`. This solution has the advantage that `0` has the same probability as other numbers.

Example

In the following example, `mayBeNegativeInt` is negative if `nextInt` returns `Integer.MIN_VALUE`. The example shows how using the two solutions described above means that `positiveInt` is always assigned a positive number.

```
1  public static void main(String args[]) {
2      Random r = new Random();
3
4      // BAD: 'mayBeNegativeInt' is negative if
5      // 'nextInt()' returns 'Integer.MIN_VALUE'.
6      int mayBeNegativeInt = Math.abs(r.nextInt());
7
8      // GOOD: 'nonNegativeInt' is always a value between 0 (inclusive)
9      // and Integer.MAX_VALUE (exclusive).
10      int nonNegativeInt = r.nextInt(Integer.MAX_VALUE);
11
12      // GOOD: When 'nextInt' returns a negative number increment the returned value.
13      int nextInt = r.nextInt();
14      if(nextInt < 0)
15          nextInt++;
16      int nonNegativeInt = Math.abs(nextInt);
17 }
```

References

- Java API Documentation: Math.abs(int), Math.abs(long), Random.
- Java Language Specification, 3rd ed: 4.2.1 Integral Types and Values.
- JavaSolutions, April 2002: Secrets of equals().

Resource Leaks

- Ensure that an input resource is closed on completion
- Ensure that an output resource is closed on completion

Ensure that an input resource is closed on completion

Category: Critical > Resource Leaks

Description: A resource that is opened for reading but not closed may cause a resource leak.

A subclass of `Reader` or `InputStream` that is opened for reading but not closed may cause a resource leak.

Recommendation

Ensure that the resource is always closed to avoid a resource leak. Note that, because of exceptions, it is safest to close a resource in a `finally` block. (However, this is unnecessary for subclasses of `StringReader` and `ByteArrayInputStream`.)

Example

In the following example, the resource `br` is opened but not closed.

```
1  public class CloseReader {
2      public static void main(String[] args) throws IOException {
3          BufferedReader br = new BufferedReader(new FileReader("C:\\test.txt"));
4          System.out.println(br.readLine());
5          // ...
6      }
7  }
```

In the following example, the resource `br` is opened in a `try` block and later closed in a `finally` block.

```
1  public class CloseReaderFix {
2      public static void main(String[] args) throws IOException {
3          BufferedReader br = null;
4          try {
5              br = new BufferedReader(new FileReader("C:\\test.txt"));
6              System.out.println(br.readLine());
7          }
8          finally {
9              if(br != null)
10                 br.close();  // 'br' is closed
11          }
12          // ...
13     }
14 }
```

References

- IBM developerWorks: Java theory and practice: Good housekeeping practices.

Ensure that an output resource is closed on completion

Category: Critical > Resource Leaks

Description: A resource that is opened for writing but not closed may cause a resource leak.

A subclass of `Writer` or `OutputStream` that is opened for writing but not properly closed later may cause a resource leak.

Recommendation

Ensure that the resource is always closed to avoid a resource leak. Note that, because of exceptions, it is safest to close a resource properly in a `finally` block. (However, this is unnecessary for subclasses of `StringWriter` and `ByteArrayOutputStream`.)

Example

In the following example, the resource `bw` is opened but not closed.

```
1 public class CloseWriter {
2     public static void main(String[] args) throws IOException {
3         BufferedWriter bw = new BufferedWriter(new FileWriter("C:\\test.txt"));
4         bw.write("Hello world!");
5         // ...
6     }
7 }
```

In the following example, the resource `bw` is opened in a `try` block and later closed in a `finally` block.

```
1 public class CloseWriterFix {
2     public static void main(String[] args) throws IOException {
3         BufferedWriter bw = null;
4         try {
5             bw = new BufferedWriter(new FileWriter("C:\\test.txt"));
6             bw.write("Hello world!");
7         }
8         finally {
9             if(bw != null)
10                bw.close();  // 'bw' is closed
11         }
12         // ...
13     }
14 }
```

References

- IBM developerWorks: Java theory and practice: Good housekeeping practices.

Strings

- Avoid appending an array to a string without converting it to a string
- Avoid calling the default implementation of 'toString'
- Avoid printing an array without converting it to a string

Avoid appending an array to a string without converting it to a string

Category: Critical > Strings

Description: Appending an array to a string, without first converting the array to a string, produces unreadable results.

Appending an array to a `String` is likely to produce unintended results. That is, the result does not contain the contents of the array. This is because the array is implicitly converted to a `String` using `Object.toString`, which just returns the following value:

> *`getClass().getName() + '@' + Integer.toHexString(hashCode())`*

Recommendation

When converting an array to a readable string, use `Arrays.toString` for one-dimensional arrays, or `Arrays.deepToString` for multi-dimensional arrays. These functions iterate over the contents of the array and produce human-readable output.

Example

In the following example, the contents of the array `words` are printed out only if `Arrays.toString` is called on the array first. Similarly, the contents of the multi-dimensional array `wordMatrix` are printed out only if `Arrays.deepToString` is called on the array first.

```
1  public static void main(String args[]) {
2      String[] words = {"Who", "is", "John", "Galt"};
3      String[][] wordMatrix = {{"There", "is"}, {"no", "spoon"}};
4
5      // BAD: This implicitly uses 'Object.toString' to convert the contents
6      // of 'words[]', and prints out something similar to:
7      // Words: [Ljava.lang.String;@459189e1
8      System.out.println("Words: " + words);
9
10      // GOOD: 'Arrays.toString' calls 'toString' on
11      // each of the array's elements. The statement prints out:
12      // Words: [Who, is, John, Galt]
13      System.out.println("Words: " + Arrays.toString(words));
14
15      // ALMOST RIGHT: This calls 'toString' on each of the multi-dimensional
16      // array's elements. However, because the elements are arrays, the statement
17      // prints out something similar to:
18      // Word matrix: [[Ljava.lang.String;@55f33675, [Ljava.lang.String;@527c6768]]
19      System.out.println("Word matrix: " + Arrays.toString(wordMatrix));
20
21      // GOOD: This properly prints out the contents of the multi-dimensional array:
22      // Word matrix: [[There, is], [no, spoon]]
23      System.out.println("Word matrix: " + Arrays.deepToString(wordMatrix));
24  }
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java 6 API Specification: Arrays.toString(), Arrays.deepToString(), Object.toString().

Avoid calling the default implementation of 'toString'

Category: Critical > Strings

Description: Calling the default implementation of 'toString' returns a value that is unlikely to be what you expect.

In most cases, calling the default implementation of `toString` in `java.lang.Object` is not what is intended when a string representation of an object is required. The output of the default `toString` method consists of the class name of the object as well as the object's hashcode, which is usually not what was intended.

This rule includes explicit and implicit calls to `toString` that resolve to `java.lang.Object.toString`, particularly calls that are used in print or log statements.

Recommendation

For objects that are printed, define a `toString` method for the object that returns a human-readable string.

Example

The following example shows that printing an object makes an implicit call to `toString`. Because the class `WrongPerson` does not have a `toString` method, `Object.toString` is called instead, which returns the class name and the `wp` object's hashcode.

```
1  // This class does not have a 'toString' method, so 'java.lang.Object.toString'
2  // is used when the class is converted to a string.
3  class WrongPerson {
4      private String name;
5      private Date birthDate;
6
7      public WrongPerson(String name, Date birthDate) {
8          this.name =name;
9          this.birthDate = birthDate;
10      }
11 }
12
13 public static void main(String args[]) throws Exception {
14     DateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-dd");
15     WrongPerson wp = new WrongPerson("Robert Van Winkle", dateFormatter.parse("1967-10-31"));
16
17     // BAD: The following statement implicitly calls 'Object.toString',
18     // which returns something similar to:
19     // WrongPerson@4383f74d
20     System.out.println(wp);
21 }
```

In contrast, in the following modification of the example, the class `Person` does have a `toString` method, which returns a string containing the arguments that were passed when the object `p` was created.

```
1  // This class does have a 'toString' method, which is used when the object is
2  // converted to a string.
3  class Person {
4      private String name;
5      private Date birthDate;
6
7      public String toString() {
8          DateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-dd");
9          return "(Name: " + name + ", Birthdate: " + dateFormatter.format(birthDate) + ")";
10      }
11
```

```
12      public Person(String name, Date birthDate) {
13          this.name =name;
14          this.birthDate = birthDate;
15      }
16  }
17
18  public static void main(String args[]) throws Exception {
19      DateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-dd");
20      Person p = new Person("Eric Arthur Blair", dateFormatter.parse("1903-06-25"));
21
22      // GOOD: The following statement implicitly calls 'Person.toString',
23      // which correctly returns a human-readable string:
24      // (Name: Eric Arthur Blair, Birthdate: 1903-06-25)
25      System.out.println(p);
26  }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 10. Addison-Wesley, 2008.
- Java 6 API Specification: Object.toString().
- Java Language Specification, 3rd ed: 5.4 String Conversion.

Avoid printing an array without converting it to a string

Category: Critical > Strings

Description: Directly printing an array, without first converting the array to a string, produces unreadable results.

Printing an array is likely to produce unintended results. That is, the result does not contain the contents of the array. This is because the array is implicitly converted to a string using `Object.toString`, which just returns the following value:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Recommendation

When converting an array to a readable string, use `Arrays.toString` for one-dimensional arrays, or `Arrays.deepToString` for multi-dimensional arrays. These functions iterate over the contents of the array and produce human-readable output.

Example

In the following example, the contents of the array `words` can be printed out only if `Arrays.toString` is called on the array first. Similarly, the contents of the multi-dimensional array `wordMatrix` can be printed out only if `Arrays.deepToString` is called on the array first.

```
1  public static void main(String args[]) {
2      String[] words = {"Who", "is", "John", "Galt"};
3      String[][] wordMatrix = {{"There", "is"}, {"no", "spoon"}};
4
5      // BAD: This implicitly uses 'Object.toString' to convert the contents
6      // of 'words[]', and prints out something similar to:
7      // [Ljava.lang.String;@459189e1
8      System.out.println(words);
9
10      // GOOD: 'Arrays.toString' calls 'toString' on
11      // each of the array's elements. The statement prints out:
12      // [Who, is, John, Galt]
13      System.out.println(Arrays.toString(words));
14
15      // ALMOST RIGHT: This calls 'toString' on each of the multi-dimensional
16      // array's elements. However, because the elements are arrays, the statement
17      // prints out something similar to:
18      // [[Ljava.lang.String;@55f33675, [Ljava.lang.String;@527c6768]]
19      System.out.println(Arrays.toString(wordMatrix));
20
21      // GOOD: This properly prints out the contents of the multi-dimensional array:
22      // [[There, is], [no, spoon]]
23      System.out.println(Arrays.deepToString(wordMatrix));
24  }
```

References

- Java 6 API Documentation: Arrays.toString(), Arrays.deepToString(), Object.toString().

Types

- Avoid boxed types

Avoid boxed types

Category: Critical > Types

Description: Implicit boxing or unboxing of primitive types, such as 'int' and 'double', may cause confusion and subtle performance problems.

For each primitive type, such as `int` or `double`, there is a corresponding *boxed* reference type, such as `Integer` or `Double`. These boxed versions differ from their primitive equivalents because they can hold an undefined `null` element in addition to numeric (or other) values, and there can be more than one instance of a boxed type representing the same value.

In Java 5 and later, automated boxing and unboxing conversions have been added to the language. Although these automated conversions reduce the verbosity of the code, they can hide potential problems. Such problems include performance issues because of unnecessary object creation, and confusion of boxed types with their primitive equivalents.

Recommendation

Generally, you should use primitive types (boolean, byte, char, short, int, long, float, double) in preference to boxed types (Boolean, Byte, Character, Short, Integer, Long, Float, Double), whenever there is a choice. Exceptions are when a primitive value is used in collections and other parameterized types, or when a `null` value is explicitly used to represent an undefined value.

Where they cannot be avoided, perform boxing and unboxing conversions explicitly to avoid possible confusion of boxed types and their primitive equivalents. In cases where boxing conversions cause performance issues, use primitive types instead.

Example

In the following example, declaring the variable `sum` to have boxed type `Long` causes it to be unboxed and reboxed during execution of the statement inside the loop.

```
1 Long sum = 0L;
2 for (long k = 0; k < Integer.MAX_VALUE; k++) {
3     sum += k;  // AVOID: Inefficient unboxing and reboxing of 'sum'
4 }
```

To avoid this inefficiency, declare `sum` to have primitive type `long` instead.

References

- J. Bloch, *Effective Java (second edition)*, Item 49. Addison-Wesley, 2008.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Language Specification: 5.1.7 Boxing Conversion.
- Java SE Documentation: Autoboxing.

Important

Rules in this category should be followed and violations of these rules should be corrected where practical.

Rule types:

- Arithmetic (1)
- Complexity
- Concurrency (1)
- Coupling
- Declarations (2)
- Duplicate Code
- Encapsulation (1)
- Equality (1)
- Exceptions (1)
- Expressions (1)
- Extensibility (1)
- Incomplete Code (1)
- Inefficient Code
- Java objects (2)
- JUnit
- Logic Errors (1)
- Magic Constants
- Naming (2)
- Random (1)
- Result Checking
- Size
- Spring
- Strings (1)
- Swing
- Types (2)
- Useless Code

Arithmetic (1)

- Avoid checking the sign of the result of a bitwise operation
- Avoid confusion when multiplying a remainder by an integer
- Do not check parity by comparing to a positive number

Avoid checking the sign of the result of a bitwise operation

Category: Important > Arithmetic (1)

Description: Checking the sign of the result of a bitwise operation may yield unexpected results.

Checking whether the result of a bitwise operation is greater than zero may yield unexpected results.

Recommendation

It is more robust to check whether the result of the bitwise operation is *non-zero*.

Example

In the following example, the expression assigned to variable `bad` is *not* a robust way to check that the `n`th bit of `x` is set. With the given values of `x` (all bits are set) and `n`, the expression `x & (1<<n)` has the value `-2147483648`, and the variable `bad` is assigned `false`, even though the 31st bit of `x` is, in fact, set.

```
1 int x = -1;
2 int n = 31;
3
4 boolean bad = (x & (1<<n)) > 0;
```

In the following example, the expression assigned to variable `good` is a robust way to check that the `n`th bit of `x` is set. With the given values of `x` and `n`, the variable `good` is assigned `true`.

```
1 int x = -1;
2 int n = 31;
3
4 boolean good = (x & (1<<n)) != 0;
```

References

- The Java Language Specification: Integer Bitwise Operators &, ^, and |.

Avoid confusion when multiplying a remainder by an integer

Category: Important > Arithmetic (1)

Description: Using the remainder operator with the multiplication operator without adding parentheses to clarify precedence may cause confusion.

Using the remainder operator `%` with the multiplication operator may not give you the result that you expect unless you use parentheses. This is because the remainder operator has the same precedence as the multiplication operator, and the operators are left-associative.

Recommendation

When you use the remainder operator with the multiplication operator, ensure that the expression is evaluated as you expect. If necessary, add parentheses.

Example

Consider a time in milliseconds, represented by `t`. To calculate the number of milliseconds remaining after the time has been converted to whole minutes, you might write `t % 60 * 1000`. However, this is equal to `(t % 60) * 1000`, which gives the wrong result. Instead, the expression should be `t % (60 * 1000)`.

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 35. Addison-Wesley, 2005.
- The Java Tutorials: Operators.

Do not check parity by comparing to a positive number

Category: Important > Arithmetic (1)

Description: Code that uses 'x % 2 == 1' or 'x % 2 > 0' to check whether a number is odd does not work for negative numbers.

Avoid using `x % 2 == 1` or `x % 2 > 0` to check whether a number `x` is odd, or `x % 2 != 1` to check whether it is even. Such code does not work for negative numbers. For example, `-5 % 2` equals `-1`, not `1`.

Recommendation

Consider using `x % 2 != 0` to check for odd and `x % 2 == 0` to check for even.

Example

-9 is an odd number but this example does not detect it as one. This is because `-9 % 2` is -1, not 1.

```
1 class CheckOdd {
2     private static boolean isOdd(int x) {
3         return x % 2 == 1;
4     }
5 
6     public static void main(String[] args) {
7         System.out.println(isOdd(-9)); // prints false
8     }
9 }
```

It would be better to check if the number is even and then invert that check.

```
1 class CheckOdd {
2     private static boolean isOdd(int x) {
3         return x % 2 != 0;
4     }
5 
6     public static void main(String[] args) {
7         System.out.println(isOdd(-9)); // prints true
8     }
9 }
```

References

- J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases*, Puzzle 1. Addison-Wesley, 2005.
- The Java Language Specification: Remainder Operator %.

Complexity

- Avoid creating classes that have a high response
- Avoid creating methods that call many other methods
- Avoid creating methods that have a high cyclomatic complexity

Avoid creating classes that have a high response

Category: Important > Complexity

Description: A class whose methods or constructors can call many unique methods or constructors may be difficult to maintain. The number of unique methods that are called should be less than 350.

Response is the number of unique methods (or constructors) that can be called by all the methods (or constructors) of a class. For example, if a class has two methods (X and Y), and one method calls methods A and B, and the other method calls methods A and C, the class's response is 3 (methods A, B, and C are called).

Classes that have a high response can be difficult to understand and test. This is because you have to read through all the methods that can possibly be called to fully understand the class.

Recommendation

Generally, when a class has a high response, it is because it contains methods that individually make large numbers of calls or because it has high efferent coupling. The solution is therefore to fix these underlying problems, and the class's response decreases accordingly.

References

- S. R. Chidamber and C. F. Kemerer, *A metrics suite for object-oriented design*. IEEE Transactions on Software Engineering, 20(6):476-493, 1994.

Avoid creating methods that call many other methods

Category: Important > Complexity

Description: A method or constructor that calls many other methods may be difficult to maintain. The number of other methods that are called should be less than 100.

If the number of calls that is made by a method (or constructor) to other methods is high, the method can be difficult to understand, because you have to read through all the methods that it calls to fully understand what it does. There are various reasons why a method may make a high number of calls, including:

- The method is simply too large in general.
- The method has too many responsibilities (see [Martin]).
- The method spends all of its time delegating rather than doing any work itself.

Recommendation

The appropriate action depends on the reason why the method makes a high number of calls:

- If the method is too large, you should refactor it into multiple smaller methods, using the 'Extract Method' refactoring from [Fowler], for example.
- If the method is taking on too many responsibilities, a new layer of methods can be introduced below the top-level method, each of which can do some of the original work. The top-level method then only needs to delegate to a much smaller number of methods, which themselves delegate to the methods lower down.
- If the method spends all of its time delegating, some of the work that is done by the subsidiary methods can be moved into the top-level method, and the subsidiary methods can be removed. This is the refactoring called 'Inline Method' in [Fowler].

References

- M. Fowler, *Refactoring*. Addison-Wesley, 1999.
- R. Martin, The Single Responsibility Principle. Published online.

Avoid creating methods that have a high cyclomatic complexity

Category: Important > Complexity

Description: A high number of possible execution paths through a method or constructor may make it difficult to understand and test. The number of execution paths should be less than 40.

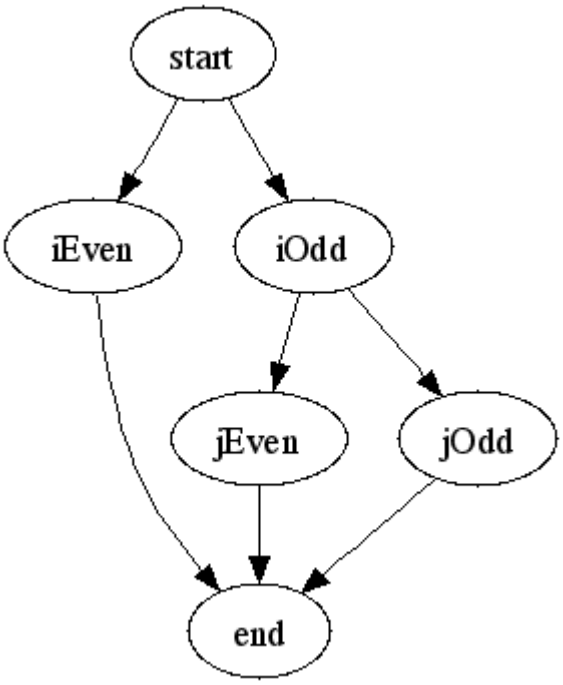
The cyclomatic complexity of a method (or constructor) is the number of possible linearly-independent execution paths through that method (see [Wikipedia]). It was originally introduced as a complexity measure by Thomas McCabe [McCabe].

A method with high cyclomatic complexity is typically difficult to understand and test.

Example

```
1  int f(int i, int j) {
2      int result;
3      if(i % 2 == 0) {
4          result = i + j;
5      }
6      else {
7          if(j % 2 == 0) {
8              result = i * j;
9          }
10          else {
11              result = i - j;
12          }
13      }
14      return result;
15  }
```

The control flow graph for this method is as follows:



As you can see from the graph, the number of linearly-independent execution paths through the method is 3. Therefore, the cyclomatic complexity is 3.

Recommendation

Simplify methods that have a high cyclomatic complexity. For example, tidy up complex logic, and/or split methods into multiple smaller methods using the 'Extract Method' refactoring from [Fowler].

References

- M. Fowler, *Refactoring*. Addison-Wesley, 1999.
- T. J. McCabe, *A Complexity Measure*. IEEE Transactions on Software Engineering, SE-2(4), December 1976.
- Wikipedia: Cyclomatic complexity.

Concurrency (1)

- API Misuse (1)

API Misuse (1)

- Do not directly call 'run'

Do not directly call 'run'

Category: Important > Concurrency (1) > API Misuse (1)

Description: Directly calling a 'Thread' object's 'run' method does not start a separate thread but executes the method within the current thread.

A direct call of a `Thread` object's `run` method does not start a separate thread. The method is executed within the current thread. This is an unusual use because `Thread.run()` is normally intended to be called from within a separate thread.

Recommendation

To execute `Runnable.run` from within a separate thread, do one of the following:

- Construct a `Thread` object using the `Runnable` object, and call `start` on the `Thread` object.
- Define a subclass of a `Thread` object, and override the definition of its `run` method. Then construct an instance of this subclass and call `start` on that instance directly.

Example

In the following example, the main thread, `ThreadDemo`, calls the child thread, `NewThread`, using `run`. This causes the child thread to run to completion before the rest of the main thread is executed, so that "Child thread activity" is printed before "Main thread activity".

```
1  public class ThreadDemo {
2      public static void main(String args[]) {
3          NewThread runnable = new NewThread();
4
5          runnable.run();    // Call to 'run' does not start a separate thread
6
7          System.out.println("Main thread activity.");
8      }
9  }
10
11  class NewThread extends Thread {
12      public void run() {
13          try {
14              Thread.sleep(10000);
15          }
16          catch (InterruptedException e) {
17              System.out.println("Child interrupted.");
18          }
19          System.out.println("Child thread activity.");
20      }
21  }
```

To enable the two threads to run concurrently, create the child thread and call `start`, as shown below. This causes the main thread to continue while the child thread is waiting, so that "Main thread activity" is printed before "Child thread activity".

```
1  public class ThreadDemo {
2      public static void main(String args[]) {
3          NewThread runnable = new NewThread();
4
5          runnable.start();                                    // Call 'start' method
6
7          System.out.println("Main thread activity.");
8      }
9  }
```

References

- The Java Tutorials: Defining and Starting a Thread.

Coupling

- Avoid creating classes that depend on many other types
- Avoid feature envy from a method to a class
- Avoid hub classes
- Avoid inappropriate intimacy between classes

Avoid creating classes that depend on many other types

Category: Important > Coupling

Description: A class that depends on many other types is quite brittle. The number of dependencies on other types should be less than 30.

Efferent coupling is the number of outgoing dependencies for each class. In other words, it is the number of other types on which each class depends.

A class that depends on many other types is quite brittle, because if any of its dependencies change, the class itself may have to change as well. Furthermore, the reason for the high number of dependencies is often that different parts of the class depend on different groups of other types, so it is common to find that classes with high efferent coupling also lack cohesion.

Recommendation

You can reduce efferent coupling by splitting up a class so that each part depends on fewer types.

Example

In the following example, class X depends on both Y and Z.

```
1  class X {
2      public void iUseY(Y y) {
3          y.doStuff();
4      }
5
6      public Y soDoY() {
7          return new Y();
8      }
9
10      public Z iUseZ(Z z1, Z z2) {
11          return z1.combine(z2);
12      }
13  }
```

However, the methods that use Y do not use Z, and the methods that use Z do not use Y. Therefore, the class can be split into two classes, one of which depends only on Y and the other only on Z

```
1  class YX {
2      public void iUseY(Y y) {
3          y.doStuff();
4      }
5
6      public Y soDoY() {
7          return new Y();
8      }
9  }
10
11  class ZX {
12      public Z iUseZ(Z z1, Z z2) {
13          return z1.combine(z2);
14      }
15  }
```

Although this is a slightly artificial example, this sort of situation does tend to occur in more complicated classes, so the general technique is quite widely applicable.

References

- IBM developerWorks: Evolutionary architecture and emergent design: Emergent design through metrics.
- R. Martin, *Agile Software Development: Principles, Patterns and Practices*. Pearson, 2011.

Avoid feature envy from a method to a class

Category: Important > Coupling

Description: A method that uses more methods or variables from another (unrelated) class than from its own class violates the principle of putting data and behavior in the same place.

Feature envy refers to situations where a method is "in the wrong place", because it does not use many methods or variables of its own class, but uses a whole range of methods or variables from some other class. This violates the principle of putting data and behavior in the same place, and exposes internals of the other class to the method.

Recommendation

For each method that may be exhibiting feature envy, see if it needs to be declared in its present location, or if it can be moved to the class it is "envious" of. A common example is a method that calls a large number of getters on another class to perform some calculation that does not rely on anything from its own class. In such cases, the method should be moved to the class containing the data. If the calculation depends on some values from the method's current class, they can either be passed as arguments or accessed using getters from the other class.

If it is inappropriate to move the entire method, see if all the dependencies on the other class are concentrated in just one part of the method. If so, they can be moved into a method of their own. This method can then be moved to the other class and called from the original method.

If a class is envious of functionality defined in a superclass, perhaps the superclass needs to be re-written to become more extensible and allow its subtypes to define new behavior without them depending so deeply on the superclass's implementation. The *template method* pattern may be useful in achieving this.

Modern IDEs provide several refactorings that may be useful in addressing instances of feature envy, typically under the names of "Move method" and "Extract method".

Occasionally behavior can be misinterpreted as feature envy when in fact it is justified. The most common examples are complex design patterns like *visitor* or *strategy*, where the goal is to separate data from behavior.

Example

In the following example, initially the method `getTotalPrice` is in the `Basket` class, but it only uses data belonging to the `Item` class. Therefore, it represents an instance of feature envy. To refactor it, `getTotalPrice` can be moved to `Item` and its parameter can be removed. The resulting code is easier to understand and keep consistent.

```
1  // Before refactoring:
2  class Item { .. }
3  class Basket {
4      // ..
5      float getTotalPrice(Item i) {
6          float price = i.getPrice() + i.getTax();
7          if (i.isOnSale())
8              price = price - i.getSaleDiscount() * price;
9          return price;
10     }
11 }
12
13 // After refactoring:
14 class Item {
15     // ..
16     float getTotalPrice() {
17         float price = getPrice() + getTax();
```

```
18          if (isOnSale())
19              price = price - getSaleDiscount() * price;
20          return price;
21      }
22  }
```

The refactored code is still appropriate, even if some data from the `Basket` class is necessary for the computation of the total price. For example, if the `Basket` class applies a bulk discount when a sufficient number of items are in the basket, an "additional discount" parameter can be added to `Item.getTotalPrice(..)`. Alternatively, the application of the discount can be performed in a method in `Basket` that calls `Item.getTotalPrice`.

References

- E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.
- W. C. Wake, *Refactoring Workbook*, pp. 93–94. Addison-Wesley Professional, 2004.

Avoid hub classes

Category: Important > Coupling

Description: Hub classes, which are classes that use, and are used by, many other classes, are complex and difficult to change without affecting the rest of the system.

A *hub class* is a class that depends on many other classes, and on which many other classes depend.

For the purposes of this rule, a *dependency* is any use of one class in another. Examples include:

- Using another class as the declared type of a variable or field
- Using another class as an argument type for a method
- Using another class as a superclass in the `extends` declaration
- Calling a method defined in the class

A class can be regarded as a hub class when both the incoming dependencies and the outgoing source dependencies are particularly high. (Outgoing source dependencies are dependencies on other source classes, rather than library classes like `java.lang.Object`.)

It is undesirable to have many hub classes because they are extremely difficult to maintain. This is because many other classes depend on a hub class, and so the other classes have to be tested and possibly adapted after each change to the hub class. Also, when one of a hub class's direct dependencies changes, the behavior of the hub class and all of its dependencies has to be checked and possibly adapted.

Recommendation

One common reason for a class to be regarded as a hub class is that it tries to do too much, including unrelated functionality that depends on different parts of the code base. If possible, split such classes into several better encapsulated classes.

Another common reason is that the class is a "struct-like" class that has many fields of different types. Introducing some intermediate grouping containers to make it clearer what fields belong together may be a good option.

References

- E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.
- W. C. Wake, *Refactoring Workbook*. Addison-Wesley Professional, 2004.

Avoid inappropriate intimacy between classes

Category: Important > Coupling

Description: Two otherwise-unrelated classes that share too much information about each other are difficult to maintain, change and understand.

Inappropriate intimacy is an anti-pattern that describes a pair of otherwise-unrelated classes that are too tightly coupled: each class uses a significant number of methods and fields of the other. This makes both classes difficult to maintain, change and understand. Inappropriate intimacy is the same as the "feature envy" anti-pattern but in both directions: each class is "envious" of some functionality or data defined in the other class.

Recommendation

The solution might be as simple as moving some misplaced methods to their rightful place, or perhaps some tangled bits of code need to be extracted to their own methods first before being moved.

Sometimes the entangled parts (both fields and methods) indicate a missing object or level of abstraction. It might make sense to combine them into a new type that can be used in both classes. Perhaps delegation needs to be introduced to hide some implementation details.

It may be necessary to convert the bidirectional association into a unidirectional relationship, possibly by using dependency inversion.

Modern IDEs provide refactoring support for this sort of issue, usually with the names "Move method", "Extract method" or "Extract class".

References

- E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.
- W. C. Wake, *Refactoring Workbook*, pp. 95–96. Addison-Wesley Professional, 2004.

Declarations (2)

- Avoid assignment to parameters in a method or constructor
- Avoid using the same name for a field and a variable

Avoid assignment to parameters in a method or constructor

Category: Important > Declarations (2)

Description: Changing a parameter's value in a method or constructor may decrease code readability.

Programmers usually assume that the value of a parameter is the value that was passed in to the method or constructor. Assigning a different value to a parameter in a method or constructor invalidates that assumption.

Recommendation

Avoid assignment to parameters by doing one of the following:

- Introduce a local variable and assign to that instead.
- Use an expression directly rather than assigning it to a parameter.

Example

In the following example, the first method shows assignment to the parameter `miles`. The second method shows how to avoid this by using the expression `miles * KM_PER_MILE`. The third method shows how to avoid the assignment by declaring a local variable `kilometres` and assigning to that.

```
1  final private static double KM_PER_MILE = 1.609344;
2
3  // AVOID: Example that assigns to a parameter
4  public double milesToKM(double miles) {
5      miles *= KM_PER_MILE;
6      return miles;
7  }
8
9  // GOOD: Example of using an expression instead
10  public double milesToKM(double miles) {
11      return miles * KM_PER_MILE;
12  }
13
14  // GOOD: Example of using a local variable
15  public double milesToKM(double miles) {
16      double kilometres = miles * KM_PER_MILE;
17      return kilometres;
18  }
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Basics: Methods 4 - Local variables.

Avoid using the same name for a field and a variable

Category: Important > Declarations (2)

Description: A method in which a variable is declared with the same name as a field is difficult to understand.

If a method declares a local variable with the same name as a field, then it is very easy to mix up the two when reading or modifying the program.

Recommendation

Consider using different names for the field and local variable to make the difference between them clear.

Example

The following example shows a local variable `values` that has the same name as a field.

```
1  public class Container
2  {
3      private int[] values; // Field called 'values'
4
5      public Container (int... values) {
6          this.values = values;
7      }
8
9      public Container dup() {
10          int length = values.length;
11          int[] values = new int[length];  // Local variable called 'values'
12          Container result = new Container(values);
13          return result;
14      }
15 }
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Language Specification: 6.4 Shadowing and Obscuring.

Duplicate Code

- Avoid duplicate anonymous classes
- Avoid duplicate methods
- Avoid mostly duplicate classes
- Avoid mostly duplicate files
- Avoid mostly duplicate methods
- Avoid mostly similar files

Avoid duplicate anonymous classes

Category: Important > Duplicate Code

Description: Duplicated anonymous classes indicate that refactoring is necessary.

Anonymous classes are a common way of creating implementations of an interface or abstract class whose functionality is really only needed once. Duplicating the definition of an anonymous class in several places is usually a sign that refactoring is necessary.

Code duplication in general is highly undesirable for a range of reasons. The artificially inflated amount of code is more difficult to understand, and sequences of similar but subtly different lines can mask the real purpose or intention behind them. Also, there is always a risk that only one of several copies of the code is updated to address a defect or add a feature.

Recommendation

Introduce a concrete class that contains the definition just once, and replace the anonymous classes with instances of this concrete class.

Example

In the following example, the definition of the class `addActionListener` is duplicated for each button that needs to use it. A better solution is shown that defines just one class, `MultiplexingListener`, which is used by each button.

```
1  // BAD: Duplicate anonymous classes:
2  button1.addActionListener(new ActionListener() {
3      public void actionPerfored(ActionEvent e)
4      {
5          for (ActionListener listener: listeners)
6              listeners.actionPerformed(e);
7      }
8  });
9
10 button2.addActionListener(new ActionListener() {
11     public void actionPerfored(ActionEvent e)
12     {
13         for (ActionListener listener: listeners)
14             listeners.actionPerformed(e);
15     }
16 });
17
18 // ... and so on.
19
20 // GOOD: Better solution:
21 class MultiplexingListener implements ActionListener {
22     public void actionPerformed(ActionEvent e) {
23         for (ActionListener listener : listeners)
24             listener.actionPerformed(e);
25     }
26 }
27
28 button1.addActionListener(new MultiplexingListener());
29 button2.addActionListener(new MultiplexingListener());
30 // ... and so on.
```

References

- E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner. *Do code clones matter?* Proceedings of the 31st International Conference on Software Engineering, 485-495, 2009.

Avoid duplicate methods

Category: Important > Duplicate Code

Description: Duplicated methods make code more difficult to understand and introduce a risk of changes being made to only one copy.

A method should never be duplicated exactly in several places in the code. The severity of this problem is higher for longer methods than for extremely short methods of one or two statements, but there are usually better ways of achieving the same effect.

Code duplication in general is highly undesirable for a range of reasons. The artificially inflated amount of code is more difficult to understand, and sequences of similar but subtly different lines can mask the real purpose or intention behind them. Also, there is always a risk that only one of several copies of the code is updated to address a defect or add a feature.

Recommendation

At its simplest, the duplication can be addressed by simply removing all but one of the duplicate method definitions, and changing calls to the removed methods so that they call the remaining function instead.

This may not be possible because of visibility or accessibility. A common example is where two classes implement the same functionality but neither is a subtype of the other, so it is not possible to inherit a single method definition. In such cases, introducing a common superclass to share the duplicated code is a possible option. Alternatively, if the methods do not need access to private object state, they can be moved to a shared utility class that just provides the functionality itself.

Example

In the following example, `RowWidget` and `ColumnWidget` contain duplicate methods. The `collectChildren` method should probably be moved into the superclass, `Widget`, and shared between `RowWidget` and `ColumnWidget`.

```
1  class RowWidget extends Widget {
2      // ...
3      public void collectChildren(Set<Widget> result) {
4          for (Widget child : this.children) {
5              if (child.isVisible()) {
6                  result.add(children);
7                  child.collectChildren(result);
8              }
9          }
10     }
11 }
12
13 class ColumnWidget extends Widget {
14     // ...
15     public void collectChildren(Set<Widget> result) {
16         for (Widget child : this.children) {
17             if (child.isVisible()) {
18                 result.add(children);
19                 child.collectChildren(result);
20             }
21         }
22     }
23 }
```

Alternatively, if not all kinds of `Widget` actually need `collectChildren` (for example, not all of them have children), it might be necessary to introduce a new, possibly abstract, class under `Widget`. For example, the new class might

be called `ContainerWidget` and include a single definition of `collectChildren`. Both `RowWidget` and `ColumnWidget` could extend the class and inherit `collectChildren`.

Modern IDEs may provide refactoring support for this sort of issue, usually with the names "Pull up" or "Extract supertype".

References

- E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner. *Do code clones matter?* Proceedings of the 31st International Conference on Software Engineering, 485-495, 2009.

Avoid mostly duplicate classes

Category: Important > Duplicate Code

Description: Classes in which most of the methods are duplicated in another class make code more difficult to understand and introduce a risk of changes being made to only one copy.

When most of the methods in one class are duplicated in one or more other classes, the classes themselves are regarded as *mostly duplicate*.

Code duplication in general is highly undesirable for a range of reasons. The artificially inflated amount of code is more difficult to understand, and sequences of similar but subtly different lines can mask the real purpose or intention behind them. Also, there is always a risk that only one of several copies of the code is updated to address a defect or add a feature.

Recommendation

Although completely duplicated classes are rare, they are usually a sign of a simple oversight (or deliberate copy/paste) by a developer. Usually the required solution is to remove all but one of them.

It is more common to see duplication of many methods between two classes, leaving just a few that are actually different. Decide whether the differences are intended or the result of an inconsistent update to one of the copies:

- If the two classes serve different purposes but many of their methods are duplicated, this indicates that there is a missing level of abstraction. Introducing a common super-class to define the common methods is likely to prevent many problems in the long term. Modern IDEs may provide refactoring support for this sort of issue, usually with the names "Pull up" or "Extract supertype".
- If the two classes serve the same purpose and are different only as a result of inconsistent updates then treat the classes as completely duplicate. Determine the most up-to-date and correct version of the code and eliminate all near duplicates.

References

- E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner. *Do code clones matter?* Proceedings of the 31st International Conference on Software Engineering, 485-495, 2009.

Avoid mostly duplicate files

Category: Important > Duplicate Code

Description: Files in which most of the lines are duplicated in another file make code more difficult to understand and introduce a risk of changes being made to only one copy.

When most of the lines in one file are duplicated in one or more other files, the files themselves are regarded as *mostly duplicate*.

Code duplication in general is highly undesirable for a range of reasons. The artificially inflated amount of code is more difficult to understand, and sequences of similar but subtly different lines can mask the real purpose or intention behind them. Also, there is always a risk that only one of several copies of the code is updated to address a defect or add a feature.

Recommendation

Although completely duplicated files are rare, they are usually a sign of a simple oversight (or deliberate copy/paste) by a developer. Usually the required solution is to remove all but one of them. A common exception is generated code that simply occurs in several places in the source tree.

It is more common to see duplication of many lines between two files, leaving just a few that are actually different. Decide whether the differences are intended or the result of an inconsistent update to one of the copies:

- If the two files serve different purposes but many of their lines are duplicated, this indicates that there is a missing level of abstraction. Look for ways to share the functionality, either by extracting a utility class for parts of it or by encapsulating the common parts into a new super class of any classes involved.
- If the two files serve the same purpose and are different only as a result of inconsistent updates then treat the files as completely duplicate. Determine the most up-to-date and correct version of the code and eliminate all near duplicates.

References

- E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner. *Do code clones matter?* Proceedings of the 31st International Conference on Software Engineering, 485-495, 2009.

Avoid mostly duplicate methods

Category: Important > Duplicate Code

Description: Methods in which most of the lines are duplicated in another method make code more difficult to understand and introduce a risk of changes being made to only one copy.

When most of the lines in one method are duplicated in one or more other methods, the methods themselves are regarded as *mostly duplicate* or *similar*.

Code duplication in general is highly undesirable for a range of reasons. The artificially inflated amount of code is more difficult to understand, and sequences of similar but subtly different lines can mask the real purpose or intention behind them. Also, there is always a risk that only one of several copies of the code is updated to address a defect or add a feature.

Recommendation

Although completely duplicated methods are rare, they are usually a sign of a simple oversight (or deliberate copy/paste) by a developer. Usually the required solution is to remove all but one of them.

It is more common to see duplication of many lines between two methods, leaving just a few that are actually different. Decide whether the differences are intended or the result of an inconsistent update to one of the copies.

- If the two methods serve different purposes but many of their lines are duplicated, this indicates that there is a missing level of abstraction. Look for ways of encapsulating the commonality and sharing it while retaining the differences in functionality. Perhaps the method can be moved to a single place and given an additional parameter, allowing it to cover all use cases. Alternatively, there may be a common pre-processing or post-processing step that can be extracted to its own (shared) method, leaving only the specific parts in the existing methods. Modern IDEs may provide refactoring support for this sort of issue, usually with the names "Extract method", "Change method signature", "Pull up" or "Extract supertype".
- If the two methods serve the same purpose and are different only as a result of inconsistent updates then treat the methods as completely duplicate. Determine the most up-to-date and correct version of the code and eliminate all near duplicates. Callers of the removed methods should be updated to call the remaining method instead.

References

- E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner. *Do code clones matter?* Proceedings of the 31st International Conference on Software Engineering, 485-495, 2009.

Avoid mostly similar files

Category: Important > Duplicate Code

Description: Files in which most of the lines are similar to those in another file make code more difficult to understand and introduce a risk of changes being made to only one copy.

When most of the lines in one file have corresponding "similar" lines in one or more other files, the files themselves are regarded as *mostly similar*. Two lines are defined as similar if they are either identical or contain only very minor differences.

Code duplication in general is highly undesirable for a range of reasons. The artificially inflated amount of code is more difficult to understand, and sequences of similar but subtly different lines can mask the real purpose or intention behind them. Also, there is always a risk that only one of several copies of the code is updated to address a defect or add a feature.

Recommendation

Consider whether the differences are deliberate or a result of an inconsistent update to one of the clones. If the latter, then treating the files as completely duplicate and eliminating all but one (while preserving any corrections or new features that may have been introduced) is the best course. If two files serve genuinely different purposes but almost all of their lines are the same, that can be a sign that there is a missing level of abstraction. Can some of the shared code be extracted into methods (perhaps with additional parameters, to cover the differences in behavior)? Should it be moved into a utility class or file that is accessible to all current implementations, or should a new level of abstraction be introduced?

References

- E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner. *Do code clones matter?* Proceedings of the 31st International Conference on Software Engineering, 485-495, 2009.

Encapsulation (1)

- Avoid creating classes that lack cohesion
- Avoid creating subclasses that have a high specialization index
- Avoid exposing an object's internal representation

Avoid creating classes that lack cohesion

Category: Important > Encapsulation (1)

Description: A class that lacks cohesion probably has multiple responsibilities. The lack of cohesion measure (LCOM) should be less than 3000.

A cohesive class is one in which most methods access the same fields. A class that lacks cohesion is usually one that has multiple responsibilities.

Various measures of lack of cohesion have been proposed. The Chidamber and Kemerer version of lack of cohesion inspects pairs of methods. If there are many pairs that access the same data, the class is cohesive. If there are many pairs that do not access any common data, the class is not cohesive. More precisely, if:

- `n1` is the number of pairs of distinct methods in a class that *do not have* at least one commonly-accessed field, and
- `n2` is the number of pairs of distinct methods in a class that *do have* at least one commonly-accessed field,

the lack of cohesion measure (LCOM) can be defined as:

LCOM = max((n1 - n2) / 2, 0)

High values of LCOM indicate a significant lack of cohesion. As a rough indication, an LCOM of 500 or more may give you cause for concern.

Recommendation

Classes generally lack cohesion because they have more responsibilities than they should (see [Martin]). In general, the solution is to identify each of the different responsibilities that the class has, and split them into multiple classes, using the 'Extract Class' refactoring from [Fowler], for example.

References

- S. R. Chidamber and C. F. Kemerer, *A metrics suite for object-oriented design*. IEEE Transactions on Software Engineering, 20(6):476-493, 1994.
- M. Fowler, *Refactoring*, pp. 65, 122-5. Addison-Wesley, 1999.
- R. Martin, The Single Responsibility Principle. Published online.
- O. de Moor et al, *Keynote Address: .QL for Source Code Analysis*. Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation, 2007.

Avoid creating subclasses that have a high specialization index

Category: Important > Encapsulation (1)

Description: A class that overrides much of the behavior of its ancestor classes indicates that the abstractions in the ancestor classes should be reviewed. The specialization index should be less than 4.

Specialization index is the extent to which a subclass overrides the behavior of its ancestor classes. It is computed as follows:

1. Determine the number of overridden methods in the subclass (not counting overrides of abstract methods).
2. Multiply this number by the subclass's depth in the inheritance hierarchy.
3. Divide the result by the subclass's total number of methods.

If a class overrides many of the methods of its ancestor classes, it indicates that the abstractions in the ancestor classes should be reviewed. This is particularly true for subclasses that are lower down in the inheritance hierarchy. In general, subclasses should *add* behavior to their superclasses, rather than *redefining* the behavior that is already there.

Recommendation

The most common reason that classes have a high specialization index is that multiple subclasses specialize a common base class in the same way. In this case, the relevant method(s) should be pulled up into the base class (see the 'Pull Up Method' refactoring in [Fowler]).

Example

In the following example, duplicating `getName` in each of the subclasses is unnecessary.

```
1  abstract class Animal {
2      protected String animalName;
3
4      public Animal(String name) {
5          animalName = name;
6      }
7
8      public String getName(){
9          return animalName;
10      }
11      public abstract String getKind();
12  }
13
14  class Dog extends Animal {
15      public Dog(String name) {
16          super(name);
17      }
18
19      public String getName() {  // This method duplicates the method in class 'Cat'.
20          return animalName + " the " +  getKind();
21      }
22
23      public String getKind() {
24          return "dog";
25      }
26   }
27
28  class Cat extends Animal {
29      public Cat(String name) {
30          super(name);
```

```
31      }
32
33      public String getName() {  // This method duplicates the method in class 'Dog'.
34          return animalName + " the " +  getKind();
35      }
36
37      public String getKind() {
38          return "cat";
39      }
40  }
```

To decrease the specialization index of the subclasses, pull up `getName` into the base class.

```
1  abstract class Animal {
2      private String animalName;
3
4      public Animal(String name) {
5          animalName = name;
6      }
7
8      public String getName() {  // Method has been pulled up into the base class.
9          return animalName + " the " +  getKind();
10      }
11
12      public abstract String getKind();
13 }
14
15 class Dog extends Animal {
16     public Dog(String name) {
17         super(name);
18     }
19
20     public String getKind() {
21         return "dog";
22     }
23 }
24
25 class Cat extends Animal {
26     public Cat(String name) {
27         super(name);
28     }
29
30     public String getKind() {
31         return "cat";
32     }
33 }
```

References

- M. Fowler, *Refactoring*, pp. 260-3. Addison-Wesley, 1999.
- M. Lorenz and J. Kidd, *Object-oriented Software Metrics*. Prentice Hall, 1994.
- O. de Moor et al, *Keynote Address: .QL for Source Code Analysis*. Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation, 2007.

Avoid exposing an object's internal representation

Category: Important > Encapsulation (1)

Description: An object that accidentally exposes its internal representation may allow the object's fields to be modified in ways that the object is not prepared to handle.

A subtle type of defect is caused when an object accidentally exposes its internal representation to the code outside the object, and the internal representation is then (deliberately or accidentally) modified in ways that the object is not prepared to handle. Most commonly, this happens when a getter returns a direct reference to a mutable field within the object, or a setter just assigns a mutable argument to its field.

Recommendation

There are three ways of addressing this problem:

- **Using immutable objects** : The fields store objects that are *immutable*, which means that once constructed their value can never be changed. Examples from the standard library are `String`, `Integer` or `Float`. Although such an object may be aliased, or shared between several contexts, there can be no unexpected changes to the internal state of the object because it cannot be modified.
- **Creating a read-only view** : The `java.util.Collections.unmodifiable*` methods can be used to create a read-only view of a collection without copying it. This tends to give better performance than creating copies of objects. Note that this technique is not suitable for every situation, because any changes to the underlying collection will spread to affect the view. This can lead to unexpected results, and is a particular danger when writing multi-threaded code.
- **Making defensive copies** : Each setter (or constructor) makes a copy or clone of the incoming parameter. In this way, it constructs an instance known only internally, and no matter what happens with the object that was passed in, the state stays consistent. Conversely, each getter for a field must also construct a copy of the field's value to return.

Example

In the following example, the private field `items` is returned directly by the getter `getItems`. Thus, a caller obtains a reference to internal object state and can manipulate the collection of items in the cart. In the example, each of the carts is emptied when `countItems` is called.

```
1  public class Cart {
2      private Set<Item> items;
3      // ...
4      // AVOID: Exposes representation
5      public Set<Item> getItems() {
6          return items;
7      }
8  }
9  ....
10  int countItems(Set<Cart> carts) {
11      int result = 0;
12      for (Cart cart : carts) {
13          Set<Item> items = cart.getItems();
14          result += items.size();
15          items.clear(); // AVOID: Changes internal representation
16      }
17      return result;
18  }
```

The solution is for `getItems` to return a *copy* of the actual field, for example `return new HashSet<Item>(items);`.

References

- J. Bloch, *Effective Java (second edition)*, Items 15 and 39. Addison-Wesley, 2008.
- Java 7 API Documentation: Collections.

Equality (1)

- Avoid unintentionally overloading 'Comparable.compareTo'
- Redefine 'equals' in subclasses that have additional fields

Avoid unintentionally overloading 'Comparable.compareTo'

Category: Important > Equality (1)

Description: Defining 'Comparable.compareTo', where the parameter of 'compareTo' is not of the appropriate type, overloads 'compareTo' instead of overriding it.

Classes that implement `Comparable<T>` and define a `compareTo` method whose parameter type is not `T` *overload* the `compareTo` method instead of *overriding* it. This may not be intended.

Example

In the following example, the call to `compareTo` on line 17 calls the method defined in class `Super`, instead of the method defined in class `Sub`, because the type of `a` and `b` is `Super`. This may not be the method that the programmer intended.

```
1  public class CovariantCompareTo {
2      static class Super implements Comparable<Super> {
3          public int compareTo(Super rhs) {
4              return -1;
5          }
6      }
7
8      static class Sub extends Super {
9          public int compareTo(Sub rhs) {  // Definition of compareTo uses a different parameter type
10             return 0;
11         }
12     }
13
14     public static void main(String[] args) {
15         Super a = new Sub();
16         Super b = new Sub();
17         System.out.println(a.compareTo(b));
18     }
19 }
```

Recommendation

To *override* the `Comparable<T>.compareTo` method, the parameter of `compareTo` must have type `T`.

In the example above, this means that the type of the parameter of `Sub.compareTo` should be changed to `Super`.

References

- J. Bloch, *Effective Java (second edition)*, Item 12. Addison-Wesley, 2008.
- The Java Language Specification: Overriding (by Instance Methods), Overloading.
- The Java Tutorials: Overriding and Hiding Methods.

Redefine 'equals' in subclasses that have additional fields

Category: Important > Equality (1)

Description: If a class overrides 'Object.equals', and a subclass defines additional fields to those it inherits but does not re-define 'equals', the results of 'equals' may be wrong.

If a class overrides the default implementation of equality defined by the `Object.equals` method, and a subclass of that class declares additional fields to the ones that it inherits, the results of `equals` may be wrong, unless that subclass also redefines `equals`.

Recommendation

See if the subclass should provide its own implementation of `equals` to take into account the additional fields that it declares.

Example

In the following example, rectangles `r1` and `r2` are calculated to be equal, even though they have different dimensions. This is because the class `Rectangle` does not override `Square.equals`, so it uses a test for equality that is only applicable to squares, not rectangles. (Note that, in practice, the example should also include an implementation of `hashCode`.)

```
1  public class DefineEqualsWhenAddingFields {
2      static class Square {
3          protected int width = 0;
4          public Square(int width) {
5              this.width = width;
6          }
7          @Override
8          public boolean equals(Object thatO) {  // This method works only for squares.
9              if(thatO != null && getClass() == thatO.getClass() ) {
10                 Square that = (Square)thatO;
11                 return width == that.width;
12             }
13             return false;
14         }
15     }
16
17     static class Rectangle extends Square {
18         private int height = 0;
19         public Rectangle(int width, int height) {
20             super(width);
21             this.height = height;
22         }
23     }
24
25     public static void main(String[] args) {
26         Rectangle r1 = new Rectangle(4, 3);
27         Rectangle r2 = new Rectangle(4, 5);
28         System.out.println(r1.equals(r2));  // Outputs 'true'
29     }
30 }
```

To get the correct result, you must override `Square.equals` in class `Rectangle`.

References

- Java API Documentation: Object.equals().

Exceptions (1)

- Avoid dereferencing a variable that may be 'null'
- Avoid unreachable 'catch' clauses
- Do not drop an exception

Avoid dereferencing a variable that may be 'null'

Category: Important > Exceptions (1)

Description: Dereferencing a variable whose value may be 'null' may cause a 'NullPointerException'.

If a variable is dereferenced, and the variable may have a `null` value on some execution paths leading to the dereferencing, the dereferencing may result in a `NullPointerException`.

Recommendation

Ensure that the variable does not have a `null` value when it is dereferenced.

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Avoid unreachable 'catch' clauses

Category: Important > Exceptions (1)

Description: An unreachable 'catch' clause may indicate a mistake in exception handling or may be unnecessary.

An unreachable `catch` clause may indicate a logical mistake in the exception handling code or may simply be unnecessary.

Although certain unreachable `catch` clauses cause a compiler error, there are also unreachable `catch` clauses that do not cause a compiler error. A `catch` clause `C` is considered reachable by the compiler if both of the following conditions are true:

- A checked exception that is thrown in the `try` block is assignable to the parameter of `C`.
- There is no previous `catch` clause whose parameter type is equal to, or a supertype of, the parameter type of `C`.

However, a `catch` clause that is considered reachable by the compiler can be unreachable if both of the following conditions are true:

- The `catch` clause's parameter type `E` does not include any unchecked exceptions.
- All exceptions that are thrown in the `try` block whose type is a (strict) subtype of `E` are already handled by previous `catch` clauses.

Recommendation

Ensure that unreachable `catch` clauses are removed or that further corrections are made to make them reachable.

Note that if a `try-catch` statement contains multiple `catch` clauses, and an exception that is thrown in the `try` block matches more than one of the `catch` clauses, only the first matching clause is executed.

Example

In the following example, the second `catch` clause is unreachable, and can be removed.

```
1  FileInputStream fis = null;
2  try {
3      fis = new FileInputStream(new File("may_not_exist.txt"));
4      // read from input stream
5  } catch (FileNotFoundException e) {
6      // ask the user and try again
7  } catch (IOException e) {
8      // more serious, abort
9  } finally {
10      if (fis!=null) { try { fis.close(); } catch (IOException e) { /*ignore*/ } }
11 }
```

References

- The Java Language Specification: Execution of try-catch, Unreachable Statements.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Do not drop an exception

Category: Important > Exceptions (1)

Description: Dropping an exception may allow an unusual program state to continue without recovery.

You should not drop an exception, because it indicates that an unusual program state has been reached. This usually requires corrective actions to be performed to recover from the exceptional state and try to resume normal program operation.

Recommendation

You should do one of the following:

- Catch and handle the exception.
- Throw the exception to the outermost level of nesting.

Note that usually you should catch and handle a checked exception, but you can throw an unchecked exception to the outermost level.

There is occasionally a valid reason for ignoring an exception. In such cases, you should document the reason to improve the readability of the code. Alternatively, you can implement a static method with an empty body to handle these exceptions. Instead of dropping the exception altogether, you can then pass it to the static method with a string explaining the reason for ignoring it.

Examples

The following example shows a dropped exception.

```
// Dropped exception, with no information on whether
// the exception is expected or not
synchronized void waitIfAutoSyncScheduled() {
    try {
        while (isAutoSyncScheduled) {
            this.wait(1000);
        }
    } catch (InterruptedException e) {
    }
}
```

The following example shows how you can improve code readability by defining a new utility method.

```
// 'ignore' method. This method does nothing, but can be called
// to document the reason why the exception can be ignored.
public static void ignore(Throwable e, String message) {

}
```

The following example shows the exception being passed to `ignore` with a comment.

```
// Exception is passed to 'ignore' method with a comment
synchronized void waitIfAutoSyncScheduled() {
    try {
        while (isAutoSyncScheduled) {
            this.wait(1000);
        }
    } catch (InterruptedException e) {
        Exceptions.ignore(e, "Expected exception. The file cannot be synchronized yet.");
    }
}
```

References

- J. Bloch, *Effective Java (second edition)*, Item 65. Addison-Wesley, 2008.
- The Java Tutorials: Unchecked Exceptions - The Controversy.

Expressions (1)

- Avoid assignments in Boolean expressions
- Avoid very complex conditions

Avoid assignments in Boolean expressions

Category: Important > Expressions (1)

Description: Assignments in Boolean conditions can be confused with equality tests and make the condition more difficult to understand.

The assignment operator (=) can easily be confused with the equality operator (==), and can make a Boolean expression more difficult to understand. Consequently, assignments in Boolean expressions should be avoided.

Some useful idioms are an exception to this rule, such as checking that some bytes have been read from an input-stream, as shown in the `readConfiguration` method in the example below. More precisely, an assignment is allowed in a Boolean expression if the result of the assignment is compared to another value.

Recommendation

Consider structuring the condition so that the side-effects are moved outside of the condition, possibly splitting the condition into several separate tests.

Example

In the following example, consider the rather confusing assignment to `restart` in the `notify` method. The assignment should be performed outside of the condition instead.

```
1  public class ScreenView
2  {
3      private static int BUF_SIZE = 1024;
4      private Screen screen;
5
6      public void notify(Change change) {
7          boolean restart = false;
8          if (change.equals(Change.MOVE)
9              || v.equals(Change.REPAINT)
10             || (restart = v.equals(Change.RESTART))  // AVOID: Confusing assignment in condition
11             || v.equals(Change.FLIP))
12         {
13             if (restart)
14                 WindowManager.restart();
15             screen.update();
16         }
17     }
18
19     // ...
20
21     public void readConfiguration(InputStream config) {
22         byte[] buf = new byte[BUF_SIZE];
23         int read;
24         while ((read = config.read(buf)) > 0) {  // OK: Assignment whose result is compared to
25                                                  // another value
26             // ...
27         }
28         // ...
29     }
30 }
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Language Specification: 15.21 Equality Operators, 15.26 Assignment Operators.

Avoid very complex conditions

Category: Important > Expressions (1)

Description: Very complex conditions are difficult to read and may include defects.

In general, very complex conditions are difficult to write and read, and increase the chance of defects.

Recommendation

Firstly, a condition can often be simplified by changing other parts of the code to initialize variables more consistently. For example, is there a semantic difference between `id` being `null` and having zero-length? If not, choosing one sentinel value and using it consistently simplifies most uses of that variable.

Secondly, extracting part of a condition into a Boolean-valued method can simplify the condition and also allow code reuse, with all its benefits.

Thirdly, assigning each subcondition of the condition to a local variable, and then using the variables in the condition instead can simplify the condition.

Example

The following example shows a complex condition found in a real program used by millions of people. The condition is so confusing that even the programmer who wrote it is not sure if he got it right (see the `TODO` comment).

```
1  public class Dialog
2  {
3      // ...
4
5      private void validate() {
6          // TODO: check that this covers all cases
7          if ((id != null && id.length() == 0) ||
8              ((partner == null || partner.id == -1) &&
9              ((option == Options.SHORT && parameter.length() == 0) ||
10              (option == Options.LONG && parameter.length() < 8))))
11          {
12              disableOKButton();
13          } else {
14              enableOKButton();
15          }
16      }
17
18      // ...
19 }
```

The condition can be simplified by extracting parts of the condition into Boolean-valued methods. These methods are then used in the condition.

```
1  public class Dialog
2  {
3      // ...
4
5      private void validate() {
6        if(idIsEmpty() || (noPartnerId() && parameterLengthInvalid())){ // GOOD: Condition is simpler
7          disableOKButton();
8        } else {
9          enableOKButton();
10       }
11      }
```

```
12
13      private boolean idIsEmpty(){
14        return id != null && id.length() == 0;
15      }
16
17      private boolean noPartnerId(){
18        return partner == null || partner.id == -1;
19      }
20
21      private boolean parameterLengthInvalid(){
22        return (option == Options.SHORT && parameter.length() == 0) ||
23               (option == Options.LONG && parameter.length() < 8);
24      }
25
26      // ...
27  }
```

References

- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, §17.G28. Prentice Hall, 2008.
- S. McConnell, *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004.

Extensibility (1)

- Avoid writing to a static field from an instance method

Avoid writing to a static field from an instance method

Category: Important > Extensibility (1)

Description: Writing to a static field from an instance method is prone to race conditions unless you use synchronization. In addition, it makes it difficult to keep the static state consistent and affects code readability.

A static field represents state shared between all instances of a particular class. Typically, static methods are provided to manipulate this static state, and it is bad practice to modify the static state of a class from an instance method (or from a constructor).

There are several reasons why this is bad practice. It can be very difficult to keep the static state consistent when there are multiple instances through which it could be modified. Such modifications represent a readability issue: most programmers would expect a static method to affect static state, and an instance method to affect instance state.

Recommendation

If the field should be an instance field, ensure that it does not have a `static` modifier.

If the field does have to be static, evaluate the assumptions in the code. Does the field really have to be modified directly in an instance method? It might be better to access the field from within static methods, so that any concerns about synchronization can be addressed without numerous synchronization statements in the code. Perhaps the field modification is part of the static initialization of the class, and should be moved to a static initializer or method.

Example

In the following example, a static field, `customers`, is written to by an instance method, `initialize`. It is entirely reasonable for another developer to assume that an instance method called `initialize` should be called on each new instance, and that is what the code in `Department` does. Unfortunately, the static field is shared between all instances of `Customer`, and so each time `initialize` is called, the old state is lost.

```
1  public class Customer {
2      private static List<Customer> customers;
3      public void initialize() {
4          // AVOID: Static field is written to by instance method.
5          customers = new ArrayList<Customer>();
6          register();
7      }
8      public static void add(Customer c) {
9          customers.add(c);
10      }
11 }
12
13 // ...
14 public class Department {
15     public void addCustomer(String name) {
16         Customer c = new Customer(n);
17         // The following call overwrites the list of customers
18         // stored in 'Customer' (see above).
19         c.initialize();
20         Customer.add(c);
21     }
22 }
```

The solution is to extract the static initialization of `customers` to a static method, where it will happen exactly once.

References

- Java Language Specification: 8.3.1.1 static Fields.

Incomplete Code (1)

- Do not include empty 'finalize' methods
- Ensure that 'TODO' or 'FIXME' comments are resolved
- Ensure that 'ZipOutputStream.write' is called when writing a ZIP file

Do not include empty 'finalize' methods

Category: Important > Incomplete Code (1)

Description: An empty 'finalize' method is useless and prevents its superclass's 'finalize' method (if any) from being called.

An empty `finalize` method is useless and may prevent finalization from working properly. This is because, unlike a constructor, a finalizer does not implicitly call the finalizer of the superclass. Thus, an empty finalizer prevents any finalization logic that is defined in any of its superclasses from being executed.

Recommendation

Do not include an empty `finalize` method.

Example

In the following example, the empty `finalize` method in class `ExtendedLog` prevents the `finalize` method in class `Log` from being called. The result is that the log file is not closed. To fix this, remove the empty `finalize` method.

```
1  class ExtendedLog extends Log
2  {
3      // ...
4
5      protected void finalize() {
6          // BAD: This empty 'finalize' stops 'super.finalize' from being executed.
7      }
8  }
9
10 class Log implements Closeable
11 {
12     // ...
13
14     public void close() {
15         // ...
16     }
17
18     protected void finalize() {
19         close();
20     }
21 }
```

References

- Java Language Specification: 12.6 Finalization of Class Instances.

Ensure that 'TODO' or 'FIXME' comments are resolved

Category: Important > Incomplete Code (1)

Description: A comment that contains 'TODO' or 'FIXME' may indicate code that is incomplete or broken, or highlight an ambiguity in the software's specification.

A comment that includes the word `TODO` or `FIXME` often marks a part of the code that is incomplete or broken, or highlights ambiguities in the software's specification.

For example, this list of comments is typical of those found in real programs:

- `TODO: move this code somewhere else`
- `FIXME: handle this case`
- `FIXME: find a better solution to this workaround`
- `TODO: test this`

Recommendation

It is very important that `TODO` or `FIXME` comments are not just removed from the code. Each of them must be addressed in some way.

Simpler comments can usually be immediately addressed by fixing the code, adding a test, doing some refactoring, or clarifying the intended behavior of a feature.

In contrast, larger issues may require discussion, and a significant amount of work to address. In these cases it is a good idea to move the comment to an issue-tracking system, so that the issue can be tracked and prioritized relative to other defects and feature requests.

References

- Approxion: TODO or not TODO.
- Wikipedia: Comment tags, Issue tracking system.

Ensure that 'ZipOutputStream.write' is called when writing a ZIP file

Category: Important > Incomplete Code (1)

Description: Omitting a call to 'ZipOutputStream.write' when writing a ZIP file to an output stream means that an empty ZIP file entry is written.

The `ZipOutputStream` class is used to write ZIP files to a file or other stream. A ZIP file consists of a number of *entries*. Usually each entry corresponds to a file in the directory structure being zipped. There is a method on `ZipOutputStream` that is slightly confusingly named `putNextEntry`. Despite its name, it does not write a whole entry. Instead, it writes the *metadata* for an entry. The content for that entry is then written using the `write` method. Finally the entry is closed using `closeEntry`.

Therefore, if you call `putNextEntry` and `closeEntry` but omit the call to `write`, an empty ZIP file entry is written to the output stream.

Recommendation

Ensure that you include a call to `ZipOutputStream.write`.

Example

In the following example, the `archive` method calls `putNextEntry` and `closeEntry` but the call to `write` is left out.

```
1  class Archive implements Closeable
2  {
3      private ZipOutputStream zipStream;
4
5      public Archive(File zip) throws IOException {
6          OutputStream stream = new FileOutputStream(zip);
7          stream = new BufferedOutputStream(stream);
8          zipStream = new ZipOutputStream(stream);
9      }
10
11      public void archive(String name, byte[] content) throws IOException {
12          ZipEntry entry = new ZipEntry(name);
13          zipStream.putNextEntry(entry);
14          // Missing call to 'write'
15          zipStream.closeEntry();
16      }
17
18      public void close() throws IOException {
19          zipStream.close();
20      }
21 }
```

References

- Java 2 Platform Standard Edition 5.0, API Specification: ZipOutputStream.

Inefficient Code

- Avoid calling 'Collection.toArray' with a zero-length array argument
- Avoid calling a boxed type's constructor directly
- Avoid checking a string for equality with an empty string
- Avoid iterating through a map using its key set
- Avoid non-static nested classes unless necessary
- Avoid performing string concatenation in a loop
- Avoid using the 'String(String)' constructor

Avoid calling 'Collection.toArray' with a zero-length array argument

Category: Important > Inefficient Code

Description: Calling 'Collection.toArray' with a zero-length array argument is inefficient.

The `java.util.Collection` interface provides a `toArray` method that can be used to convert a collection of objects into an array of a particular type. This method takes an array as an argument, which is used for two purposes. Firstly, it determines the type of the returned array. Secondly, if it is big enough to hold all values in the collection, it is filled with those values and returned; otherwise, a new array of sufficient size and the appropriate type is allocated and filled.

It is common to pass a fresh zero-length array to `toArray`, simply because it is easy to construct one. Unfortunately, this allocation is wasteful, because the array clearly is not big enough to hold the elements of the collection. This can cause considerable garbage collector churn, impacting performance.

Recommendation

It is always best to call `toArray` with a new array allocated with a sufficient size to hold the contents of the collection. Usually, this involves calling the collection's `size` method and allocating an array with that many elements. While it may seem odd that adding a call to `size` improves performance, if you do not pass a large enough array, the `toArray` method makes this call automatically. Calling `size` explicitly and then calling `toArray` with a large enough array avoids the possible creation of two arrays (one too small and consequently unused).

Example

In the following example, the first version of class `Company` uses an inefficient call to `toArray` by passing a zero-length array. The second version uses a more efficient call that passes an array that is big enough to store the customer list.

```
1  class Company {
2      private List<Customer> customers = ...;
3
4      public Customer[] getCustomerArray() {
5          // AVOID: Inefficient call to 'toArray' with zero-length argument
6          return customers.toArray(new Customer[0]);
7      }
8  }
9
10 class Company {
11     private List<Customer> customers = ...;
12
13     public Customer[] getCustomerArray() {
14         // GOOD: More efficient call to 'toArray' with argument that is big enough to store the list
15         return customers.toArray(new Customer[customers.size()]);
16     }
17 }
```

References

- Java Platform, Standard Edition 6, API Specification: toArray.

Avoid calling a boxed type's constructor directly

Category: Important > Inefficient Code

Description: Calling the constructor of a boxed type is inefficient.

Primitive values (for example `int`, `float`, `boolean`) all have corresponding reference types known as *boxed types* (for example `Integer`, `Float`, `Boolean`). These boxed types can be used when an actual object is required. While they all provide constructors that take a primitive value of the appropriate type, it is usually considered bad practice to call those constructors directly.

Each boxed type provides a static `valueOf` method that takes an argument of the appropriate primitive type and returns an object representing it. The advantage of calling `valueOf` over calling a constructor is that it allows for some caching of instances. By reusing these cached instances instead of constructing new heap objects all the time, a significant amount of garbage collector effort can be saved.

Recommendation

In almost all circumstances, a call of, for example, `Integer.valueOf(42)` can be used instead of `new Integer(42)`.

Note that sometimes you can rely on Java's *autoboxing* feature, which implicitly calls `valueOf`. For details, see the example.

Example

The following example shows the three ways of creating a new integer. In the autoboxing example, the zero is autoboxed to an `Integer` because the constructor `Account` takes an argument of this type.

```
1  public class Account {
2      private Integer balance;
3      public Account(Integer startingBalance) {
4          this.balance = startingBalance;
5      }
6  }
7
8  public class BankManager {
9      public void openAccount(Customer c) {
10          ...
11          // AVOID: Inefficient primitive constructor
12          accounts.add(new Account(new Integer(0)));
13          // GOOD: Use 'valueOf'
14          accounts.add(new Account(Integer.valueOf(0)));
15          // GOOD: Rely on autoboxing
16          accounts.add(new Account(0));
17      }
18  }
```

References

- J. Bloch, *Effective Java (second edition)*, Items 1 and 5. Addison-Wesley, 2008.
- Java Platform, Standard Edition 6, API Documentation: Boolean.valueOf(), Byte.valueOf(), Short.valueOf(), Integer.valueOf(), Long.valueOf(), Float.valueOf(), Double.valueOf().

Avoid checking a string for equality with an empty string

Category: Important > Inefficient Code

Description: Checking a string for equality with an empty string is inefficient.

When checking whether a string `s` is empty, perhaps the most obvious solution is to write something like `s.equals("")` (or `"".equals(s)`). However, this actually carries a fairly significant overhead, because `String.equals` performs a number of type tests and conversions before starting to compare the content of the strings.

Recommendation

The preferred way of checking whether a string `s` is empty is to check if its length is equal to zero. Thus, the condition is `s.length() == 0`. The `length` method is implemented as a simple field access, and so should be noticeably faster than calling `equals`.

Note that in Java 6 and later, the `String` class has an `isEmpty` method that checks whether a string is empty. If the codebase does not need to support Java 5, it may be better to use that method instead.

Example

In the following example, class `InefficientDBClient` uses `equals` to test whether the strings `user` and `pw` are empty. Note that the test `"".equals(pw)` guards against `NullPointerException`, but the test `user.equals("")` throws a `NullPointerException` if `user` is `null`.

In contrast, the class `EfficientDBClient` uses `length` instead of `equals`. The class preserves the behavior of `InefficientDBClient` by guarding `pw.length() == 0` but not `user.length() == 0` with an explicit test for `null`. Whether or not this guard is desirable depends on the intended behavior of the program.

```
1  // Inefficient version
2  class InefficientDBClient {
3      public void connect(String user, String pw) {
4          if (user.equals("") || "".equals(pw))
5              throw new RuntimeException();
6          ...
7      }
8  }
9
10 // More efficient version
11 class EfficientDBClient {
12     public void connect(String user, String pw) {
13         if (user.length() == 0 || (pw != null && pw.length() == 0))
14             throw new RuntimeException();
15         ...
16     }
17 }
```

References

- Java Platform, Standard Edition 6, API Specification: String.length(), String.equals().

Avoid iterating through a map using its key set

Category: Important > Inefficient Code

Description: Iterating through the values of a map using the key set is inefficient.

Java's Collections Framework provides several different ways of iterating the contents of a map. You can retrieve the set of keys, the collection of values, or the set of "entries" (which are, in effect, key/value pairs).

The choice of iterator can affect performance. For example, it is considered bad practice to iterate the key set of a map if the body of the loop performs a map lookup on each retrieved key anyway.

Recommendation

Evaluate the requirements of the loop body. If it does not actually need the key apart from looking it up in the map, iterate the map's values (obtained by a call to `values`) instead. If the loop genuinely needs both key and value for each mapping in the map, iterate the entry set (obtained by a call to `entrySet`) and retrieve the key and value from each entry. This saves a more expensive map lookup each time.

Example

In the following example, the first version of the method `findId` iterates the map `people` using the key set. This is inefficient because the body of the loop needs to access the value for each key. In contrast, the second version iterates the map using the entry set because the loop body needs both the key and the value for each mapping.

```
1  // AVOID: Iterate the map using the key set.
2  class AddressBook {
3      private Map<String, Person> people = ...;
4      public String findId(String first, String last) {
5          for (String id : people.keySet()) {
6              Person p = people.get(id);
7              if (first.equals(p.firstName()) && last.equals(p.lastName()))
8                  return id;
9          }
10          return null;
11     }
12 }
13
14 // GOOD: Iterate the map using the entry set.
15 class AddressBook {
16     private Map<String, Person> people = ...;
17     public String findId(String first, String last) {
18         for (Entry<String, Person> entry: people.entrySet()) {
19             Person p = entry.getValue();
20             if (first.equals(p.firstName()) && last.equals(p.lastName()))
21                 return entry.getKey();
22         }
23         return null;
24     }
25 }
```

References

- Java Platform, Standard Edition 6, API Specification: Map.entrySet().

Avoid non-static nested classes unless necessary

Category: Important > Inefficient Code

Description: A non-static nested class keeps a reference to the enclosing object, which makes the nested class bigger and may cause a memory leak.

Nested classes allow logical grouping of related concerns, increasing encapsulation and readability. However, there is a potential disadvantage when using them that you should be aware of.

Any non-static nested class implicitly holds onto its "enclosing instance". This means that:

- The nested class has an implicitly defined field. The field holds a reference to the instance of the enclosing class that constructed the nested class.
- The nested class's constructors have an implicit parameter. The parameter is used for passing in the instance of the enclosing class. A reference to the instance is then stored in the field mentioned above.

Often, this is useful and necessary, because non-static nested class instances have access to instance state on their enclosing classes. However, if this instance state is not needed by the nested class, this makes nested class instances larger than necessary, and hidden references to the enclosing classes are often the source of subtle memory leaks.

Recommendation

When a nested class does not need the enclosing instance, it is better to declare the nested class `static`, avoiding the implicit field. As a result, instances of the nested class become smaller, and hidden references to the enclosing class are made explicit.

If a reference to the enclosing class instance is required during construction of the nested class instance (but not subsequently), the constructor of the nested class should be refactored so that it is explicitly given a reference to the enclosing instance.

References

- J. Bloch, *Effective Java (second edition)*, Item 22. Addison-Wesley, 2008.
- Java Language Specification: 8.1.3. Inner Classes and Enclosing Instances.
- The Java Tutorials: Nested Classes.

Avoid performing string concatenation in a loop

Category: Important > Inefficient Code

Description: Performing string concatenation in a loop that iterates many times may affect performance.

When string concatenation is performed using the "+" operator, the compiler translates this operation to a suitable manipulation, possibly constructing several intermediate strings. In general, because strings are immutable, at least one new string has to be constructed to hold the result.

Building up a string one piece at a time in a loop requires a new string on every iteration, repeatedly copying longer and longer prefixes to fresh string objects. As a result, performance can be severely degraded.

Recommendation

Whenever a string is constructed using a loop that iterates more than just a few times, it is usually better to create a `StringBuffer` or `StringBuilder` object and append to that. Because such buffers are based on mutable character arrays, which do not require a new string to be created for each concatenation, they can reduce the cost of repeatedly growing the string.

To choose between `StringBuffer` and `StringBuilder`, check if the new buffer object can possibly be accessed by several different threads while in use. If multi-thread safety is required, use a `StringBuffer`. For purely local string buffers, you can avoid the overhead of synchronization by using a `StringBuilder`.

Example

The following example shows a simple test that measures the time taken to construct a string. It constructs the same string of 65,536 binary digits, character-by-character, first by repeatedly appending to a string, and then by using a `StringBuilder`. The second method is three orders of magnitude faster.

```
1  public class ConcatenationInLoops {
2      public static void main(String[] args) {
3          Random r = new Random(123);
4          long start = System.currentTimeMillis();
5          String s = "";
6          for (int i = 0; i < 65536; i++)
7              s += r.nextInt(2);
8          System.out.println(System.currentTimeMillis() - start);  // This prints roughly 4500.
9
10          r = new Random(123);
11          start = System.currentTimeMillis();
12          StringBuilder sb = new StringBuilder();
13          for (int i = 0; i < 65536; i++)
14              sb.append(r.nextInt(2));
15          s = sb.toString();
16          System.out.println(System.currentTimeMillis() - start);  // This prints 5.
17      }
18  }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 51. Addison-Wesley, 2008.
- Java Platform, Standard Edition 6, API Specification: StringBuffer, StringBuilder.

Avoid using the 'String(String)' constructor

Category: Important > Inefficient Code

Description: Using the 'String(String)' constructor is less memory efficient than using the constructor argument directly.

The `String` class is immutable, which means that there is no way to change the string that it represents. Consequently, there is rarely a need to copy a `String` object or construct a new instance based on an existing string, for example by writing something like `String hello = new String("hello")`. Furthermore, this practice is not memory efficient.

Recommendation

The copied string is functionally indistinguishable from the argument that was passed into the `String` constructor, so you can simply omit the constructor call and use the argument passed into it directly. Unless an explicit copy of the argument string is needed, this is a safe transformation.

Example

The following example shows three cases of copying a string using the `String` constructor, which is inefficient. In each case, simply removing the constructor call `new String` and leaving the argument results in better code and less memory churn.

```
1  public void sayHello(String world) {
2      // AVOID: Inefficient 'String' constructor
3      String message = new String("hello ");
4
5      // AVOID: Inefficient 'String' constructor
6      message = new String(message + world);
7
8      // AVOID: Inefficient 'String' constructor
9      System.out.println(new String(message));
10 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 5. Addison-Wesley, 2008.
- Java Platform, Standard Edition 6, API Specification: String(String).

Java objects (2)

- Cloning (1)
- Garbage collection (1)
- Serialization (1)

Cloning (1)

- Ensure that a class that implements 'Cloneable' overrides 'clone'

Ensure that a class that implements 'Cloneable' overrides 'clone'

Category: Important > Java objects (2) > Cloning (1)

Description: A class that implements 'Cloneable' but does not override the 'clone' method will have undesired behavior.

A class that implements `Cloneable` should override `Object.clone`. For non-trivial objects, the `Cloneable` contract requires a deep copy of the object's internal state. A class that does not have a `clone` method indicates that the class is breaking the contract and will have undesired behavior.

The Java API documentation states that, for an object `x`, the general intent of the `clone` method is for it to satisfy the following three properties:

- `x.clone() != x` (the cloned object is a different object instance)
- `x.clone().getClass() == x.getClass()` (the cloned object is the same type as the source object)
- `x.clone().equals(x)` (the cloned object has the same 'contents' as the source object)

For the cloned object to be of the same type as the source object, non-final classes must call `super.clone` and that call must eventually reach `Object.clone`, which creates an instance of the right type. If it were to create a new object using a constructor, a subclass that does not implement the `clone` method returns an object of the wrong type. In addition, all of the class's supertypes that also override `clone` must call `super.clone`. Otherwise, it never reaches `Object.clone` and creates an object of the incorrect type.

However, as `Object.clone` only does a shallow copy of the fields of an object, any `Cloneable` objects that have a "deep structure" (for example, objects that use an array or `Collection`) must take the clone that results from the call to `super.clone` and assign explicitly created copies of the structure to the clone's fields. This means that the cloned instance does not share its internal state with the source object. If it *did* share its internal state, any changes made in the cloned object would also affect the internal state of the source object, probably causing unintended behavior.

One added complication is that `clone` cannot modify values in final fields, which would be already set by the call to `super.clone`. Some fields must be made non-final to correctly implement the `clone` method.

Recommendation

The necessity of creating a deep copy of an object's internal state means that, for most objects, `clone` must be overridden to satisfy the `Cloneable` contract. Implement a `clone` method that properly creates the internal state of the cloned object.

Notable exceptions to this recommendation are:

- Classes that contain only primitive types (which will be properly cloned by `Object.clone` as long as its `Cloneable` supertypes all call `super.clone`).
- Subclasses of `Cloneable` classes that do not introduce new state.

Example

In the following example, `WrongStack` does not implement `clone`. This means that when `ws1clone` is cloned from `ws1`, the default `clone` implementation is used. This results in operations on the `ws1clone` stack affecting the `ws1` stack.

```
1 abstract class AbstractStack implements Cloneable {
2     public AbstractStack clone() {
3         try {
4             return (AbstractStack) super.clone();
5         } catch (CloneNotSupportedException e) {
```

```
            throw new AssertionError("Should not happen");
        }
    }
}

class WrongStack extends AbstractStack {
    private static final int MAX_STACK = 10;
    int[] elements = new int[MAX_STACK];
    int top = -1;

    void push(int newInt) {
        elements[++top] = newInt;
    }
    int pop() {
        return elements[top--];
    }
    // BAD: No 'clone' method to create a copy of the elements.
    // Therefore, the default 'clone' implementation (shallow copy) is used, which
    // is equivalent to:
    //
    //  public WrongStack clone() {
    //      WrongStack cloned = (WrongStack) super.clone();
    //      cloned.elements = elements;  // Both 'this' and 'cloned' now use the same elements.
    //      return cloned;
    //  }
}

public class MissingMethodClone {
    public static void main(String[] args) {
        WrongStack ws1 = new WrongStack();              // ws1: {}
        ws1.push(1);                                    // ws1: {1}
        ws1.push(2);                                    // ws1: {1,2}
        WrongStack ws1clone = (WrongStack) ws1.clone(); // ws1clone: {1,2}
        ws1clone.pop();                                 // ws1clone: {1}
        ws1clone.push(3);                               // ws1clone: {1,3}
        System.out.println(ws1.pop());                  // Because ws1 and ws1clone have the same
                                                        // elements, this prints 3 instead of 2
    }
}
```

In the following modified example, `RightStack` *does* implement `clone`. This means that when `rs1clone` is cloned from `rs1`, operations on the `rs1clone` stack do not affect the `rs1` stack.

```
abstract class AbstractStack implements Cloneable {
    public AbstractStack clone() {
        try {
            return (AbstractStack) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError("Should not happen");
        }
    }
}

class RightStack extends AbstractStack {
    private static final int MAX_STACK = 10;
    int[] elements = new int[MAX_STACK];
    int top = -1;

    void push(int newInt) {
        elements[++top] = newInt;
    }
    int pop() {
        return elements[top--];
    }

```

```
23      // GOOD: 'clone' method to create a copy of the elements.
24      public RightStack clone() {
25          RightStack cloned = (RightStack) super.clone();
26          cloned.elements = elements.clone();  // 'cloned' has its own elements.
27          return cloned;
28      }
29  }
30
31  public class MissingMethodClone {
32      public static void main(String[] args) {
33          RightStack rs1 = new RightStack();              // rs1: {}
34          rs1.push(1);                                    // rs1: {1}
35          rs1.push(2);                                    // rs1: {1,2}
36          RightStack rs1clone = rs1.clone();              // rs1clone: {1,2}
37          rs1clone.pop();                                 // rs1clone: {1}
38          rs1clone.push(3);                               // rs1clone: {1,3}
39          System.out.println(rs1.pop());                  // Correctly prints 2
40      }
41  }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 11. Addison-Wesley, 2008.
- Java 6 API Specification: Object.clone().

Garbage collection (1)

- Do not set fields to 'null' in a finalizer
- Do not trigger garbage collection explicitly
- Ensure that a 'finalize' method calls 'super.finalize'

Do not set fields to 'null' in a finalizer

Category: Important > Java objects (2) > Garbage collection (1)

Description: Setting fields to 'null' in a finalizer does not cause the object to be collected by the garbage collector any earlier, and may adversely affect performance.

A finalizer does not need to set an object's fields to `null` to help the garbage collector. At the point in the Java object life-cycle when the `finalize` method is called, the object is no longer reachable from the garbage collection roots. Explicitly setting the object's fields to `null` does not cause the referenced objects to be collected by the garbage collector any earlier, and may even adversely affect performance.

The life-cycle of a Java object has 7 stages:

- **Created** : Memory is allocated for the object and the initializers and constructors have been run.
- **In use** : The object is reachable through a chain of strong references from a garbage collection root. A garbage collection root is a special class of variable (which includes variables on the stack of any thread, static variables of any class, and references from Java Native Interface code).
- **Invisible** : The object has already gone out of scope, but the stack frame of the method that contained the scope is still in memory. Not all objects transition into this state.
- **Unreachable** : The object is no longer reachable through a chain of strong references. It becomes a candidate for garbage collection.
- **Collected** : The garbage collector has identified that the object can be deallocated. If it has a finalizer, it is marked for finalization. Otherwise, it is deallocated.
- **Finalized** : An object with a `finalize` method transitions to this state after the finalize method is completed and the object still remains unreachable.
- **Deallocated** : The object is a candidate for deallocation.

The call to the `finalize` method occurs when the object is in the 'Collected' stage. At that point, it is already unreachable from the garbage collection roots so any of its references to other objects no longer contribute to their reference counts.

Recommendation

Ensure that the finalizer does not contain any `null` assignments because they are unlikely to help garbage collection.

If a finalizer does nothing but nullify an object's fields, it is best to completely remove the finalizer. Objects with finalizers severely affect performance, and you should avoid defining `finalize` where possible.

Example

In the following example, `finalize` unnecessarily assigns the object's fields to null.

```
1  class FinalizedClass {
2      Object o = new Object();
3      String s = "abcdefg";
4      Integer i = Integer.valueOf(2);
5
6      @Override
7      protected void finalize() throws Throwable {
8          super.finalize();
9          //No need to nullify fields
10          this.o = null;
```

```
11          this.s = null;
12          this.i = null;
13      }
14  }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 7. Addison-Wesley, 2008.
- IBM developerWorks: Explicit nulling.
- Oracle Technology Network: How to Handle Java Finalization's Memory-Retention Issues .
- S. Wilson and J. Kesselman, *Java Platform Performance: Strategies and Tactics, 1st ed.*, Appendix A. Prentice Hall, 2001.

Do not trigger garbage collection explicitly

Category: Important > Java objects (2) > Garbage collection (1)

Description: Triggering garbage collection explicitly may either have no effect or may trigger unnecessary garbage collection.

You should avoid making calls to explicit garbage collection methods (`Runtime.gc` and `System.gc`). The calls are not guaranteed to trigger garbage collection, and they may also trigger unnecessary garbage collection passes that lead to decreased performance.

Recommendation

It is better to let the Java Virtual Machine (JVM) handle garbage collection. If it becomes necessary to control how the JVM handles memory, it is better to use the JVM's memory and garbage collection options (for example, `-Xmx`, `-XX:NewRatio`, `-XX:Use*GC`) than to trigger garbage collection in the application.

The memory management classes that are used by Real-Time Java are an exception to this rule, because they are designed to handle garbage collection differently from the JVM default.

Example

The following example shows code that makes connection requests, and tries to trigger garbage collection after it has processed each request.

```
1  class RequestHandler extends Thread {
2      private boolean isRunning;
3      private Connection conn = new Connection();
4
5      public void run() {
6          while (isRunning) {
7              Request req = conn.getRequest();
8              // Process the request ...
9
10              System.gc();  // This call may cause a garbage collection after each request.
11                            // This will likely reduce the throughput of the RequestHandler
12                            // because the JVM spends time on unnecessary garbage collection passes.
13          }
14      }
15 }
```

It is better to remove the call to `System.gc` and rely on the JVM to dispose of the connection.

References

- Java 6 API Documentation: System.gc().
- Oracle Technology Network: Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning.

Ensure that a 'finalize' method calls 'super.finalize'

Category: Important > Java objects (2) > Garbage collection (1)

Description: A 'finalize' method that does not call `super.finalize` may leave cleanup actions undone.

A `finalize` method that overrides the finalizer of a superclass but does not call `super.finalize` may leave system resources undisposed of or cause other cleanup actions to be left undone.

Recommendation

Make sure that all `finalize` methods call `super.finalize` to ensure that the finalizer of its superclass is executed. Finalizer chaining is not automatic in Java.

It is also possible to defend against subclasses that do not call `super.finalize` by putting the cleanup code into a *finalizer guardian* instead of the `finalize` method. A finalizer guardian is an anonymous object instance that contains the cleanup code for the enclosing object in its `finalize` method. The only reference to the finalizer guardian is stored in a private field of the enclosing instance, which means that both the guardian and the enclosing instance can be finalized at the same time. This way, a subclass cannot block the execution of the cleanup code by not calling `super.finalize`.

Example

In the following example, `WrongCache.finalize` does not call `super.finalize`, which means that native resources are not disposed of. However, `RightCache.finalize` *does* call `super.finalize`, which means that native resources *are* disposed of.

```
1  class LocalCache {
2      private Collection<NativeResource> localResources;
3
4      //...
5
6      protected void finalize() throws Throwable {
7          for (NativeResource r : localResources) {
8              r.dispose();
9          }
10     };
11 }
12
13 class WrongCache extends LocalCache {
14     //...
15     @Override
16     protected void finalize() throws Throwable {
17         // BAD: Empty 'finalize', which does not call 'super.finalize'.
18         //        Native resources in LocalCache are not disposed of.
19     }
20 }
21
22 class RightCache extends LocalCache {
23     //...
24     @Override
25     protected void finalize() throws Throwable {
26         // GOOD: 'finalize' calls 'super.finalize'.
27         //        Native resources in LocalCache are disposed of.
28         super.finalize();
29     }
30 }
```

The following example shows a finalizer guardian.

```
class GuardedLocalCache {
    private Collection<NativeResource> localResources;
    // A finalizer guardian, which performs the finalize actions for 'GuardedLocalCache'
    // even if a subclass does not call 'super.finalize' in its 'finalize' method
    private Object finalizerGuardian = new Object() {
        protected void finalize() throws Throwable {
            for (NativeResource r : localResources) {
                r.dispose();
            }
        };
    };
}
```

References

- Java 7 API Documentation: Object.finalize().
- J. Bloch, *Effective Java (second edition)*, Item 7. Addison-Wesley, 2008.

Serialization (1)

- Do not use 'transient' in a non-serializable class
- Ensure that 'readResolve' has the correct signature
- Ensure that a class that implements 'Externalizable' has a public no-argument constructor
- Ensure that each non-transient, non-static field in a serializable class is serializable
- Ensure that the signatures of 'readObject' and 'writeObject' on a serializable class are correct

Do not use 'transient' in a non-serializable class

Category: Important > Java objects (2) > Serialization (1)

Description: Using the 'transient' field modifier in non-serializable classes has no effect.

The `transient` modifier is used to identify fields that are not part of the persistent state of the class. As such, it only has an effect if the class is serializable, and has no purpose in a non-serializable class.

A field that is marked `transient` in a non-serializable class is likely to be a leftover from a time when the class was serializable.

Recommendation

If the class is non-serializable, leave out the `transient` modifier. Otherwise, use the modifier, and ensure that the class (or a relevant supertype) implements `Serializable`.

Example

The following example shows two fields that are declared `transient`. The modifier only has an effect in the class that implements `Serializable`.

```
1  class State {
2      // The 'transient' modifier has no effect here because
3      // the 'State' class does not implement 'Serializable'.
4      private transient int[] stateData;
5  }
6
7  class PersistentState implements Serializable {
8      private int[] stateData;
9      // The 'transient' modifier indicates that this field is not part of
10      // the persistent state and should therefore not be serialized.
11     private transient int[] cachedComputedData;
12 }
```

References

- Java Language Specification, 3rd Ed: 8.3.1.3 transient Fields.
- Java 6 Object Serialization Specification: 1.5 Defining Serializable Fields for a Class.

Ensure that 'readResolve' has the correct signature

Category: Important > Java objects (2) > Serialization (1)

Description: An implementation of 'readResolve' that does not have the signature that is expected by the Java serialization framework is not recognized by the serialization mechanism.

If a class uses the `readResolve` method to specify a replacement object instance when the object is read from a stream, ensure that the signature of `readResolve` is *exactly* what the Java serialization mechanism expects.

Recommendation

Ensure that the signature of the `readResolve` method in the class matches the expected signature:

```
ANY-ACCESS-MODIFIER Object readResolve() throws ObjectStreamException;
```

Note that the method *must* return a `java.lang.Object`.

If `readResolve` is used for instance control of a serializable singleton, (that is, to make sure that deserializing a singleton class does not result in another instance of the singleton) it may be possible to use an `enum` with a single element instead. The Java serialization specification explicitly ensures that deserializing an `enum` does not create a new instance. (For details about this technique, see [Bloch].)

Example

In the following example, `FalseSingleton.readResolve` has the wrong signature, which causes deserialization to create a new instance of the singleton. However, `Singleton.readResolve` has the correct signature, which means that deserialization does not result in another instance of the singleton.

```
1  class FalseSingleton implements Serializable {
2      private static final long serialVersionUID = -7480651116825504381L;
3      private static FalseSingleton instance;
4
5      private FalseSingleton() {}
6
7      public static FalseSingleton getInstance() {
8          if (instance == null) {
9              instance = new FalseSingleton();
10         }
11         return instance;
12     }
13
14     // BAD: Signature of 'readResolve' does not match the exact signature that is expected
15     // (that is, it does not return 'java.lang.Object').
16     public FalseSingleton readResolve() throws ObjectStreamException {
17         return FalseSingleton.getInstance();
18     }
19 }
20
21 class Singleton implements Serializable {
22     private static final long serialVersionUID = -7480651116825504381L;
23     private static Singleton instance;
24
25     private Singleton() {}
26
27     public static Singleton getInstance() {
28         if (instance == null) {
29             instance = new Singleton();
30         }
31         return instance;
```

```
32      }
33
34      // GOOD: Signature of 'readResolve' matches the exact signature that is expected.
35      // It replaces the singleton that is read from a stream with an instance of 'Singleton',
36      // instead of creating a new singleton.
37      private Object readResolve() throws ObjectStreamException {
38          return Singleton.getInstance();
39      }
40  }
```

References

- Java API Documentation: Serializable.
- Java 6 Object Serialization Specification: 3.7 The readResolve Method, 1.12 Serialization of Enum Constants.
- J. Bloch, *Effective Java (second edition)*, Item 77. Addison-Wesley, 2008.

Ensure that a class that implements 'Externalizable' has a public no-argument constructor

Category: Important > Java objects (2) > Serialization (1)

Description: A class that implements 'Externalizable' but does not have a public no-argument constructor causes an 'InvalidClassException' to be thrown.

A class that implements `java.io.Externalizable` must have a public no-argument constructor. The constructor is used by the Java serialization framework when it creates the object during deserialization. If the class does not define such a constructor, the Java serialization framework throws an `InvalidClassException`.

The Java Development Kit API documentation for `Externalizable` states:

> *When an `Externalizable` object is reconstructed, an instance is created using the public no-arg constructor, then the `readExternal` method called.*

Recommendation

Make sure that externalizable classes always have a no-argument constructor.

Example

In the following example, `WrongMemo` does not declare a public no-argument constructor. When the Java serialization framework tries to deserialize the object, an `InvalidClassException` is thrown. However, `Memo` does declare a public no-argument constructor, so that the object is deserialized successfully.

```
1  class WrongMemo implements Externalizable {
2      private String memo;
3
4      // BAD: No public no-argument constructor is defined. Deserializing this object
5      // causes an 'InvalidClassException'.
6
7      public WrongMemo(String memo) {
8          this.memo = memo;
9      }
10
11      public void writeExternal(ObjectOutput arg0) throws IOException {
12          //...
13      }
14      public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
15          //...
16      }
17 }
18
19 class Memo implements Externalizable {
20      private String memo;
21
22      // GOOD: Declare a public no-argument constructor, which is used by the
23      // serialization framework when the object is deserialized.
24      public Memo() {
25      }
26
27      public Memo(String memo) {
28          this.memo = memo;
29      }
30
31      public void writeExternal(ObjectOutput out) throws IOException {
32          //...
```

```
33      }
34      public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
35          //...
36      }
37  }
```

References

- Java API Documentation: Externalizable.

Ensure that each non-transient, non-static field in a serializable class is serializable

Category: Important > Java objects (2) > Serialization (1)

Description: A non-transient field in a serializable class must also be serializable otherwise it causes the class to fail to serialize with a 'NotSerializableException'.

If a serializable class is serialized using the default Java serialization mechanism, each non-static, non-transient field in the class must also be serializable. Otherwise, the class generates a `java.io.NotSerializableException` as its fields are written out by `ObjectOutputStream.writeObject`.

As an exception, classes that define their own `readObject` and `writeObject` methods can have fields that are not themselves serializable. The `readObject` and `writeObject` methods are responsible for encoding any state in those fields that needs to be serialized.

Recommendation

To avoid causing a `NotSerializableException`, do one of the following:

- **Mark the field as `transient` :** Marking the field as `transient` makes the serialization mechanism skip the field. Before doing this, make sure that the field is not really intended to be part of the persistent state of the object.
- **Define custom `readObject` and `writeObject` methods for the `Serializable` class :** Explicitly defining the `readObject` and `writeObject` methods enables you to choose which fields to read from, or write to, an object stream during serialization.
- **Make the type of the field `Serializable` :** If the field is part of the object's persistent state and you wish to use Java's default serialization mechanism, the type of the field must implement `Serializable`. When choosing this option, make sure that you follow best practices for serialization.

Example

In the following example, `WrongPerformanceRecord` contains a field `factors` that is not serializable but is in a serializable class. This causes a `java.io.NotSerializableException` when the field is written out by `writeObject`. However, `PerformanceRecord` contains a field `factors` that is marked as `transient`, so that the serialization mechanism skips the field. This means that a correctly serialized record is output by `writeObject`.

```
1  class DerivedFactors {             // Class that contains derived values computed from entries in a
2      private Number efficiency;     // performance record
3      private Number costPerItem;
4      private Number profitPerItem;
5      ...
6  }
7
8  class WrongPerformanceRecord implements Serializable {
9      private String unitId;
10     private Number dailyThroughput;
11     private Number dailyCost;
12     private DerivedFactors factors;  // BAD: 'DerivedFactors' is not serializable
13                                      // but is in a serializable class. This
14                                      // causes a 'java.io.NotSerializableException'
15                                      // when 'WrongPerformanceRecord' is serialized.
16     ...
17 }
18
19 class PerformanceRecord implements Serializable {
20     private String unitId;
21     private Number dailyThroughput;
22     private Number dailyCost;
```

```
23      transient private DerivedFactors factors;  // GOOD: 'DerivedFactors' is declared
24                                                 // 'transient' so it does not contribute to the
25                                                 // serializable state of 'PerformanceRecord'.
26      ...
27  }
```

References

- Java API Documentation: Serializable, ObjectOutputStream.

Ensure that the signatures of 'readObject' and 'writeObject' on a serializable class are correct

Category: Important > Java objects (2) > Serialization (1)

Description: A serialized class that implements 'readObject' or 'writeObject' but does not use the correct signatures causes the default serialization mechanism to be used.

A serializable object that defines its own serialization protocol using the methods `readObject` and `writeObject` must use the signature that is expected by the Java serialization framework. Otherwise, the default serialization mechanism is used.

Recommendation

Make sure that the signatures of `readObject` and `writeObject` on serializable classes use these exact signatures:

```
1 private void readObject(java.io.ObjectInputStream in)
2      throws IOException, ClassNotFoundException;
3 private void writeObject(java.io.ObjectOutputStream out)
4      throws IOException;
```

Example

In the following example, `WrongNetRequest` defines `readObject` and `writeObject` using the wrong signatures. However, `NetRequest` defines them correctly.

```
1  class WrongNetRequest implements Serializable {
2      // BAD: Does not match the exact signature required for a custom
3      // deserialization protocol. Will not be called during deserialization.
4      void readObject(ObjectInputStream in) {
5          //...
6      }
7
8      // BAD: Does not match the exact signature required for a custom
9      // serialization protocol. Will not be called during serialization.
10      protected void writeObject(ObjectOutputStream out) {
11          //...
12      }
13 }
14
15 class NetRequest implements Serializable {
16     // GOOD: Signature for a custom deserialization implementation.
17     private void readObject(ObjectInputStream in) {
18         //...
19     }
20
21     // GOOD: Signature for a custom serialization implementation.
22     private void writeObject(ObjectOutputStream out) {
23         //...
24     }
25 }
```

References

- Java API Documentation: Serializable.
- Oracle Technology Network: Discover the secrets of the Java Serialization API.

JUnit

- Ensure that a JUnit test case class contains correctly declared test methods
- Ensure that a JUnit test method that overrides 'tearDown' calls 'super.tearDown'
- Use the correct signature for a 'suite' method in JUnit

Ensure that a JUnit test case class contains correctly declared test methods

Category: Important > JUnit

Description: A test case class whose test methods are not recognized by JUnit 3.8 as valid declarations is not used.

A JUnit 3.8 test case class (that is, a class that is a subtype of `junit.framework.TestCase`) should contain test methods, and each method must have the correct signature to be used by JUnit.

Recommendation

Ensure that the test case class contains some test methods, and that each method is of the form:

```
public void testXXX()
```

Note that the method name must start with `test` and the method must not take any parameters.

This rule applies only to JUnit 3.8-style test case classes. In JUnit 4, it is no longer required to extend `junit.framework.TestCase` to mark test methods.

Example

In the following example, `TestCaseNoTests38` does not contain any valid JUnit test methods. However, `MyTests` contains two valid JUnit test methods.

```
1  // BAD: This test case class does not have any valid JUnit 3.8 test methods.
2  public class TestCaseNoTests38 extends TestCase {
3      // This is not a test case because it does not start with 'test'.
4      public void simpleTest() {
5          //...
6      }
7
8      // This is not a test case because it takes two parameters.
9      public void testNotEquals(int i, int j) {
10         assertEquals(i != j, true);
11     }
12
13     // This is recognized as a test, but causes JUnit to fail
14     // when run because it is not public.
15     void testEquals() {
16         //...
17     }
18 }
19
20 // GOOD: This test case class correctly declares test methods.
21 public class MyTests extends TestCase {
22     public void testEquals() {
23         assertEquals(1, 1);
24     }
25     public void testNotEquals() {
26         assertFalse(1 == 2);
27     }
28 }
```

References

- JUnit: JUnit Cookbook.

Ensure that a JUnit test method that overrides 'tearDown' calls 'super.tearDown'

Category: Important > JUnit

Description: A JUnit 3.8 test method that overrides 'tearDown' but does not call 'super.tearDown' may result in subsequent tests failing, or allow the current state to persist and affect subsequent tests.

A JUnit 3.8 test method that overrides a non-empty `tearDown` method should call `super.tearDown` to make sure that the superclass performs its de-initialization routines. Not calling `tearDown` may result in test failures in subsequent tests, or allow the current state to persist and affect any following tests.

Recommendation

Call `super.tearDown` at the end of the overriding `tearDown` method.

Example

In the following example, `TearDownNoSuper.tearDown` does not call `super.tearDown`, which may cause subsequent tests to fail, or allow the internal state to be maintained. However, `TearDownSuper.tearDown` *does* call `super.tearDown`, at the end of the method, to enable `FrameworkTestCase.tearDown` to perform de-initialization.

```
1  // Abstract class that initializes then shuts down the
2  // framework after each set of tests
3  abstract class FrameworkTestCase extends TestCase {
4      @Override
5      protected void setUp() throws Exception {
6          super.setUp();
7          Framework.init();
8      }
9
10      @Override
11     protected void tearDown() throws Exception {
12         super.tearDown();
13         Framework.shutdown();
14     }
15 }
16
17 // The following classes extend 'FrameworkTestCase' to reuse the
18 // 'setUp' and 'tearDown' methods of the framework.
19
20 public class TearDownNoSuper extends FrameworkTestCase {
21     @Override
22     protected void setUp() throws Exception {
23         super.setUp();
24     }
25
26     public void testFramework() {
27         //...
28     }
29
30     public void testFramework2() {
31         //...
32     }
33
34     @Override
35     protected void tearDown() throws Exception {
36         // BAD: Does not call 'super.tearDown'. May cause later tests to fail
37         // when they try to re-initialize an already initialized framework.
38         // Even if the framework allows re-initialization, it may maintain the
39         // internal state, which could affect the results of succeeding tests.
```

```
40          System.out.println("Tests complete");
41      }
42  }
43
44  public class TearDownSuper extends FrameworkTestCase {
45      @Override
46      protected void setUp() throws Exception {
47          super.setUp();
48      }
49
50      public void testFramework() {
51          //...
52      }
53
54      public void testFramework2() {
55          //...
56      }
57
58      @Override
59      protected void tearDown() throws Exception {
60          // GOOD: Correctly calls 'super.tearDown' to shut down the
61          // framework.
62          System.out.println("Tests complete");
63          super.tearDown();
64      }
65  }
```

References

- JUnit: JUnit Cookbook.

Use the correct signature for a 'suite' method in JUnit

Category: Important > JUnit

Description: A 'suite' method in a JUnit 3.8 test that does not match the expected signature is not detected by JUnit.

JUnit 3.8 requires that a `suite` method for defining a `TestSuite` that will be used by a `TestRunner` has a specific signature. If the `suite` method does not have the expected signature, JUnit does not detect the method as a `suite` method.

Recommendation

Make sure that `suite` methods in junit `TestCase` classes are declared both `public` and `static`, and that they have a return type of `junit.framework.Test` or one of its subtypes.

Example

In the following example, `BadSuiteMethod.suite` is not detected by JUnit because it is not declared `public`. However, `CorrectSuiteMethod.suite` *is* detected by JUnit because it has the expected signature.

```
1  public class BadSuiteMethod extends TestCase {
2      // BAD: JUnit 3.8 does not detect the following method as a 'suite' method.
3      // The method should be public, static, and return 'junit.framework.Test'
4      // or one of its subtypes.
5      static Test suite() {
6          TestSuite suite = new TestSuite();
7          suite.addTest(new MyTests("testEquals"));
8          suite.addTest(new MyTests("testNotEquals"));
9          return suite;
10     }
11 }
12
13 public class CorrectSuiteMethod extends TestCase {
14     // GOOD: JUnit 3.8 correctly detects the following method as a 'suite' method.
15     public static Test suite() {
16         TestSuite suite = new TestSuite();
17         suite.addTest(new MyTests("testEquals"));
18         suite.addTest(new MyTests("testNotEquals"));
19         return suite;
20     }
21 }
```

References

- JUnit: JUnit Cookbook.

Logic Errors (1)

- Avoid extending or implementing an annotation
- Avoid nested loops that use the same variable
- Do not compare identical expressions

Avoid extending or implementing an annotation

Category: Important > Logic Errors (1)

Description: Extending or implementing an annotation is unlikely to be what the programmer intends.

Although an annotation type is a special kind of interface that can be implemented by a concrete class, this is not its intended use. It is more likely that an annotation type should be used to annotate a class.

Recommendation

Ensure that any annotations are used to annotate a class, unless they are really supposed to be extended or implemented by the class.

Example

In the following example, the annotation `Deprecated` is implemented by the class `ImplementsAnnotation`.

```
1 public abstract class ImplementsAnnotation implements Deprecated {
2     // ...
3 }
```

The following example shows the intended use of annotations: to annotate the class `ImplementsAnnotationFix`.

```
1 @Deprecated
2 public abstract class ImplementsAnnotationFix {
3     // ...
4 }
```

References

- The Java Language Specification: Annotation Types.
- The Java Tutorials: Annotations.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Avoid nested loops that use the same variable

Category: Important > Logic Errors (1)

Description: Nested loops in which the iteration variable is the same for each loop are difficult to understand.

The behavior of nested loops in which the iteration variable is the same for both loops is difficult to understand because the inner loop affects the iteration variable of the outer loop. This is probably a typographical error.

Recommendation

Ensure that a different iteration variable is used for each loop.

References

- The Java Language Specification: The basic for Statement.

Do not compare identical expressions

Category: Important > Logic Errors (1)

Description: If the same expression occurs on both sides of a comparison operator, the operator is redundant, and probably indicates a mistake.

If two identical expressions are compared (that is, checked for equality or inequality), this is typically an indication of a mistake, because the Boolean value of the comparison is always the same. Often, it indicates that the wrong qualifier has been used on a field access.

Recommendation

It is never good practice to compare a value with itself. If you require constant behavior, use the Boolean literals `true` and `false`, rather than encoding them obscurely as `1 == 1` or similar.

Example

In the example below, the original version of `Customer` compares `id` with `id`, which always returns `true`. The corrected version of `Customer` includes the missing qualifier `o` in the comparison of `id` with `o.id`.

```
1  class Customer {
2      ...
3      public boolean equals(Object o) {
4          if (o == null) return false;
5          if (Customer.class != o.getClass()) return false;
6          Customer other = (Customer)o;
7          if (!name.equals(o.name)) return false;
8          if (id != id) return false;  // Comparison of identical values
9          return true;
10     }
11 }
12
13 class Customer {
14     ...
15     public boolean equals(Object o) {
16         if (o == null) return false;
17         if (Customer.class != o.getClass()) return false;
18         Customer other = (Customer)o;
19         if (!name.equals(o.name)) return false;
20         if (id != o.id) return false;  // Comparison corrected
21         return true;
22     }
23 }
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Magic Constants

- Avoid magic numbers and add a named constant
- Avoid magic numbers and use an existing named constant
- Avoid magic strings and add a named constant
- Avoid magic strings and use an existing named constant

Avoid magic numbers and add a named constant

Category: Important > Magic Constants

Description: A magic number makes code less readable and maintainable.

A *magic number* is a numeric literal (for example, `8080`, `2048`) that is used in the middle of a block of code without explanation. It is considered bad practice to use magic numbers because:

- A number in isolation can be difficult for other programmers to understand.
- It can be difficult to update the code if the requirements change. For example, if the magic number represents the number of guests allowed, adding one more guest means that you must change every occurrence of the magic number.

Recommendation

Assign the magic number to a new named constant, and use this instead. This overcomes the two problems with magic numbers:

- A named constant (such as `MAX_GUESTS`) is more easily understood by other programmers.
- Using the same named constant in many places makes the code much easier to update if the requirements change, because you have to update the number in only one place.

Example

The following example shows a magic number `timeout`. This should be replaced by a new named constant, as shown in the fixed version.

```
1  // Problem version
2  public class MagicConstants
3  {
4      final static public String IP = "127.0.0.1";
5      final static public int PORT = 8080;
6      final static public String USERNAME = "test";
7
8      public void serve(String ip, int port, String user, int timeout) {
9          // ...
10      }
11
12      public static void main(String[] args) {
13          int timeout = 60000;  // AVOID: Magic number
14
15          new MagicConstants().serve(IP, PORT, USERNAME, timeout);
16      }
17 }
18
19
20 // Fixed version
21 public class MagicConstants
22 {
23     final static public String IP = "127.0.0.1";
24     final static public int PORT = 8080;
25     final static public String USERNAME = "test";
26     final static public int TIMEOUT = 60000;  // Magic number is replaced by named constant
27
28     public void serve(String ip, int port, String user, int timeout) {
29         // ...
30     }
31
```

```
32      public static void main(String[] args) {
33
34          new MagicConstants().serve(IP, PORT, USERNAME, TIMEOUT);  // Use 'TIMEOUT' constant
35      }
36  }
```

References

- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, §17.G25. Prentice Hall, 2008.

Avoid magic numbers and use an existing named constant

Category: Important > Magic Constants

Description: A magic number, which is used instead of an existing named constant, makes code less readable and maintainable.

A *magic number* is a numeric literal (for example, `8080`, `2048`) that is used in the middle of a block of code without explanation. It is considered bad practice to use magic numbers because:

- A number in isolation can be difficult for other programmers to understand.
- It can be difficult to update the code if the requirements change. For example, if the magic number represents the number of guests allowed, adding one more guest means that you must change every occurrence of the magic number.

Recommendation

Replace the magic number with the existing named constant. This overcomes the two problems with magic numbers:

- A named constant (such as `MAX_GUESTS`) is more easily understood by other programmers.
- Using the same named constant in many places makes the code much easier to update if the requirements change, because you have to update the number in only one place.

Example

The following example shows a magic number `internal_port`. This should be replaced by the existing named constant, as shown in the fixed version.

```
1  // Problem version
2  public class MagicConstants
3  {
4      final static public String IP = "127.0.0.1";
5      final static public int PORT = 8080;
6      final static public String USERNAME = "test";
7      final static public int TIMEOUT = 60000;
8
9      public void serve(String ip, int port, String user, int timeout) {
10         // ...
11      }
12
13      public static void main(String[] args) {
14         int internal_port = 8080;  // AVOID: Magic number
15
16         new MagicConstants().serve(IP, internal_port, USERNAME, TIMEOUT);
17     }
18 }
19
20
21 // Fixed version
22 public class MagicConstants
23 {
24     final static public String IP = "127.0.0.1";
25     final static public int PORT = 8080;
26     final static public String USERNAME = "test";
27     final static public int TIMEOUT = 60000;
28
29     public void serve(String ip, int port, String user, int timeout) {
30         // ...
```

```
31      }
32
33      public static void main(String[] args) {
34
35          new MagicConstants().serve(IP, PORT, USERNAME, TIMEOUT);  // Use 'PORT' constant
36      }
37  }
```

References

- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, §17.G25. Prentice Hall, 2008.

Avoid magic strings and add a named constant

Category: Important > Magic Constants

Description: A magic string makes code less readable and maintainable.

A *magic string* is a string literal (for example, `"SELECT"`, `"127.0.0.1"`) that is used in the middle of a block of code without explanation. It is considered bad practice to use magic strings because:

- A string in isolation can be difficult for other programmers to understand.
- It can be difficult to update the code if the requirements change. For example, if the magic string represents a protocol, changing the protocol means that you must change every occurrence of the protocol.

Recommendation

Assign the magic string to a new named constant, and use this instead. This overcomes the two problems with magic strings:

- A named constant (such as `SMTP_HELO`) is more easily understood by other programmers.
- Using the same named constant in many places makes the code much easier to update if the requirements change, because you have to update the string in only one place.

Example

The following example shows a magic string `username`. This should be replaced by a new named constant, as shown in the fixed version.

```
1  // Problem version
2  public class MagicConstants
3  {
4      final static public String IP = "127.0.0.1";
5      final static public int PORT = 8080;
6      final static public int TIMEOUT = 60000;
7
8      public void serve(String ip, int port, String user, int timeout) {
9          // ...
10      }
11
12      public static void main(String[] args) {
13          String username = "test";  // AVOID: Magic string
14
15          new MagicConstants().serve(IP, PORT, username, TIMEOUT);
16      }
17 }
18
19
20 // Fixed version
21 public class MagicConstants
22 {
23     final static public String IP = "127.0.0.1";
24     final static public int PORT = 8080;
25     final static public int USERNAME = "test";  // Magic string is replaced by named constant
26     final static public int TIMEOUT = 60000;
27
28     public void serve(String ip, int port, String user, int timeout) {
29         // ...
30     }
31
```

```
32      public static void main(String[] args) {
33
34          new MagicConstants().serve(IP, PORT, USERNAME, TIMEOUT);  // Use 'USERNAME' constant
35      }
36  }
```

References

- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, §17.G25. Prentice Hall, 2008.

Avoid magic strings and use an existing named constant

Category: Important > Magic Constants

Description: A magic string, which is used instead of an existing named constant, makes code less readable and maintainable.

A *magic string* is a string literal (for example, `"SELECT"`, `"127.0.0.1"`) that is used in the middle of a block of code without explanation. It is considered bad practice to use magic strings because:

- A string in isolation can be difficult for other programmers to understand.
- It can be difficult to update the code if the requirements change. For example, if the magic string represents a protocol, changing the protocol means that you must change every occurrence of the protocol.

Recommendation

Replace the magic string with the existing named constant. This overcomes the two problems with magic strings:

- A named constant (such as `SMTP_HELO`) is more easily understood by other programmers.
- Using the same named constant in many places makes the code much easier to update if the requirements change, because you have to update the string in only one place.

Example

The following example shows a magic string `internal_ip`. This should be replaced by the existing named constant, as shown in the fixed version.

```
1  // Problem version
2  public class MagicConstants
3  {
4      final static public String IP = "127.0.0.1";
5      final static public int PORT = 8080;
6      final static public String USERNAME = "test";
7      final static public int TIMEOUT = 60000;
8
9      public void serve(String ip, int port, String user, int timeout) {
10         // ...
11      }
12
13      public static void main(String[] args) {
14         String internal_ip = "127.0.0.1";  // AVOID: Magic string
15
16         new MagicConstants().serve(internal_ip, PORT, USERNAME, TIMEOUT);
17     }
18 }
19
20
21 // Fixed version
22 public class MagicConstants
23 {
24     final static public String IP = "127.0.0.1";
25     final static public int PORT = 8080;
26     final static public String USERNAME = "test";
27     final static public int TIMEOUT = 60000;
28
29     public void serve(String ip, int port, String user, int timeout) {
30         // ...
31     }
```

```
32
33      public static void main(String[] args) {
34
35          new MagicConstants().serve(IP, PORT, USERNAME, TIMEOUT);  //Use 'IP' constant
36      }
37  }
```

References

- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, §17.G25. Prentice Hall, 2008.

Naming (2)

- Avoid declaring a method with the name 'equal'
- Avoid declaring a method with the name 'hashcode'
- Avoid declaring a method with the name 'tostring'
- Avoid methods in the same class whose names differ only in capitalization
- Avoid naming a class with the same name as its superclass
- Avoid overloaded methods that have similar parameter types
- Avoid using 'enum' as an identifier

Avoid declaring a method with the name 'equal'

Category: Important > Naming (2)

Description: A method named 'equal' may be intended to be named 'equals'.

A method named `equal` may be a typographical error. `equals` may have been intended instead.

Recommendation

Ensure that any such method is intended to have this name. Even if it is, it may be better to rename it to avoid confusion with the inherited method `Object.equals`.

Example

The following example shows a method named `equal`. It may be better to rename it.

```
1  public class Complex
2  {
3      private double real;
4      private double complex;
5
6      // ...
7
8      public boolean equal(Object obj) {  // The method is named 'equal'.
9          if (!getClass().equals(obj.getClass()))
10             return false;
11         Complex other = (Complex) obj;
12         return real == other.real && complex == other.complex;
13     }
14 }
```

References

- Java 2 Platform, Standard Edition 5.0, API Specification: equals.

Avoid declaring a method with the name 'hashcode'

Category: Important > Naming (2)

Description: A method named 'hashcode' may be intended to be named 'hashCode'.

A method named `hashcode` may be a typographical error. `hashCode` (different capitalization) may have been intended instead.

Recommendation

Ensure that any such method is intended to have this name. Even if it is, it may be better to rename it to avoid confusion with the inherited method `Object.hashCode`.

Example

The following example shows a method named `hashcode`. It may be better to rename it.

```
1  public class Person
2  {
3      private String title;
4      private String forename;
5      private String surname;
6
7      // ...
8
9      public int hashcode() {  // The method is named 'hashcode'.
10         int hash = 23 * title.hashCode();
11         hash ^= 13 * forename.hashCode();
12         return hash ^ surname.hashCode();
13     }
14 }
```

References

- Java 2 Platform, Standard Edition 5.0, API Specification: hashCode.

Avoid declaring a method with the name 'tostring'

Category: Important > Naming (2)

Description: A method named 'tostring' may be intended to be named 'toString'.

A method named `tostring` may be a typographical error. `toString` (different capitalization) may have been intended instead.

Recommendation

Ensure that any such method is intended to have this name. Even if it is, it may be better to rename it to avoid confusion with the inherited method `Object.toString`.

Example

The following example shows a method named `tostring`. It may be better to rename it.

```
1  public class Customer
2  {
3      private String title;
4      private String forename;
5      private String surname;
6
7      // ...
8
9      public String tostring() {  // The method is named 'tostring'.
10         return title + " " + forename + " " + surname;
11     }
12 }
```

References

- Java 2 Platform, Standard Edition 5.0, API Specification: toString.

Avoid methods in the same class whose names differ only in capitalization

Category: Important > Naming (2)

Description: Methods in the same class whose names differ only in capitalization are confusing.

It is bad practice to have methods in a class with names that differ only in their capitalization. This can be confusing and lead to mistakes.

Recommendation

Name the methods to make the distinction between them clear.

Example

The following example shows a class that contains two methods: `toUri` and `toURI`. One or both of them should be renamed.

```
1  public class InternetResource
2  {
3      private String protocol;
4      private String host;
5      private String path;
6
7      // ...
8
9      public String toUri() {
10         return protocol + "://" + host + "/" + path;
11     }
12
13     // ...
14
15     public String toURI() {
16         return toUri();
17     }
18 }
```

References

- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 17.N4. Prentice Hall, 2008.

Avoid naming a class with the same name as its superclass

Category: Important > Naming (2)

Description: A class that has the same name as its superclass may be confusing.

A class that has the same name as its superclass may be confusing.

Recommendation

Clarify the difference between the subclass and the superclass by using different names.

Example

In the following example, it is not clear that the `attendees` field refers to the inner class `Attendees` and not the class `com.company.util.Attendees`.

```
1  import com.company.util.Attendees;
2
3  public class Meeting
4  {
5      private Attendees attendees;
6
7      // ...
8      // Many lines
9      // ...
10
11      // AVOID: This class has the same name as its superclass.
12      private static class Attendees extends com.company.util.Attendees
13      {
14          // ...
15      }
16 }
```

To fix this, the inner class should be renamed.

References

- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, §17.N4. Prentice Hall, 2008.

Avoid overloaded methods that have similar parameter types

Category: Important > Naming (2)

Description: Overloaded methods that have the same number of parameters, where each pair of corresponding parameter types is convertible by casting or autoboxing, may be confusing.

Overloaded method declarations that have the same number of parameters may be confusing if none of the corresponding pairs of parameter types is substantially different. A pair of parameter types A and B is substantially different if A cannot be cast to B and B cannot be cast to A. If the parameter types are not substantially different then the programmer may assume that the method with parameter type A is called when in fact the method with parameter type B is called.

Recommendation

It is generally best to avoid declaring overloaded methods with the same number of parameters, unless at least one of the corresponding parameter pairs is substantially different.

Example

Declaring overloaded methods `process(Object obj)` and `process(String s)` is confusing because the parameter types are not substantially different. It is clearer to declare methods with different names: `processObject(Object obj)` and `processString(String s)`.

In contrast, declaring overloaded methods `process(Object obj, String s)` and `process(String s, int i)` is not as confusing because the second parameters of each method are substantially different.

References

- J. Bloch, *Effective Java (second edition)*, Item 41. Addison-Wesley, 2008.
- Java Language Specification: 15.12 Method Invocation Expressions.

Avoid using 'enum' as an identifier

Category: Important > Naming (2)

Description: Using 'enum' as an identifier makes the code incompatible with Java 5 and later.

Enumerations, or enums, were introduced in Java 5, with the keyword `enum`. Code written before this may use `enum` as an identifier. To compile such code, you must compile it with a command such as `javac -source 1.4 ...`. However, this means that you cannot use any new features that are provided in Java 5 and later.

Recommendation

To make it easier to compile the code and add code that uses new Java features, rename any identifiers that are named `enum` in legacy code.

Example

In the following example, `enum` is used as the name of a variable. This means that the code does not compile unless the compiler's source language is set to 1.4 or earlier. To avoid this constraint, the variable should be renamed.

```
1 class Old
2 {
3     public static void main(String[] args) {
4         int enum = 13;  // AVOID: 'enum' is a variable.
5         System.out.println("The value of enum is " + enum);
6     }
7 }
```

References

- Java Language Specification: 8.9 Enums.

Random (1)

- Do not create an instance of 'Random' for each pseudo-random number required

Do not create an instance of 'Random' for each pseudo-random number required

Category: Important > Random (1)

Description: Creating an instance of 'Random' for each pseudo-random number required does not guarantee an evenly distributed sequence of random numbers.

A program that uses `java.util.Random` to generate a sequence of pseudo-random numbers *should not* create a new instance of `Random` every time a new pseudo-random number is required (for example, `new Random().nextInt()`).

According to the Java API specification:

> *If two instances of `Random` are created with the same seed, and the same sequence of method calls is made for each, they will generate and return identical sequences of numbers.*

The sequence of pseudo-random numbers returned by these calls depends only on the value of the seed. If you construct a new `Random` object each time a pseudo-random number is needed, this does not generate a good distribution of pseudo-random numbers, even though the parameterless `Random()` constructor tries to initialize itself with a unique seed.

Recommendation

Create a `Random` object once and use the same instance when generating sequences of pseudo-random numbers (by calling `nextInt`, `nextLong`, and so on).

Example

In the following example, generating a series of pseudo-random numbers, such as `notReallyRandom` and `notReallyRandom2`, by creating a new instance of `Random` each time is unlikely to result in a good distribution of pseudo-random numbers. In contrast, generating a series of pseudo-random numbers, such as `random1` and `random2`, by calling `nextInt` each time *is* likely to result in a good distribution. This is because the numbers are based on only one `Random` object.

```
1  public static void main(String args[]) {
2      // BAD: A new 'Random' object is created every time
3      // a pseudo-random integer is required.
4      int notReallyRandom = new Random().nextInt();
5      int notReallyRandom2 = new Random().nextInt();
6
7      // GOOD: The same 'Random' object is used to generate
8      // two pseudo-random integers.
9      Random r = new Random();
10      int random1 = r.nextInt();
11      int random2 = r.nextInt();
12 }
```

References

- Java API Documentation: Random.

Result Checking

- Avoid calling 'next' from an iterator implementation of 'hasNext'
- Do not ignore a method's return value
- Ensure that the results of all method calls are used
- Handle the results of calls to a particular method consistently

Avoid calling 'next' from an iterator implementation of 'hasNext'

Category: Important > Result Checking

Description: Iterator implementations whose 'hasNext' method calls 'next' are most likely incorrect.

Iterator implementations with a `hasNext` method that calls the `next` method are most likely incorrect. This is because `next` changes the iterator's position to the next element and returns that element, which is unlikely to be desirable in the implementation of `hasNext`.

Recommendation

Ensure that any calls to `next` from within `hasNext` are legitimate. The `hasNext` method should indicate whether there are further elements remaining in the iteration without changing the iterator's state by calling `next`.

Example

In the following example, which outputs the contents of a string, `hasNext` calls `next`, which has the effect of changing the iterator's position. Given that `main` also calls `next` when it outputs an item, some items are skipped and only half the items are output.

```
1  public class NextFromIterator implements Iterator<String> {
2      private int position = -1;
3      private List<String> list = new ArrayList<String>() {{
4          add("alpha"); add("bravo"); add("charlie"); add("delta"); add("echo"); add("foxtrot");
5      }};
6
7      public boolean hasNext() {
8          return next() != null;  // BAD: Call to 'next'
9      }
10
11      public String next() {
12          position++;
13          return position < list.size() ? list.get(position) : null;
14      }
15
16      public void remove() {
17          // ...
18      }
19
20      public static void main(String[] args) {
21          NextFromIterator x = new NextFromIterator();
22          while(x.hasNext()) {
23              System.out.println(x.next());
24          }
25      }
26  }
```

Instead, the implementation of `hasNext` should use another way of indicating whether there are further elements in the string without calling `next`. For example, `hasNext` could check the underlying array directly to see if there is an element at the next position.

References

- Java API Documentation: Iterator.hasNext(), Iterator.next().

Do not ignore a method's return value

Category: Important > Result Checking

Description: Ignoring an exceptional value that is returned by a method may cause subsequent code to fail.

Many methods in the Java Development Kit (for examples, see the references below) return status values (for example, as an `int`) to indicate whether the method execution finished normally. They may return an error code if the method did not finish normally. If the method result is not checked, exceptional method executions may cause subsequent code to fail.

Recommendation

You should insert additional code to check the return value and take appropriate action.

Example

The following example uses the `java.io.InputStream.read` method to read 16 bytes from an input stream and store them in an array. However, `read` may not actually be able to read as many bytes as requested, for example because the stream is exhausted. Therefore, the code should not simply rely on the array `b` being filled with precisely 16 bytes from the input stream. Instead, the code should check the method's return value, which indicates the number of bytes actually read.

```
1 java.io.InputStream is = (...);
2 byte[] b = new byte[16];
3 is.read(b);
```

References

- CERT Secure Coding Standards: EXP00-J. Do not ignore values returned by methods.
- Java API Documentation, java.util.Queue: offer.
- Java API Documentation, java.util.concurrent.BlockingQueue: offer.
- Java API Documentation, java.util.concurrent.locks.Condition: await, awaitUntil, awaitNanos.
- Java API Documentation, java.io.File: createNewFile, delete, mkdir, mkdirs, renameTo, setLastModified, setReadOnly, setWritable(boolean), setWritable(boolean, boolean).
- Java API Documentation, java.io.InputStream: skip, read(byte[]), read(byte[], int, int).

Ensure that the results of all method calls are used

Category: Important > Result Checking

Description: If most of the calls to a method use the return value of that method, the calls that do not check the return value may be mistakes.

If the result of a method call is used in most cases, any calls to that method where the result is ignored are inconsistent, and may be erroneous uses of the API. Often, the result is some kind of status indicator, and is therefore important to check.

Recommendation

Ensure that the results of *all* calls to a particular method are used. The return value of a method that returns a status value should normally be checked before any modified data or allocated resource is used.

Example

Line 1 of the following example shows the value returned by `get` being ignored. Line 3 shows it being assigned to `fs`.

```
1 FileSystem.get(conf);  // Return value is not used
2
3 FileSystem fs = FileSystem.get(conf);  // Return value is assigned to 'fs'
```

References

- CERT Secure Coding Standards: EXP00-J. Do not ignore values returned by methods.

Handle the results of calls to a particular method consistently

Category: Important > Result Checking

Description: If the same operation is usually performed on the result of a method call, any cases where it is not performed may indicate resource leaks or other problems.

If the same operation (for example, `free`, `delete`, `close`) is usually performed on the result of a method call, any instances where it is not performed may be misuses of the API, leading to resource leaks or other problems.

Recommendation

Ensure that the same operation is performed on the result of *all* calls to a particular method, if appropriate.

Example

In the following example of good usage, the result of the call to `writer.prepareAppendValue` is assigned to `outValue`, and later `close` is called on `outValue`. Any instances where `close` is *not* called may cause resource leaks.

```
1  DataOutputStream outValue = null;
2  try {
3      outValue = writer.prepareAppendValue(6);
4      outValue.write("value0".getBytes());
5  }
6  catch (IOException e) {
7  }
8  finally {
9      if (outValue != null) {
10         outValue.close();
11      }
12 }
```

Size

- Avoid creating classes that contain many fields
- Avoid creating files that contain many lines of code
- Avoid creating methods that contain many levels of nesting
- Avoid creating methods that contain many lines of code
- Avoid creating methods that have many parameters
- Avoid too many complex statements in a block
- Review files that have been changed by many authors

Avoid creating classes that contain many fields

Category: Important > Size

Description: A class that contains a high number of fields may be too big or need refactoring. The number of fields should be less than 26.

A class that contains a high number of fields may indicate the following problems:

- The class may be too big or have too many responsibilities.
- Several of the fields may be part of the same abstraction.

Recommendation

The solution depends on the reason for the high number of fields:

- If the class is too big, you should split it into multiple smaller classes.
- If several of the fields are part of the same abstraction, you should group them into a separate class, using the 'Extract Class' refactoring described in [Fowler].

Example

In the following example, class `Person` contains a number of fields.

```
1  class Person {
2      private String m_firstName;
3      private String m_LastName;
4      private int m_houseNumber;
5      private String m_street;
6      private String m_settlement;
7      private Country m_country;
8      private Postcode m_postcode;
9      // ...
10 }
```

This can be refactored by grouping fields that are part of the same abstraction into new classes `Name` and `Address`.

```
1  class Name {
2      private String m_firstName;
3      private String m_lastName;
4      // ...
5  }
6
7  class Address {
8      private int m_houseNumber;
9      private String m_street;
10      private String m_settlement;
11      private Country m_country;
12      private Postcode m_postcode;
13      // ...
14 }
15
16 class Person {
17     private Name m_name;
18     private Address m_address;
19     // ...
20 }
```

References

- M. Fowler, *Refactoring*. Addison-Wesley, 1999.

Avoid creating files that contain many lines of code

Category: Important > Size

Description: A file that contains a high number of lines of code may be difficult to maintain, increases the likelihood of merge conflicts, may increase network traffic, and may indicate weak code organisation. The number of lines in the file should be less than 1000.

A file that contains a high number of lines of code has a number of problems:

- It can be difficult to understand and maintain, even with good tool support.
- It increases the likelihood of multiple developers needing to work on the same file at once, and it therefore increases the likelihood of merge conflicts.
- It may increase network traffic if you use a version control system that requires the whole file to be transmitted even for a tiny change.
- It may arise as a result of bundling many unrelated things into the same file, and so it can indicate weak code organisation.

Recommendation

The solution depends on the reason for the high number of lines:

- If the file's main class is too large, you should refactor it into smaller classes, for example by using the 'Extract Class' refactoring from [Fowler].
- If the file's main class contains many nested classes, you should move the nested classes to their own files (in a subsidiary package, where appropriate).
- If the file contains multiple non-public classes in addition to its main class, you should move them into separate files. This is particularly important if they are logically unrelated to the file's main class.
- If the file has been automatically generated by a tool, no changes are required because the file will not be maintained by a programmer.

References

- M. Fowler, *Refactoring*. Addison-Wesley, 1999.

Avoid creating methods that contain many levels of nesting

Category: Important > Size

Description: A method that contains a high level of nesting may be difficult to understand. The number of levels should be less than 10.

A method that contains a high level of nesting can be very difficult to understand. As noted in [McConnell], the human brain cannot easily handle more than three levels of nested `if` statements.

Recommendation

Extract nested statements into new methods, for example by using the 'Extract Method' refactoring from [Fowler].

For more ways to reduce the level of nesting in a method, see [McConnell].

Furthermore, a method that has a high level of nesting often indicates that its design can be improved in other ways, as well as dealing with the nesting problem itself.

Example

In the following example, the code has four levels of nesting and is unnecessarily difficult to read.

```
1  public static void printCharacterCodes_Bad(String[] strings) {
2      if (strings != null) {
3          for (String s : strings) {
4              if (s != null) {
5                  for (int i = 0; i < s.length(); i++) {
6                      System.out.println(s.charAt(i) + "=" + (int) s.charAt(i));
7                  }
8              }
9          }
10      }
11  }
```

In the following modified example, some of the nested statements have been extracted into a new method `PrintAllCharInts`.

```
1  public static void printAllCharInts(String s) {
2      if (s != null) {
3          for (int i = 0; i < s.length(); i++) {
4              System.out.println(s.charAt(i) + "=" + (int) s.charAt(i));
5          }
6      }
7  }
8  public static void printCharacterCodes_Good(String[] strings) {
9      if (strings != null) {
10           for(String s : strings){
11               printAllCharInts(s);
12           }
13      }
14  }
```

References

- M. Fowler, *Refactoring*, pp. 89-95. Addison-Wesley, 1999.
- S. McConnell, *Code Complete*, 2nd Edition, §19.4. Microsoft Press, 2004.

Avoid creating methods that contain many lines of code

Category: Important > Size

Description: A method that contains a high number of lines of code may be difficult to maintain and is likely to lack cohesion. The number of lines in the method should be less than 300.

A method that contains a high number of lines of code has a number of problems:

- It can be difficult to understand, difficult to check, and a common source of defects (particularly towards the end of the method, because few people read that far).
- It is likely to lack cohesion because it has too many responsibilities.
- It increases the risk of introducing new defects during routine code changes.

Recommendation

Break up long methods into smaller methods by extracting parts of their functionality into simpler methods, for example by using the 'Extract Method' refactoring from [Fowler]. As an approximate guide, a method should fit on one screen or side of Letter/A4 paper.

References

- M. Fowler, *Refactoring*, pp. 89-95. Addison-Wesley, 1999.

Avoid creating methods that have many parameters

Category: Important > Size

Description: A method or constructor that has a high number of parameters makes maintenance more difficult. The number of parameters should be less than 9.

A method (or constructor) that uses a high number of formal parameters makes maintenance more difficult:

- It is difficult to write a call to the method, because the programmer must know how to supply an appropriate value for each parameter.
- It is *externally* difficult to understand, because calls to the method are longer than a single line of code.
- It can be *internally* difficult to understand, because it has so many dependencies.

Recommendation

Restrict the number of formal parameters for a method, according to the reason for the high number:

- Several of the parameters are logically related, but are passed into the method separately. The parameters that are logically related should be grouped together (see the 'Introduce Parameter Object' refactoring on pp. 238-242 of [Fowler]).
- The method has too many responsibilities. It should be broken into multiple methods (see the 'Extract Method' refactoring on pp. 89-95 of [Fowler]), and each new method should be passed a subset of the original parameters.
- The method has redundant parameters that are not used. The two main reasons for this are: (1) parameters were added for future extensibility but are never used; (2) the body of the method was changed so that it no longer uses certain parameters, but the method signature was not correspondingly updated. In both cases, the theoretically correct solution is to delete the unused parameters (see the 'Remove Parameter' refactoring on pp. 223-225 of [Fowler]), although you must do this cautiously if the method is part of a published interface.

When a method is part of a published interface, one possible solution is to add a new, wrapper method to the interface that has a tidier signature. Alternatively, you can publish a new version of the interface that has a better design. Clearly, however, neither of these solutions is ideal, so you should take care to design interfaces the right way from the start.

The practice of adding parameters for future extensibility is especially bad. It is confusing to other programmers, who are uncertain what values they should pass in for these unnecessary parameters, and it adds unused code that is potentially difficult to remove later.

Examples

In the following example, although the parameters are logically related, they are passed into the `printAnnotation` method separately.

```
1 void printAnnotation(String annotationMessage, int annotationLine, int annotationOffset,
2                       int annotationLength) {
3     System.out.println("Message: " + annotationMessage);
4     System.out.println("Line: " + annotationLine);
5     System.out.println("Offset: " + annotationOffset);
6     System.out.println("Length: " + annotationLength);
```

In the following modified example, the parameters that are logically related are grouped together in a class, and an instance of the class is passed into the method instead.

```
1 class Annotation {
```

```
    //...
}

void printAnnotation(Annotation annotation) {
    System.out.println("Message: " + annotation.getMessage());
    System.out.println("Line: " + annotation.getLine());
    System.out.println("Offset: " + annotation.getOffset());
    System.out.println("Length: " + annotation.getLength());
}
```

In the following example, the `printMembership` method has too many responsibilities, and so needs to be passed four arguments.

```
void printMembership(Set<Fellow> fellows, Set<Member> members,
                     Set<Associate> associates, Set<Student> students) {
    for(Fellow f: fellows) {
        System.out.println(f);
    }
    for(Member m: members) {
        System.out.println(m);
    }
    for(Associate a: associates) {
         System.out.println(a);
     }
     for(Student s: students) {
         System.out.println(s);
     }
}

void printRecords() {
    //...
    printMembership(fellows, members, associates, students);
}
```

In the following modified example, `printMembership` has been broken into four methods. (For brevity, only one method is shown.) As a result, each new method needs to be passed only one of the original four arguments.

```
void printFellows(Set<Fellow> fellows) {
    for(Fellow f: fellows) {
        System.out.println(f);
    }
}

//...

void printRecords() {
    //...
    printFellows(fellows);
    printMembers(members);
    printAssociates(associates);
    printStudents(students);
}
```

References

- M. Fowler, *Refactoring*. Addison-Wesley, 1999.

Avoid too many complex statements in a block

Category: Important > Size

Description: A block that contains too many complex statements becomes unreadable and unmaintainable.

Code has a tendency to become more complex over time. A method that is initially simple may need to be extended to accommodate additional functionality or to address defects. Before long it becomes unreadable and unmaintainable, with many complex statements nested within each other.

This rule applies to a block that contains a significant number of complex statements. Note that this is quite different from just considering the number of statements in a block, because each complex statement is potentially a candidate for being extracted to a new method as part of refactoring. For the purposes of this rule, loops and switch statements are considered to be complex.

Recommendation

To make the code more understandable and less complex, identify logical units and extract them to new methods. As a result, the top-level logic becomes clearer.

References

- M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
- W. C. Wake, *Refactoring Workbook*. Addison-Wesley Professional, 2004.

Review files that have been changed by many authors

Category: Important > Size

Description: A file that has been worked on by a high number of authors is a potential source of defects, and may lack conceptual integrity. Ideally, the number of authors should be less than 4.

A file's Javadoc comment can include a tag that lists the authors who have worked on the file.

A file that has been changed by a large number of different authors is the product of many minds. New authors working on the file may be less familiar with the design and implementation of the code than the original authors, which can be a potential source of defects. Furthermore, if the code is not carefully maintained, it often results in a lack of conceptual integrity.

Recommendation

There is clearly no way to reduce the number of authors that have worked on a file - it is impossible to rewrite history. However, you should pay special attention in a code review to a file that has been worked on by a large number of authors. The file may be need to be refactored or rewritten by an individual, experienced programmer.

References

- F. P. Brooks Jr, *The Mythical Man-Month*, Chapter 4. Addison-Wesley, 1974.

Spring

- Add 'description' elements to Spring bean definitions
- A non-abstract parent Spring bean must not specify an abstract class
- Avoid defining too many Spring beans in the same file
- Avoid overriding a property with the same contents in a child Spring bean
- Avoid using autowiring in Spring beans
- Create a common parent bean for Spring beans that share properties
- Ensure that each property in a Spring bean definition has a matching setter
- Put 'import' statements before Spring bean definitions
- Use 'id' instead of 'name' to name a Spring bean
- Use a type name instead of an index number in a Spring 'constructor-arg' element
- Use local references when referring to Spring beans in the same file
- Use setter injection instead of constructor injection when using Spring
- Use shortcut forms in Spring bean definitions

Add 'description' elements to Spring bean definitions

Category: Important > Spring

Description: Adding 'description' elements to a Spring XML bean definition file is good practice.

In a Spring XML bean definition file, adding a `<description>` element to a `<bean>` element or the enclosing `<beans>` element to document the purpose of the bean specification is good practice. A `description` element also has the advantage of making it easier for tools to detect and display the documentation for your bean specifications.

Recommendation

Add a `<description>` element either in the `<bean>` element or its enclosing `<beans>` element.

Example

The following example shows a Spring XML bean definition file that includes `<description>` elements.

```
1  <beans>
2      <!--Using a description element makes it easier for tools to pick up
3          documentation of the bean configuration-->
4      <description>
5      This file configures the various service beans.
6      </description>
7
8      <!--You can also put a description element in a bean-->
9      <bean id="baseService" abstract="true">
10          <description>
11          This bean defines base properties common to the service beans
12          </description>
13          ...
14      </bean>
15
16      <bean id="shippingService"
17              class="documentation.examples.spring.ShippingService"
18              parent="baseService">
19          ...
20      </bean>
21
22      <bean id="orderService"
23              class="documentation.examples.spring.OrderService"
24              parent="baseService">
25          ...
26      </bean>
27  </beans>
```

References

- ONJava: Twelve Best Practices For Spring XML Configurations.

A non-abstract parent Spring bean must not specify an abstract class

Category: Important > Spring

Description: A non-abstract Spring bean that is a parent of other beans and specifies an abstract class causes an error during bean instantiation.

A non-abstract Spring bean that is a parent of other beans must not specify an abstract class. Doing so causes an error during bean instantiation.

Recommendation

Make sure that a non-abstract bean does not specify an abstract class, by doing one of the following:

- Specify that the bean is also abstract by adding `abstract="true"` to the bean specification.
- If possible, update the class that is specified by the bean so that it is not abstract.

You can also make the XML parent bean definition abstract and remove any references from it to any class (in which case it becomes a pure bean template). Note that, like an abstract class, an abstract bean cannot be used on its own and only provides property and constructor definitions to its children.

Example

In the following example, the bean `wrongConnectionPool` is using an abstract class, `ConnectionPool`, which causes an error. Instead, the bean should be declared `abstract`, as shown in the definition of `connectionPool`.

```
1  <beans>
2      <!--BAD: A non-abstract bean should use a concrete class.
3          'ConnectionPool' is an abstract class.-->
4      <bean id="wrongConnectionPool"
5              class="documentation.examples.spring.ConnectionPool"/>
6      <bean id="appReqPool1" class="documentation.examples.spring.AppRequestConnectionPool"
7              parent="wrongConnectionPool"/>
8
9      <!--GOOD: A bean that specifies an abstract class should be declared 'abstract'.-->
10      <bean id="connectionPool"
11              class="documentation.examples.spring.ConnectionPool" abstract="true"/>
12      <bean id="appReqPool2" class="documentation.examples.spring.AppRequestConnectionPool"
13              parent="connectionPool"/>
14 </beans>
```

References

- Spring Framework Reference Documentation 3.0: 3.7 Bean definition inheritance.

Avoid defining too many Spring beans in the same file

Category: Important > Spring

Description: Too many beans in a file can make the file difficult to understand and maintain.

Too many bean definitions in a single file can make the file difficult to understand and maintain. It is also an indication that the architecture of the system is too tightly coupled and can be refactored.

Recommendation

Refactor related bean definitions into separate files, and compose them using the `<import/>` element.

Example

The following example shows a configuration file that imports two other configuration files. These two files were created by refactoring a file that contained too many bean definitions.

```
<beans>
    <!--Compose configuration files by using the 'import' element.-->
    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>
```

References

- Spring Framework Reference Documentation 3.0: 3.2.2.1 Composing XML-based configuration metadata.

Avoid overriding a property with the same contents in a child Spring bean

Category: Important > Spring

Description: A bean property that overrides the same property in a parent bean, and has the same contents, is useless.

A property in a child bean that overrides a property with the same name in its parent and has the same contents is useless. This is because the bean inherits the property from its parent anyway.

Recommendation

If possible, remove the property in the child bean.

Example

In the following example, `registry` is defined in both the parent bean and the child bean. It should be removed from the child bean.

```
<beans>
    <bean id="baseShippingService" abstract="true">
        <property name="transactionHelper">
            <ref bean="transactionHelper"/>
        </property>
        <property name="dao">
            <ref bean="dao"/>
        </property>
        <property name="registry">
             <ref bean="basicRegistry"/>
         </property>
     </bean>

     <bean id="shippingService"
             class="documentation.examples.spring.ShippingService"
             parent="baseShippingService">
         <!--AVOID: This property is already defined with the same value in the parent bean.-->
         <property name="registry">
             <ref bean="basicRegistry"/>
         </property>
         <property name="shippingProvider" value="Federal Parcel Service"/>
     </bean>
</beans>
```

References

- Spring Framework Reference Documentation 3.0: 3.7 Bean definition inheritance.

Avoid using autowiring in Spring beans

Category: Important > Spring

Description: Using autowiring in Spring beans may make it difficult to maintain large projects.

Using Spring autowiring can make it difficult to see what beans get passed to constructors or setters. The Spring Framework Reference documentation cites the following disadvantages of autowiring:

- Explicit dependencies in `property` and `constructor-arg` settings always override autowiring. You cannot autowire so-called *simple* properties such as primitives, `Strings`, and `Classes` (and arrays of such simple properties). This limitation is by design.
- Autowiring is less exact than explicit wiring. Although ... Spring is careful to avoid guessing in case of ambiguity that might have unexpected results, the relationships between your Spring-managed objects are no longer documented explicitly.
- Wiring information may not be available to tools that may generate documentation from a Spring container.
- Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Maps, this is not necessarily a problem. However for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

Recommendation

The Spring Framework Reference documentation suggests the following ways to address problems with autowired beans:

- Abandon autowiring in favor of explicit wiring.
- Avoid autowiring for a bean definition by setting its `autowire-candidate` attributes to `false`.
- Designate a single bean definition as the primary candidate by setting the `primary` attribute of its `<bean/>` element to true.
- If you are using Java 5 or later, implement the more fine-grained control available with annotation-based configuration.

Example

The following example shows a bean, `autoWiredOrderService`, that is defined using autowiring, and an improved version of the bean, `orderService`, that is defined using explicit wiring.

```
1  <!--AVOID: Using autowiring makes it difficult to see the dependencies of the bean-->
2  <bean id="autoWiredOrderService"
3          class="documentation.examples.spring.OrderService"
4          autowire="byName"/>
5
6  <!--GOOD: Explicitly specifying the properties of the bean documents its dependencies
7      and makes the bean configuration easier to maintain-->
8  <bean id="orderService"
9          class="documentation.examples.spring.OrderService">
10          <property name="DAO">
11              <idref bean="dao"/>
12          </property>
13 </bean>
```

References

- Spring Framework Reference Documentation 3.0: 3.4.5.1 Limitations and disadvantages of autowiring.
- ONJava: Twelve Best Practices For Spring XML Configurations.

Create a common parent bean for Spring beans that share properties

Category: Important > Spring

Description: Beans that share similar properties exhibit unnecessary repetition in the bean definitions and make the system's architecture more difficult to see.

Beans that share a considerable number of similar properties exhibit unnecessary repetition in the bean definitions and make the system's architecture more difficult to see.

Recommendation

Try to move the properties that the bean definitions share to a common parent bean. This reduces repetition in the bean definitions and gives a clearer picture of the system's architecture.

Example

The following example shows a configuration file that contains two beans that share several properties with the same values.

```
<!--AVOID: 'shippingService' and 'orderService' share several properties with the same values-->
<bean id="shippingService" class="documentation.examples.spring.ShippingService">
    <property name="transactionHelper">
        <ref bean="transactionHelper"/>
    </property>
    <property name="dao">
        <ref bean="dao"/>
    </property>
    <property name="registry">
         <ref bean="basicRegistry"/>
     </property>

     <property name="shippingProvider" value="Federal Parcel Service"/>
</bean>

<bean id="orderService" class="documentation.examples.spring.OrderService">
    <property name="transactionHelper">
        <ref bean="transactionHelper"/>
    </property>
    <property name="dao">
        <ref bean="dao"/>
    </property>
    <property name="registry">
        <ref bean="basicRegistry"/>
    </property>

    <property name="orderReference" value="8675309"/>
</bean>
```

The following example shows how the shared properties have been moved into a parent bean, `baseService`.

```
<!--The 'baseService' bean contains common property definitions for services.-->
<bean id="baseService" abstract="true">
    <property name="transactionHelper">
        <ref bean="transactionHelper"/>
    </property>
    <property name="dao">
        <ref bean="dao"/>
    </property>
    <property name="registry">
         <ref bean="basicRegistry"/>
```

```
11      </property>
12  </bean>
13
14  <bean id="shippingService"
15          class="documentation.examples.spring.ShippingService"
16          parent="baseService">
17      <property name="shippingProvider" value="Federal Parcel Service"/>
18  </bean>
19
20  <bean id="orderService"
21          class="documentation.examples.spring.OrderService"
22          parent="baseService">
23      <property name="orderReference" value="8675309"/>
24  </bean>
```

References

- Spring Framework Reference Documentation 3.0: 3.4.2.2 References to other beans (collaborators).

Ensure that each property in a Spring bean definition has a matching setter

Category: Important > Spring

Description: Not declaring a setter for a property that is defined in a Spring XML file causes a compilation error.

The absence of a matching setter method for a property that is defined in a Spring XML bean causes a validation error when the project is compiled.

Recommendation

Ensure that there is a setter method in the bean file that matches the property name.

Example

The following example shows a bean file in which there is no match for the setter method that is in the class.

```
<bean id="contentService" class="documentation.examples.spring.ContentService">
    <!--BAD: The setter method in the class is 'setHelper', so this property
             does not match the setter method.-->
    <property name="transactionHelper">
        <ref bean="transactionHelper"/>
    </property>
</bean>
```

This is the bean class.

```
// bean class
public class ContentService {
    private TransactionHelper helper;

    // This method does not match the property in the bean file.
    public void setHelper(TransactionHelper helper) {
        this.helper = helper;
    }
}
```

The property `transactionHelper` should instead have the name `helper`.

References

- Spring Framework Reference Documentation 3.0: 3.4.1.2 Setter-based dependency injection.

Put 'import' statements before Spring bean definitions

Category: Important > Spring

Description: Putting 'import' statements before bean definitions in a Spring bean configuration file makes it easier to immediately see all the file's dependencies.

Putting `import` statements at the top of Spring XML bean definition files is good practice because they give a quick summary of the file's dependencies, and can even be used to document the general architecture of a system.

Recommendation

Make sure that all `import` statements are at the top of the `<beans>` section of a Spring XML bean definition file.

Example

The following example shows a `<beans>` section of a Spring XML bean definition file in which an `import` statement is in the middle, and a `<beans>` section in which all the `import` statements are at the top.

```
1  <beans>
2      <import resource="services.xml"/>
3
4      <bean id="bean1" class="..."/>
5      <bean id="bean2" class="..."/>
6
7      <!--AVOID: Imports in the middle of a bean configuration make it difficult
8          to immediately determine the dependencies of the configuration-->
9      <import resource="resources/messageSource.xml"/>
10
11      <bean id="bean3" class="..."/>
12      <bean id="bean4" class="..."/>
13 </beans>
14
15
16 <beans>
17     <!--GOOD: Having the imports at the top immediately gives an idea of
18         what the dependencies of the configuration are-->
19     <import resource="services.xml"/>
20     <import resource="resources/messageSource.xml"/>
21
22     <bean id="bean1" class="..."/>
23     <bean id="bean2" class="..."/>
24     <bean id="bean3" class="..."/>
25     <bean id="bean4" class="..."/>
26 </beans>
```

References

- Spring Framework Reference Documentation 3.0: 3.2.2.1 Composing XML-based configuration metadata.

Use 'id' instead of 'name' to name a Spring bean

Category: Important > Spring

Description: Using 'id' instead of 'name' to name a Spring bean enables the XML parser to perform additional checks.

To name a Spring bean, it is best to use the `id` attribute instead of the `name` attribute. Using the `id` attribute enables the XML parser to perform additional checks (for example, checking if the `id` in a `ref` attribute is an actual `id` of an XML element).

Recommendation

Use the `id` attribute instead of the `name` attribute when naming a bean.

Example

In the following example, the `dao` bean is shown using the `name` attribute, which allows a typo to go undetected because the XML parser does not check `name`. In contrast, using the `id` attribute allows the XML parser to catch the typo.

```
1  <!--AVOID: Using the 'name' attribute disables checking of bean references at XML parse time-->
2  <bean name="dao" class="documentation.examples.spring.DAO"/>
3
4  <bean id="orderService" class="documentation.examples.spring.OrderService">
5      <!--The XML parser cannot catch this typo-->
6      <property name="dao" ref="da0"/>
7  </bean>
8
9
10 <!--GOOD: Using the 'id' attribute enables checking of bean references at XML parse time-->
11 <bean id="dao" class="documentation.examples.spring.DAO"/>
12
13 <bean id="orderService" class="documentation.examples.spring.OrderService">
14     <!--The XML parser can catch this typo-->
15     <property name="dao" ref="da0"/>
16 </bean>
```

References

- Spring Framework Reference Documentation 3.0: 3.3.1 Naming beans.
- W3C: 3.3.1 Attribute Types.

Use a type name instead of an index number in a Spring 'constructor-arg' element

Category: Important > Spring

Description: Using a type name instead of an index number in a Spring 'constructor-arg' element improves readability.

Using type matching instead of index matching in a Spring `constructor-arg` element produces a more readable bean definition and is less vulnerable to being broken by a change to the constructor of the bean's underlying class. Index matching should be used only if type matching is not sufficient to remove ambiguity in the constructor arguments.

Recommendation

The bean definition's `constructor-arg` elements should use type matching instead of index matching.

Example

The following example shows a bean, `billingService1`, whose `constructor-arg` elements use index matching, and an improved version of the bean, `billingService2`, whose `constructor-arg` elements use type matching.

```
1  <!--AVOID: Using explicit constructor indices makes the bean configuration
2             vulnerable to changes to the constructor-->
3  <bean id="billingService1" class="documentation.examples.spring.BillingService">
4      <constructor-arg index="0" value="John Doe"/>
5      <constructor-arg index="1" ref="dao"/>
6  </bean>
7
8  <!--GOOD: Using type matching makes the bean configuration more robust to changes in
9      the constructor-->
10 <bean id="billingService2" class="documentation.examples.spring.BillingService">
11     <constructor-arg ref="dao"/>
12     <constructor-arg type="java.lang.String" value="Jane Doe"/>
13 </bean>
```

References

- Spring Framework Reference Documentation 3.0: 3.4.1.1 Constructor-based dependency injection.
- ONJava: Twelve Best Practices For Spring XML Configurations.

Use local references when referring to Spring beans in the same file

Category: Important > Spring

Description: Using local references when referring to Spring beans in the same file allows reference errors to be detected during XML parsing.

If at all possible, refer to Spring beans in the same XML file using local references, that is `<idref local="targetBean">`. This requires that the bean being referenced is in the same XML file, and is named using the `id` attribute. Using local references has the advantage of allowing reference errors to be detected during XML parsing, instead of during deployment or instantiation.

From the Spring Framework Reference documentation on `idref` elements:

> *[Using the `idref` tag in a `property` element] is preferable to [using the bean name in the property's `value` attribute], because using the `idref` tag allows the container to validate at deployment time that the referenced, named bean actually exists. In the second variation, no validation is performed on the value that is passed to the [`name`] property of the client bean. Typos are only discovered (with most likely fatal results) when the client bean is actually instantiated. If the client bean is a prototype bean, this typo and the resulting exception may only be discovered long after the container is deployed.*
>
> *Additionally, if the referenced bean is in the same XML unit, and the bean name is the bean `id`, you can use the `local` attribute, which allows the XML parser itself to validate the bean `id` earlier, at XML document parse time.*

Recommendation

Use a local `idref` when referring to beans in the same XML file. This allows errors to be detected earlier, at XML parse time rather than during instantiation.

Example

In the following example, the `shippingService` bean is shown using the `ref` element, which cannot be checked by the XML parser. The `orderService` bean is shown using the `idref` element, which allows the XML parser to find any errors at parse time.

```
<beans>
    <bean id="shippingService" class="documentation.examples.spring.ShippingService">
        <!--AVOID: This form of reference cannot be checked by the XML parser-->
        <property name="dao">
            <ref bean="dao"/>
        </property>
    </bean>

    <bean id="orderService" class="documentation.examples.spring.OrderService">
         <!--GOOD: This form of reference allows the XML parser to find any errors at parse time-->
         <property name="dao">
             <idref local="dao"/>
         </property>
     </bean>

     <bean id="dao" class="documentation.examples.spring.DAO"/>
</beans>
```

References

- Spring Framework Reference Documentation 3.0: 3.4.2.1 Straight values (primitives, Strings, and so on).

Use setter injection instead of constructor injection when using Spring

Category: Important > Spring

Description: When using the Spring Framework, using setter injection instead of constructor injection is more flexible, especially when several properties are optional.

When you use the Spring Framework, using setter injection instead of constructor injection is more flexible, particularly for Spring beans with a large number of optional properties. Constructor injection should be used only on required bean properties; using constructor injection on optional bean properties requires a large number of constructors to handle different combinations of properties.

Although the generally accepted best practice is to use constructor injection for mandatory dependencies, and setter injection for optional dependencies, the `@Required` annotation allows you to forgo constructor injection completely. Using the `@Required` annotation on a setter method makes the framework check that a dependency is injected using that method.

Recommendation

Use setter injection in bean configurations, marking required properties with the `@Required` annotation. It makes it easier to accommodate a large number of optional properties, and makes the bean more flexible by allowing for re-injection of dependencies.

Example

The following example shows a bean that is defined using constructor injection. The bean configuration is followed by the class definition.

```
<!--AVOID: Using constructor args for optional parameters requires one constructor per combination
of properties. This leads to a large number of constructors in the bean class.-->
<bean id="chart1" class="documentation.examples.spring.WrongChartMaker">
    <constructor-arg ref="customTrend"/>
    <constructor-arg ref="customAxis"/>
</bean>
```

```
// Class for bean 'chart1'
public class WrongChartMaker {
    private AxisRenderer axisRenderer = new DefaultAxisRenderer();
    private TrendRenderer trendRenderer = new DefaultTrendRenderer();

    public WrongChartMaker() {}

    // Each combination of the optional parameters must be represented by a constructor.
    public WrongChartMaker(AxisRenderer customAxisRenderer) {
        this.axisRenderer = customAxisRenderer;
     }

     public WrongChartMaker(TrendRenderer customTrendRenderer) {
        this.trendRenderer = customTrendRenderer;
     }

     public WrongChartMaker(AxisRenderer customAxisRenderer,
                            TrendRenderer customTrendRenderer) {
        this.axisRenderer = customAxisRenderer;
        this.trendRenderer = customTrendRenderer;
     }
 }
```

The following example shows how the same bean can be defined using setter injection instead. Again, the bean

configuration is followed by the class definition.

```
<!--GOOD: Using setter injection requires only one setter for each property.-->
<bean id="chart2" class="documentation.examples.spring.ChartMaker">
    <property name="axisRenderer" ref="customAxis"/>
</bean>
```

```
// Class for bean 'chart2'
public class ChartMaker {
    private AxisRenderer axisRenderer = new DefaultAxisRenderer();
    private TrendRenderer trendRenderer = new DefaultTrendRenderer();

    public ChartMaker() {}

    public void setAxisRenderer(AxisRenderer axisRenderer) {
        this.axisRenderer = axisRenderer;
     }

     public void setTrendRenderer(TrendRenderer trendRenderer) {
         this.trendRenderer = trendRenderer;
     }
}
```

References

- Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern.
- ONJava: Twelve Best Practices for Spring XML Configurations.
- Spring Framework Reference Documentation 3.0: 3.4.1.1 Constructor-based dependency injection, 3.4.1.2 Setter-based dependency injection.
- SpringSource: Setter injection versus constructor injection and the use of @Required.

Use shortcut forms in Spring bean definitions

Category: Important > Spring

Description: Using shortcut forms may make a Spring XML configuration file less cluttered.

Shortcut forms, introduced in Spring 1.2, allow nested `value` elements to instead be defined as attributes in the enclosing `property` entry. This leads to shorter XML bean configurations that are easier to read.

Recommendation

When possible, use the shortcut form for defining bean property values.

Note that this does *not* apply to `idref` elements, which are the preferred form of referring to another bean. These do not have a shortcut form that can still be checked by the XML parser.

Example

The following example shows how a bean that is defined using shortcut forms is more concise than the same bean defined using nested `value` elements.

```
<!--AVOID: Using nested 'value' elements can make the configuration file difficult to read-->
<bean id="serviceRegistry" class="documentation.examples.spring.ServiceRegistry">
    <constructor-arg type="java.lang.String">
        <value>main_service_registry</value>
    </constructor-arg>
    <property name="description">
        <value>Top-level registry for services</value>
    </property>
    <property name="serviceMap">
         <map>
             <entry>
                 <key>
                     <value>orderService</value>
                 </key>
                 <value>com.foo.bar.OrderService</value>
             </entry>
             <entry>
                 <key>
                     <value>billingService</value>
                 </key>
                 <value>com.foo.bar.BillingService</value>
             </entry>
         </map>
    </property>
</bean>


<!--GOOD: Shortcut forms (Spring 1.2) result in more concise bean definitions-->
<bean id="serviceRegistry" class="documentation.examples.spring.ServiceRegistry">
    <constructor-arg type="java.lang.String" value="main_service_registry"/>
    <property name="description" value="Top-level registry for services"/>
    <property name="serviceMap">
        <map>
            <entry key="orderService" value="com.foo.bar.OrderService"/>
            <entry key="billingService" value="com.foo.bar.BillingService"/>
        </map>
    </property>
</bean>
```

References

- ONJava: Twelve Best Practices for Spring XML Configurations.
- Spring Framework Reference Documentation 3.0: 3.4.2.1 Straight values (primitives, Strings, and so on).

Strings (1)

- Avoid calling 'toString' on a string
- Avoid calling 'toUpperCase()' or 'toLowerCase()' without specifying the locale

Avoid calling 'toString' on a string

Category: Important > Strings (1)

Description: Calling 'toString' on a string is redundant.

There is no need to call `toString` on a `String` because it just returns the object itself. From the Java API Specification entry for `String.toString()`:

> *`public String toString()`*
> *This object (which is already a string!) is itself returned.*

Recommendation

Do not call `toString` on a `String` object.

Example

The following example shows an unnecessary call to `toString` on the string `name`.

```
1 public static void main(String args[]) {
2     String name = "John Doe";
3 
4     // BAD: Unnecessary call to 'toString' on 'name'
5     System.out.println("Hi, my name is " + name.toString());
6 
7     // GOOD: No call to 'toString' on 'name'
8     System.out.println("Hi, my name is " + name);
9 }
```

References

- Java 6 API Specification: String.toString().

Avoid calling 'toUpperCase()' or 'toLowerCase()' without specifying the locale

Category: Important > Strings (1)

Description: Calling 'String.toUpperCase()' or 'String.toLowerCase()' without specifying the locale may cause unexpected results for certain default locales.

The parameterless versions of `String.toUpperCase()` and `String.toLowerCase()` use the default locale of the Java Virtual Machine when transforming strings. This can cause unexpected behavior for certain locales.

Recommendation

Use the corresponding methods with explicit locale parameters to ensure that the results are consistent across all locales. For example:

```
System.out.println("I".toLowerCase(java.util.Locale.ENGLISH));
```

prints `i`, regardless of the default locale.

Example

In the following example, the calls to the parameterless functions may return different strings for different locales. For example, if the default locale is ENGLISH, the function `toLowerCase()` converts a capital `I` to `i`; if the default locale is TURKISH, the function `toLowerCase()` converts a capital `I` to the Unicode Character "Latin small letter dotless i" (U+0131) (Turkish HTML Codes, Unicode Hexadecimal & HTML Names).

To ensure that an English string is returned, regardless of the default locale, the example shows how to call `toLowerCase` and pass `locale.ENGLISH` as the argument. (This assumes that the text is English. If the text is Turkish, you should pass `locale.TURKISH` as the argument.)

```
1  public static void main(String args[]) {
2      String phrase = "I miss my home in Mississippi.";
3
4      // AVOID: Calling 'toLowerCase()' or 'toUpperCase()'
5      // produces different results depending on what the default locale is.
6      System.out.println(phrase.toUpperCase());
7      System.out.println(phrase.toLowerCase());
8
9      // GOOD: Explicitly setting the locale when calling 'toLowerCase()' or
10      // 'toUpperCase()' ensures that the resulting string is
11      // English, regardless of the default locale.
12      System.out.println(phrase.toLowerCase(Locale.ENGLISH));
13      System.out.println(phrase.toUpperCase(Locale.ENGLISH));
14 }
```

References

- Java API Documentation: String.toUpperCase().

Swing

- Avoid calling Swing methods from a thread other than the event-dispatching thread
- Ensure that event handler overrides have exactly the right name

Avoid calling Swing methods from a thread other than the event-dispatching thread

Category: Important > Swing

Description: Calling Swing methods from a thread other than the event-dispatching thread may result in multi-threading errors.

Because Swing components are not thread-safe (that is, they do not support concurrent access from multiple threads), Swing has a rule that states that method calls on Swing components that have been *realized* (see below) must be made from a special thread known as the *event-dispatching thread*. Failure to observe this rule may result in multiple threads accessing a Swing component concurrently, with the potential for deadlocks, race conditions and other errors related to multi-threading.

A component is considered *realized* if its `paint` method has been, or could be, called at this point. Realization is triggered according to the following rules:

- A top-level window is realized if `setVisible(true)`, `show` or `pack` is called on it.
- Realizing a container realizes the components it contains.

There are a few exceptions to the rule. These are documented more fully in [The Swing Connection] but the key exceptions are:

- It is safe to call the `repaint`, `revalidate` and `invalidate` methods on a Swing component from any thread.
- It is safe to add and remove listeners from any thread. Therefore, any method of the form `add*Listener` or `remove*Listener` is thread-safe.

Recommendation

Ensure that method calls on Swing components are made from the event-dispatching thread. If you need to call a method on a Swing component from another thread, you must do so using the event-dispatching thread. Swing provides a `SwingUtilities` class that you can use to ask the event-dispatching thread to run arbitrary code on your components, by calling one of two methods. Each method takes a `Runnable` as its only argument:

- `SwingUtilities.invokeLater` asks the event-dispatching thread to run some code and then immediately returns (that is, it is non-blocking). The code is run at some indeterminate time in the future, but the thread that calls `invokeLater` does not wait for it.
- `SwingUtilities.invokeAndWait` asks the event-dispatching thread to run some code and then waits for it to complete (that is, it is blocking).

Example

In the following example, there is a call from the main thread to a method, `setTitle`, on the `MyFrame` object after the object has been realized by the `setVisible(true)` call. This represents an unsafe call to a Swing method from a thread other than the event-dispatching thread.

```
1  class MyFrame extends JFrame {
2      public MyFrame() {
3          setSize(640, 480);
4          setTitle("BrokenSwing");
5      }
6  }
7
8  public class BrokenSwing {
9      private static void doStuff(MyFrame frame) {
10         // BAD: Direct call to a Swing component after it has been realized
11         frame.setTitle("Title");
12     }
```

```
13
14      public static void main(String[] args) {
15          MyFrame frame = new MyFrame();
16          frame.setVisible(true);
17          doStuff(frame);
18      }
19  }
```

In the following modified example, the call to `setTitle` is instead called from within a call to `invokeLater`.

```
1  class MyFrame extends JFrame {
2      public MyFrame() {
3          setSize(640, 480);
4          setTitle("BrokenSwing");
5      }
6  }
7
8  public class GoodSwing {
9      private static void doStuff(final MyFrame frame) {
10         SwingUtilities.invokeLater(new Runnable() {
11             public void run() {
12                 // GOOD: Call to Swing component made via the
13                 // event-dispatching thread using 'invokeLater'
14                 frame.setTitle("Title");
15             }
16         });
17     }
18
19     public static void main(String[] args) {
20         MyFrame frame = new MyFrame();
21         frame.setVisible(true);
22         doStuff(frame);
23     }
24 }
```

References

- D. Flanagan, *Java Foundation Classes in a Nutshell*, p.28. O'Reilly, 1999.
- Java Developer's Journal: Building Thread-Safe GUIs with Swing.
- The Java Tutorials: Concurrency in Swing.
- The Swing Connection: Threads and Swing.

Ensure that event handler overrides have exactly the right name

Category: Important > Swing

Description: In a class that extends a Swing or Abstract Window Toolkit event adapter, an event handler that does not have exactly the same name as the event handler that it overrides means that the overridden event handler is not called.

Event adapters in Swing (and Abstract Window Toolkit) provide a convenient way for programmers to implement event listeners. However, care must be taken to get the names of the overridden methods right, or the event handlers will not be called.

In Depth

The event listener interfaces in Swing (and Abstract Window Toolkit) have many methods. For example, `java.awt.event.MouseListener` is defined as follows:

```
1 public interface MouseListener extends EventListener {
2     public abstract void mouseClicked(MouseEvent);
3     public abstract void mousePressed(MouseEvent);
4     public abstract void mouseReleased(MouseEvent);
5     public abstract void mouseEntered(MouseEvent);
6     public abstract void mouseExited(MouseEvent);
7 }
```

The large number of methods can make such interfaces lengthy and tedious to implement, especially because it is rare that all of the methods need to be overridden. It is much more common that you need to override only one method, for example the `mouseClicked` event.

For this reason, Swing supplies *adapter* classes that provide default, blank implementations of interface methods. An example is `MouseAdapter`, which provides default implementations for the methods in `MouseListener`, `MouseWheelListener` and `MouseMotionListener`. (Note that an adapter often implements multiple interfaces to avoid a large number of small adapter classes.) This makes it easy for programmers to implement just the methods they need from a given interface.

Unfortunately, adapter classes are also a source of potential defects. Because the `@Override` annotation is not compulsory, it is very easy for programmers not to use it and then mistype the name of the method. This introduces a new method rather than implementing the relevant event handler.

Recommendation

Ensure that any overriding methods have exactly the same name as the overridden method.

Example

In the following example, the programmer has tried to implement the `mouseClicked` function but has misspelled the function name. This makes the function inoperable but the programmer gets no warning about this from the compiler.

```
1 add(new MouseAdapter() {
2     public void mouseClickd(MouseEvent e) {
3         // ...
4     }
5 });
```

In the following modified example, the function name is spelled correctly. It is also preceded by the `@Override` annotation, which will cause the compiler to display an error if there is not a function of the same name to be

overridden.

```
add(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        // ...
    }
});
```

References

- D. Flanagan, *Java Foundation Classes in a Nutshell*, Chapter 26. O'Reilly, 1999.
- Java Platform, Standard Edition 7, API Specification: Annotation Type Override.
- The Java Tutorials: Event Adapters.

Types (2)

- Avoid naming a type variable the same as another type that is in scope
- Avoid trying to extend a final type using a wildcard
- Do not call a varargs method with an ambiguous argument

Avoid naming a type variable the same as another type that is in scope

Category: Important > Types (2)

Description: A type variable with the same name as another type that is in scope can cause the two types to be confused.

Type shadowing occurs if two types have the same name but one is defined within the scope of the other. This can arise if you introduce a type variable with the same name as an imported class.

Type shadowing may cause the two types to be confused, which can lead to various problems.

Recommendation

Name the type variable so that its name does not clash with the imported class.

Example

In the following example, the type `java.util.Map.Entry` is imported at the top of the file, but the class `Mapping` is defined with two type variables, `Key` and `Entry`. Uses of `Entry` within the `Mapping` class refer to the type variable, and not the imported interface. The type variable therefore shadows `Map.Entry`.

```
1 import java.util.Map;
2 import java.util.Map.Entry;
3
4 class Mapping<Key, Entry>  // The type variable 'Entry' shadows the imported interface 'Entry'.
5 {
6     // ...
7 }
```

To fix the code, the type variable `Entry` on line 4 should be renamed.

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Language Specification: 6.4 Shadowing and Obscuring.

Avoid trying to extend a final type using a wildcard

Category: Important > Types (2)

Description: If 'C' is a final class, a type bound such as '? extends C' is confusing because it implies that 'C' has subclasses, but a final class has no subclasses.

A type wildcard with an `extends` clause (for example `? extends String`) implicitly suggests that a type (in this case `String`) has subclasses. If the type in the `extends` clause is final, the code is confusing because a final class cannot have any subclasses. The only type that satisfies `? extends String` is `String`.

Recommendation

To make the code more readable, omit the wildcard to leave just the final type.

Example

In the following example, a wildcard is used to refer to any type that is a subclass of `String`.

```
1 class Printer
2 {
3     void print(List<? extends String> strings) {  // Unnecessary wildcard
4         for (String s : strings)
5             System.out.println(s);
6     }
7 }
```

However, because `String` is declared `final`, it does not have any subclasses. Therefore, it is clearer to replace `? extends String` with `String`.

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Language Specification: 4.5.1 Type Arguments and Wildcards, 8.1.1.2 final Classes.

Do not call a varargs method with an ambiguous argument

Category: Important > Types (2)

Description: Calling a varargs method where is is unclear whether the arguments should be interpreted as a list of arguments or as a single argument, may lead to compiler-dependent behavior.

A variable arity method, commonly known as a varargs method, may be called with different numbers of arguments. For example, the method `sum(int... values)` may be called in all of the following ways:

- `sum()`
- `sum(1)`
- `sum(1,2,3)`
- `sum(new int[] { 1, 2, 3 })`

When a method `foo(T... x)` is called with an argument that is neither `T` nor `T[]`, but the argument can be cast as either, the choice of which type the argument is cast as is compiler-dependent.

Recommendation

When a variable arity method, for example `m(T... ts)`, is called with a single argument (for example `m(arg)`), the type of the argument should be either `T` or `T[]` (insert a cast if necessary).

Example

In the following example, the calls to `length` do not pass an argument of the same type as the parameter of `length`, which is `Object` or an array of `Object`. Therefore, when the program is compiled with javac, the output is:

```
3
2
```

When the program is compiled with a different compiler, for example the default compiler for some versions of Eclipse, the output may be:

```
3
1
```

```
1  class InexactVarArg
2  {
3      private static void length(Object... objects) {
4          System.out.println(objects.length);
5      }
6
7      public static void main(String[] args) {
8          String[] words = { "apple", "banana", "cherry" };
9          String[][] lists = { words, words };
10          length(words);  // BAD: Argument does not clarify
11          length(lists);  // which parameter type is used.
12      }
13 }
```

To fix the code, `length(words)` should be replaced by either of the following:

- `length((Object) words)`
- `length((Object[]) words)`

Similarly, `length(lists)` should be replaced by one of the following:

- `length((Object) lists)`
- `length((Object[]) lists)`

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Language Specification: 8.4.1 Formal Parameters, 15.12.4.2 Evaluate Arguments.

Useless Code

- Avoid futile assignments to a local variable
- Avoid local variables that are never read
- Avoid redundant types
- Avoid unnecessary 'import' statements
- Avoid unnecessary casts
- Avoid unused fields
- Avoid unused labels
- Ensure that fields are explicitly initialized
- Ensure that interface methods are compatible with 'java.lang.Object'

Avoid futile assignments to a local variable

Category: Important > Useless Code

Description: An assignment to a local variable that is not used before a further assignment is made has no effect.

A value is assigned to a local variable, but whenever the variable is subsequently read, there has been at least one other assignment to that variable. This means that the original assignment is suspect, because the state of the local variable that it creates is never used.

Recommendation

Ensure that you check the control and data flow in the method carefully. If a value is really not needed, consider omitting the assignment. Be careful, though: if the right-hand side has a side-effect (like performing a method call), it is important to keep this to preserve the overall behavior.

Example

In the following example, the value assigned to `result` on line 5 is always overwritten (line 6) before being read (line 7). This is a strong indicator that there is something wrong. By examining the code, we can see that the loop in lines 3-5 seems to be left over from an old way of storing the list of persons, and line 6 represents the new (and better-performing) way. Consequently, we can delete lines 3-5 while preserving behavior.

```
1 Person find(String name) {
2     Person result;
3     for (Person p : people.values())
4         if (p.getName().equals(name))
5             result = p;  // Redundant assignment
6     result = people.get(name);
7     return result;
```

Avoid local variables that are never read

Category: Important > Useless Code

Description: A local variable that is never read is redundant.

A local variable that is never read is useless.

As a matter of good practice, there should be no unused or useless code. It makes the program more difficult to understand and maintain, and can waste a programmer's time.

Recommendation

This rule applies to variables that are never used as well as variables that are only written to but never read. In both cases, ensure that no operations are missing that would use the local variable. If appropriate, simply remove the declaration. However, if the variable is written to, ensure that any side-effects in the assignments are retained. (For further details, see the example.)

Example

In the following example, the local variable `oldQuantity` is assigned a value but never read. In the fixed version of the example, the variable is removed but the call to `items.put` in the assignment is retained.

```
1  // Version containing unread local variable
2  public class Cart {
3      private Map<Item, Integer> items = ...;
4      public void add(Item i) {
5          Integer quantity = items.get(i);
6          if (quantity = null)
7              quantity = 1;
8          else
9              quantity++;
10         Integer oldQuantity = items.put(i, quantity);  // AVOID: Unread local variable
11     }
12 }
13
14 // Version with unread local variable removed
15 public class Cart {
16     private Map<Item, Integer> items = ...;
17     public void add(Item i) {
18         Integer quantity = items.get(i);
19         if (quantity = null)
20             quantity = 1;
21         else
22             quantity++;
23         items.put(i, quantity);
24     }
25 }
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Avoid redundant types

Category: Important > Useless Code

Description: A non-public class or interface that is not used anywhere in the program wastes programmer resources.

A non-public class or interface that is not used anywhere in the program may cause a programmer to waste time and effort maintaining and documenting it.

Recommendation

Ensure that redundant types are removed from the program.

References

- Wikipedia: Unreachable code.

Avoid unnecessary 'import' statements

Category: Important > Useless Code

Description: A redundant 'import' statement introduces unnecessary and undesirable dependencies.

An `import` statement that is not necessary (because no part of the file that it is in uses any imported type) should be avoided. Although importing too many types does not affect performance, redundant `import` statements introduce unnecessary and undesirable dependencies in the code. If an imported type is renamed or deleted, the source code cannot be compiled because the `import` statement cannot be resolved.

Unnecessary `import` statements are often an indication of incomplete refactoring.

Recommendation

Avoid including an `import` statement that is not needed. Many modern IDEs have automated support for doing this, typically under the name 'Organize imports'. This sorts the `import` statements and removes any that are not used, and it is good practice to run such a command before every commit.

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Avoid unnecessary casts

Category: Important > Useless Code

Description: Casting an object to its own type is unnecessary.

A cast is unnecessary if the type of the operand is already the same as the type that is being cast to.

Recommendation

Avoid including unnecessary casts.

Example

In the following example, casting `i` to an `Integer` is not necessary. It is already an `Integer`.

```
1 public class UnnecessaryCast {
2     public static void main(String[] args) {
3         Integer i = 23;
4         Integer j = (Integer)i;  // AVOID: Redundant cast
5     }
6 }
```

To fix the code, delete `(Integer)` on the right-hand side of the assignment on line 4.

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Avoid unused fields

Category: Important > Useless Code

Description: A field that is never used is probably unnecessary.

A field that is neither public nor protected and never accessed is typically a leftover from old refactorings or a sign of incomplete or pending code changes.

This rule does not apply to a field in a serializable class because it may be accessed during serialization and deserialization.

Recommendation

If an unused field is a leftover from old refactorings, you should just remove it. If it indicates incomplete or pending code changes, finish making the changes and remove the field if it is not needed.

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Avoid unused labels

Category: Important > Useless Code

Description: An unused label for a loop or 'switch' statement is either redundant or indicates incorrect 'break' or 'continue' statements.

Loop and `switch` statements can be labeled. These labels can serve as targets for `break` or `continue` statements, to specify which loop or `switch` statement they refer to.

Apart from serving as such jump targets, the labels have no effect on program behavior, which means that having an unused label is suspicious.

Recommendation

If the label is used to document the intended behavior of a loop or `switch` statement, remove it. It is better to use comments for this purpose. However, an unused label may indicate that something is wrong: that some of the nested `break` or `continue` statements should be using the label. In this case, the current control flow is probably wrong, and you should adjust some jumps to use the label after checking the desired behavior.

Example

The following example uses a loop and a nested loop to check whether any of the currently active shopping carts contains a particular item. On line 4, the `carts:` label is unused. Inspecting the code, we can see that the `break` statement on line 10 is inefficient because it only breaks out of the nested loop. It could in fact break out of the outer loop, which should improve performance in common cases. By changing the statement on line 10 to read `break carts;`, the label is no longer unused and we improve the code.

```
1  public class WebStore {
2      public boolean itemIsBeingBought(Item item) {
3          boolean found = false;
4          carts:  // AVOID: Unused label
5          for (int i = 0; i < carts.size(); i++) {
6              Cart cart = carts.get(i);
7              for (int j = 0; j < cart.numItems(); j++) {
8                  if (item.equals(cart.getItem(j))) {
9                      found = true;
10                      break;
11                 }
12             }
13         }
14         return found;
15     }
16 }
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.

Ensure that fields are explicitly initialized

Category: Important > Useless Code

Description: A field that is never assigned a value (except possibly 'null') just returns the default value when it is read.

It is good practice to initialize every field in a constructor explicitly. A field that is never assigned any value (except possibly `null`) just returns the default value when it is read, or throws a `NullPointerException`.

Recommendation

A field whose value is always `null` (or the corresponding default value for primitive types, for example `0`) is not particularly useful. Ensure that the code contains an assignment or initialization for each field. To help satisfy this rule, it is good practice to explicitly initialize every field in the constructor, even if the default value is acceptable.

If the field is genuinely never expected to hold a non-default value, check the statements that read the field and ensure that they are not making incorrect assumptions about the value of the field. Consider completely removing the field and rewriting the statements that read it, as appropriate.

Example

In the following example, the private field `name` is not initialized in the constructor (and thus is implicitly set to `null`), but there is a getter method to access it.

```
1  class Person {
2      private String name;
3      private int age;
4
5      public Person(String name, int age) {
6          this.age = age;
7      }
8
9      public String getName() {
10          return name;
11      }
12
13      public int getAge() {
14          return age;
15      }
16  }
```

Therefore, the following code throws a `NullPointerException`:

```
1  Person p = new Person("Arthur Dent", 30);
2  int l = p.getName().length();
```

To fix the code, `name` should be initialized in the constructor by adding the following line: `this.name = name;`

Ensure that interface methods are compatible with 'java.lang.Object'

Category: Important > Useless Code

Description: An interface method that is incompatible with a protected method on 'java.lang.Object' means that the interface cannot be implemented.

An interface that contains methods whose return types clash with protected methods on `java.lang.Object` can never be implemented, because methods cannot be overloaded based simply on their return type.

Recommendation

If the interface is useful, name methods so that they do not clash with methods in `Object`. Otherwise you should delete the interface.

Example

In the following example, the interface `I` is useless because the `clone` method must return type `java.lang.Object`:

```
1 interface I {
2     int clone();
3 }
4
5 class C implements I {
6     public int clone() {
7         return 23;
8     }
9 }
```

Any attempt to implement the interface produces an error:

```
InterfaceCannotBeImplemented.java:6: clone() in C cannot override
  clone() in java.lang.Object; attempting to use incompatible return
  type
found   : int
required: java.lang.Object
  public int clone() {
             ^
1 error
```

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Language Specification, Third Edition: 9.2 Interface Members.

Advisory

Rules in this category represent good practice. Violations of these rules are allowed but not recommended.

Rule types:

- Declarations (1)
- Deprecated Code
- Documentation
- Java Objects (1)
- Naming (1)
- Statements
- Types (1)

Declarations (1)

- Avoid implicit imports
- Declare immutable fields 'final'
- Do not make mutable fields public
- Use '@Override' annotation when overriding a method

Avoid implicit imports

Category: Advisory > Declarations (1)

Description: An implicit import obscures the dependencies of a file and may cause confusing compile-time errors.

Imports can be categorized as *explicit* (for example `import java.util.List;`) or *implicit* (also known as 'on-demand', for example `import java.util.*;`):

- Implicit imports give access to all visible types in the type (or package) that precedes the ".*"; types imported in this way never shadow other types.
- Explicit imports give access to just the named type; they can shadow other types that would normally be visible through an implicit import, or through the normal package visibility rules.

It is often considered bad practice to use implicit imports. The only advantage to doing so is making the code more concise, and there are a number of disadvantages:

- The exact dependencies of a file are not visible at a glance.
- Confusing compile-time errors can be introduced if a type name is used that could originate from several implicit imports.

Recommendation

For readability, it is recommended to use explicit imports instead of implicit imports. Many modern IDEs provide automatic functionality to help achieve this, typically under the name "Organize imports". They can also fold away the import declarations, and automatically manage imports: adding them when a particular type is auto-completed by the editor, and removing them when they are not necessary. This functionality makes implicit imports mainly redundant.

Example

The following example uses implicit imports. This means that it is not clear to a programmer where the `List` type on line 5 is imported from.

```
1 import java.util.*;  // AVOID: Implicit import statements
2 import java.awt.*;
3
4 public class Customers {
5     public List getCustomers() {  // Compiler error: 'List' is ambiguous.
6         ...
7     }
8 }
```

To improve readability, the implicit imports should be replaced by explicit imports. For example, `import java.util.*;` should be replaced by `import java.util.List;` on line 1.

References

- Java Language Specification: 6.4.1 Shadowing, 7.5.2 Type-Import-on-Demand Declarations.

Declare immutable fields 'final'

Category: Advisory > Declarations (1)

Description: A field of immutable type that is assigned to only in a constructor or static initializer of its declaring type, but is not declared 'final', may lead to defects and makes code less readable.

A field of immutable type that is not declared `final`, but is assigned to only in a constructor or static initializer of its declaring type, may lead to defects and makes code less readable. This is because other parts of the code may be based on the assumption that the field has a constant value, and a later modification, which includes an assignment to the field, may invalidate this assumption.

Recommendation

If a field of immutable type is assigned to only during class or instance initialization, you should usually declare it `final`. This forces the compiler to verify that the field value cannot be changed subsequently, which can help to avoid defects and increase code readability.

References

- Java Language Specification: 4.12.4 final Variables, 8.3.1.2 final Fields.

Do not make mutable fields public

Category: Advisory > Declarations (1)

Description: A non-constant field that is not declared 'private', but is not accessed outside of its declaring type, may decrease code maintainability.

A non-final or non-static field that is not declared `private`, but is not accessed outside of its declaring type, may decrease code maintainability. This is because a field that is accessible from outside the class that it is declared in tends to restrict the class to a particular implementation.

Recommendation

In the spirit of encapsulation, it is generally advisable to choose the most restrictive access modifier (`private`) for a field, unless there is a good reason to increase its visibility.

References

- J. Bloch, *Effective Java (second edition)*, Item 13. Addison-Wesley, 2008.
- The Java Tutorials: Controlling Access to Members of a Class.

Use '@Override' annotation when overriding a method

Category: Advisory > Declarations (1)

Description: A method that overrides a method in a superclass but does not have an 'Override' annotation cannot take advantage of compiler checks, and makes code less readable.

Java enables you to annotate methods that are intended to override a method in a superclass. Compilers are required to generate an error if such an annotated method does not override a method in a superclass, which provides increased protection from potential defects. An annotated method also improves code readability.

Recommendation

Add an `@Override` annotation to a method that is intended to override a method in a superclass.

Example

In the following example, `Triangle.getArea` overrides `Rectangle.getArea`, so it is annotated with `@Override`.

```
1  class Rectangle
2  {
3      private int w = 10, h = 10;
4      public int getArea() {
5          return w * h;
6      }
7  }
8
9  class Triangle extends Rectangle
10 {
11     @Override  // Annotation of an overriding method
12     public int getArea() {
13         return super.getArea() / 2;
14     }
15 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 36. Addison-Wesley, 2008.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Platform, Standard Edition 6, API Specification: Annotation Type Override.
- The Java Tutorials: Predefined Annotation Types.

Deprecated Code

- Avoid using a deprecated method or constructor

Avoid using a deprecated method or constructor

Category: Advisory > Deprecated Code

Description: Using a method or constructor that has been marked as deprecated may be dangerous or fail to take advantage of a better method or constructor.

A method (or constructor) can be marked as deprecated using either the `@Deprecated` annotation or the `@deprecated` Javadoc tag. Using a method that has been marked as deprecated is bad practice, typically for one or more of the following reasons:

- The method is dangerous.
- There is a better alternative method.
- Methods that are marked as deprecated are often removed from future versions of an API. So using a deprecated method may cause extra maintenance effort when the API is upgraded.

Recommendation

Avoid using a method that has been marked as deprecated. Follow any guidance that is provided with the `@deprecated` Javadoc tag, which should explain how to replace the call to the deprecated method.

References

- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- Java Platform, Standard Edition 6, API Specification: Annotation Type Deprecated.
- Java SE Documentation: How and When To Deprecate APIs.

Documentation

- Include a Javadoc comment for each public class or interface
- Include a Javadoc comment for each public method or constructor
- Include a Javadoc tag for each exception thrown by a public method or constructor
- Include a Javadoc tag for each parameter of a public method or constructor
- Include a Javadoc tag for the return value of a public method or constructor

Include a Javadoc comment for each public class or interface

Category: Advisory > Documentation

Description: A public class or interface that does not have a Javadoc comment affects maintainability.

A public class or interface that does not have a Javadoc comment makes an API more difficult to understand and maintain.

Recommendation

Public classes and interfaces should be documented to make an API usable. For the purpose of code maintainability, it is also advisable to document non-public classes and interfaces.

Documentation for users of an API should be written using the standard Javadoc format. This can be accessed conveniently by users of an API from within standard IDEs, and can be transformed automatically into HTML format.

Example

The following example shows a good Javadoc comment, which clearly explains what the class does, its author, and version.

```
1 /**
2  * The Stack class represents a last-in-first-out stack of objects.
3  *
4  * @author  Joseph Bergin
5  * @version 1.0, May 2000
6  * Note that this version is not thread safe.
7  */
8 public class Stack {
9 // ...
```

References

- J. Bloch, *Effective Java (second edition)*, Item 44. Addison-Wesley, 2008.
- Help - Eclipse Platform: Java Compiler Javadoc Preferences.
- Java SE Documentation: How to Write Doc Comments for the Javadoc Tool, Requirements for Writing Java API Specifications.

Include a Javadoc comment for each public method or constructor

Category: Advisory > Documentation

Description: A public method or constructor that does not have a Javadoc comment affects maintainability.

A public method or constructor that does not have a Javadoc comment makes an API more difficult to understand and maintain.

Recommendation

Public methods and constructors should be documented to make an API usable. For the purpose of code maintainability, it is also advisable to document non-public methods and constructors.

The Javadoc comment should describe *what* the method or constructor does rather than *how*, to allow for any potential implementation change that is invisible to users of an API. It should include the following:

- A description of any preconditions or postconditions
- Javadoc tag elements that describe any parameters, return value, and thrown exceptions
- Any other important aspects such as side-effects and thread safety

Documentation for users of an API should be written using the standard Javadoc format. This can be accessed conveniently by users of an API from within standard IDEs, and can be transformed automatically into HTML format.

Example

The following example shows a good Javadoc comment, which clearly explains what the method does, its parameter, return value, and thrown exception.

```
1  /**
2   * Extracts the user's name from the input arguments.
3   *
4   * Precondition: 'args' should contain at least one element, the user's name.
5   *
6   * @param  args            the command-line arguments.
7   * @return                 the user's name (the first command-line argument).
8   * @throws NoNameException if 'args' contains no element.
9   */
10  public static String getName(String[] args) throws NoNameException {
11      if(args.length == 0) {
12          throw new NoNameException();
13      } else {
14          return args[0];
15      }
16  }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 44. Addison-Wesley, 2008.
- Help - Eclipse Platform: Java Compiler Javadoc Preferences.
- Java SE Documentation: How to Write Doc Comments for the Javadoc Tool, Requirements for Writing Java API Specifications.

Include a Javadoc tag for each exception thrown by a public method or constructor

Category: Advisory > Documentation

Description: A public method or constructor that throws an exception but does not have a Javadoc tag for the exception affects maintainability.

A public method or constructor that throws an exception but does not have a Javadoc tag for the exception makes an API more difficult to understand and maintain. This includes checked exceptions in `throws` clauses and unchecked exceptions that are explicitly thrown in `throw` statements.

Recommendation

The Javadoc comment for a method or constructor should include a Javadoc tag element that describes each thrown exception.

Example

The following example shows a good Javadoc comment, which clearly explains the method's thrown exception.

```
1  /**
2   * Extracts the user's name from the input arguments.
3   *
4   * Precondition: 'args' should contain at least one element, the user's name.
5   *
6   * @param  args            the command-line arguments.
7   * @return                 the user's name (the first command-line argument).
8   * @throws NoNameException if 'args' contains no element.
9   */
10 public static String getName(String[] args) throws NoNameException {
11     if(args.length == 0) {
12         throw new NoNameException();
13     } else {
14         return args[0];
15     }
16 }
```

References

- J. Bloch, *Effective Java (second edition)*, Items 44 and 62. Addison-Wesley, 2008.
- Help - Eclipse Platform: Java Compiler Javadoc Preferences.
- Java SE Documentation: How to Write Doc Comments for the Javadoc Tool, Requirements for Writing Java API Specifications.

Include a Javadoc tag for each parameter of a public method or constructor

Category: Advisory > Documentation

Description: A public method or constructor that does not have a Javadoc tag for each parameter affects maintainability.

A public method or constructor that does not have a Javadoc tag for each parameter makes an API more difficult to understand and maintain.

Recommendation

The Javadoc comment for a method or constructor should include a Javadoc tag element that describes each parameter.

Example

The following example shows a good Javadoc comment, which clearly explains the method's parameter.

```
1  /**
2   * Extracts the user's name from the input arguments.
3   *
4   * Precondition: 'args' should contain at least one element, the user's name.
5   *
6   * @param  args            the command-line arguments.
7   * @return                 the user's name (the first command-line argument).
8   * @throws NoNameException if 'args' contains no element.
9   */
10  public static String getName(String[] args) throws NoNameException {
11      if(args.length == 0) {
12          throw new NoNameException();
13      } else {
14          return args[0];
15      }
16  }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 44. Addison-Wesley, 2008.
- Help - Eclipse Platform: Java Compiler Javadoc Preferences.
- Java SE Documentation: How to Write Doc Comments for the Javadoc Tool, Requirements for Writing Java API Specifications.

Include a Javadoc tag for the return value of a public method or constructor

Category: Advisory > Documentation

Description: A public method that does not have a Javadoc tag for its return value affects maintainability.

A public method that does not have a Javadoc tag for its return value makes an API more difficult to understand and maintain.

Recommendation

The Javadoc comment for a method should include a Javadoc tag element that describes the return value.

Example

The following example shows a good Javadoc comment, which clearly explains the method's return value.

```
1  /**
2   * Extracts the user's name from the input arguments.
3   *
4   * Precondition: 'args' should contain at least one element, the user's name.
5   *
6   * @param  args            the command-line arguments.
7   * @return                 the user's name (the first command-line argument).
8   * @throws NoNameException if 'args' contains no element.
9   */
10 public static String getName(String[] args) throws NoNameException {
11     if(args.length == 0) {
12         throw new NoNameException();
13     } else {
14         return args[0];
15     }
16 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 44. Addison-Wesley, 2008.
- Help - Eclipse Platform: Java Compiler Javadoc Preferences.
- Java SE Documentation: How to Write Doc Comments for the Javadoc Tool, Requirements for Writing Java API Specifications.

Java Objects (1)

- Avoid overriding 'Object.clone'
- Avoid overriding 'Object.finalize'
- Avoid using a method that overrides 'Object.clone'
- Avoid using the 'Cloneable' interface

Avoid overriding 'Object.clone'

Category: Advisory > Java Objects (1)

Description: Overriding 'Object.clone' is bad practice. Copying an object using the 'Cloneable interface' and 'Object.clone' is error-prone.

Copying an object using the `Cloneable` interface and the `Object.clone` method is error-prone. This is because the `Cloneable` interface and the `clone` method are unusual:

- The `Cloneable` interface has no methods. Its only use is to trigger different behavior of `Object.clone`.
- `Object.clone` is protected.
- `Object.clone` creates a shallow copy without calling a constructor.

The first two points mean that a programmer must do two things to get a useful implementation of `clone`: first, make the class implement `Cloneable` to change the behavior of `Object.clone` so that it makes a copy instead of throwing a `CloneNotSupportedException`; second, override `clone` to make it public, to allow it to be called. Another consequence of `Cloneable` not having any methods is that it does not say anything about an object that implements it, which means that you cannot perform a polymorphic clone operation.

The third point, `Object.clone` creating a shallow copy, is the most serious one. A shallow copy shares internal state with the original object. This includes private fields that the programmer might not be aware of. A change to the internal state of the original object could affect the copy, and conversely the opposite is true, which could easily lead to unexpected behavior.

Recommendation

Define either a dedicated copy method or a copy constructor (with a parameter whose type is the same as the type that declares the constructor). In most cases, this is at least as good as using the `Cloneable` interface and the `Object.clone` method, without the subtlety involved in implementing and using `clone` correctly.

Example

In the following example, class `Galaxy` includes a copy constructor. Its parameter is of type `Galaxy`.

```
1  public final class Galaxy {
2
3      // This is the original constructor.
4      public Galaxy (double aMass, String aName) {
5          fMass = aMass;
6          fName = aName;
7      }
8
9      // This is the copy constructor.
10      public Galaxy(Galaxy aGalaxy) {
11          this(aGalaxy.getMass(), aGalaxy.getName());
12      }
13
14      // ...
15 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 11. Addison-Wesley, 2008.
- Java Platform, Standard Edition 6, API Specification: Interface Cloneable, Object.clone.

Avoid overriding 'Object.finalize'

Category: Advisory > Java Objects (1)

Description: Overriding 'Object.finalize' is not a reliable way to terminate use of resources.

Overriding the `Object.finalize` method is not a reliable way to terminate use of resources. In particular, there are no guarantees regarding the timeliness of finalizer execution.

Recommendation

Provide explicit termination methods, which should be called by users of an API.

References

- J. Bloch, *Effective Java (second edition)*, Item 7. Addison-Wesley, 2008.
- Java Language Specification: 12.6. Finalization of Class Instances.

Avoid using a method that overrides 'Object.clone'

Category: Advisory > Java Objects (1)

Description: Calling a method that overrides 'Object.clone' is bad practice. Copying an object using the 'Cloneable interface' and 'Object.clone' is error-prone.

Copying an object using the `Cloneable` interface and the `Object.clone` method is error-prone. This is because the `Cloneable` interface and the `clone` method are unusual:

- The `Cloneable` interface has no methods. Its only use is to trigger different behavior of `Object.clone`.
- `Object.clone` is protected.
- `Object.clone` creates a shallow copy without calling a constructor.

The first two points mean that a programmer must do two things to get a useful implementation of `clone`: first, make the class implement `Cloneable` to change the behavior of `Object.clone` so that it makes a copy instead of throwing a `CloneNotSupportedException`; second, override `clone` to make it public, to allow it to be called. Another consequence of `Cloneable` not having any methods is that it does not say anything about an object that implements it, which means that you cannot perform a polymorphic clone operation.

The third point, `Object.clone` creating a shallow copy, is the most serious one. A shallow copy shares internal state with the original object. This includes private fields that the programmer might not be aware of. A change to the internal state of the original object could affect the copy, and conversely the opposite is true, which could easily lead to unexpected behavior.

Recommendation

Define either a dedicated copy method or a copy constructor (with a parameter whose type is the same as the type that declares the constructor). In most cases, this is at least as good as using the `Cloneable` interface and the `Object.clone` method, without the subtlety involved in implementing and using `clone` correctly.

Example

In the following example, class `Galaxy` includes a copy constructor. Its parameter is of type `Galaxy`.

```
1  public final class Galaxy {
2
3      // This is the original constructor.
4      public Galaxy (double aMass, String aName) {
5          fMass = aMass;
6          fName = aName;
7      }
8
9      // This is the copy constructor.
10      public Galaxy(Galaxy aGalaxy) {
11          this(aGalaxy.getMass(), aGalaxy.getName());
12      }
13
14      // ...
15 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 11. Addison-Wesley, 2008.
- Java Platform, Standard Edition 6, API Specification: Interface Cloneable, Object.clone.

Avoid using the 'Cloneable' interface

Category: Advisory > Java Objects (1)

Description: Using the 'Cloneable' interface is bad practice. Copying an object using the 'Cloneable interface' and 'Object.clone' is error-prone.

Copying an object using the `Cloneable` interface and the `Object.clone` method is error-prone. This is because the `Cloneable` interface and the `clone` method are unusual:

- The `Cloneable` interface has no methods. Its only use is to trigger different behavior of `Object.clone`.
- `Object.clone` is protected.
- `Object.clone` creates a shallow copy without calling a constructor.

The first two points mean that a programmer must do two things to get a useful implementation of `clone`: first, make the class implement `Cloneable` to change the behavior of `Object.clone` so that it makes a copy instead of throwing a `CloneNotSupportedException`; second, override `clone` to make it public, to allow it to be called. Another consequence of `Cloneable` not having any methods is that it does not say anything about an object that implements it, which means that you cannot perform a polymorphic clone operation.

The third point, `Object.clone` creating a shallow copy, is the most serious one. A shallow copy shares internal state with the original object. This includes private fields that the programmer might not be aware of. A change to the internal state of the original object could affect the copy, and conversely the opposite is true, which could easily lead to unexpected behavior.

Recommendation

Define either a dedicated copy method or a copy constructor (with a parameter whose type is the same as the type that declares the constructor). In most cases, this is at least as good as using the `Cloneable` interface and the `Object.clone` method, without the subtlety involved in implementing and using `clone` correctly.

Example

In the following example, class `Galaxy` includes a copy constructor. Its parameter is of type `Galaxy`.

```
1  public final class Galaxy {
2
3      // This is the original constructor.
4      public Galaxy (double aMass, String aName) {
5          fMass = aMass;
6          fName = aName;
7      }
8
9      // This is the copy constructor.
10      public Galaxy(Galaxy aGalaxy) {
11          this(aGalaxy.getMass(), aGalaxy.getName());
12      }
13
14      // ...
15 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 11. Addison-Wesley, 2008.
- Java Platform, Standard Edition 6, API Specification: Interface Cloneable, Object.clone.

Naming (1)

- Begin a class or interface name with an uppercase letter
- Begin a method name with a lowercase letter
- Begin a variable name with a lowercase letter
- Use lowercase letters throughout a package name
- Use uppercase letters throughout a constant name

Begin a class or interface name with an uppercase letter

Category: Advisory > Naming (1)

Description: A class or interface name that begins with a lowercase letter decreases readability.

A class or interface name that begins with a lowercase letter does not follow standard naming conventions, which decreases code readability. For example, `hotelbooking`.

Recommendation

Begin the class name with an uppercase letter and use camel case: capitalize the first letter of each word within the class name. For example, `HotelBooking`.

References

- J. Bloch, *Effective Java (second edition)*, Item 56. Addison-Wesley, 2008.
- Java Language Specification: 6.1. Declarations.
- Java SE Documentation: 9 - Naming Conventions.

Begin a method name with a lowercase letter

Category: Advisory > Naming (1)

Description: A method name that begins with an uppercase letter decreases readability.

A method name that begins with an uppercase letter does not follow standard naming conventions, which decreases code readability. For example, `Getbackground`.

Recommendation

Begin the method name with a lowercase letter and use camel case: capitalize the first letter of each word within the method name. For example, `getBackground`.

References

- J. Bloch, *Effective Java (second edition)*, Item 56. Addison-Wesley, 2008.
- Java Language Specification: 6.1. Declarations.
- Java SE Documentation: 9 - Naming Conventions.

Begin a variable name with a lowercase letter

Category: Advisory > Naming (1)

Description: A variable name that begins with an uppercase letter decreases readability.

A variable name that begins with an uppercase letter does not follow standard naming conventions, which decreases code readability. For example, `Numberofguests`. This applies to local variables, parameters, and non-constant fields.

Recommendation

Begin the variable name with a lowercase letter and use camel case: capitalize the first letter of each word within the variable name. For example, `numberOfGuests`.

References

- J. Bloch, *Effective Java (second edition)*, Item 56. Addison-Wesley, 2008.
- Java Language Specification: 6.1. Declarations.
- Java SE Documentation: 9 - Naming Conventions.

Use lowercase letters throughout a package name

Category: Advisory > Naming (1)

Description: A package name that contains uppercase letters decreases readability.

A package name that contains uppercase letters does not follow standard naming conventions, which decreases code readability. For example, `Com.Sun.Eng`.

Recommendation

Use lowercase letters throughout a package name. For example, `com.sun.eng`.

References

- J. Bloch, *Effective Java (second edition)*, Item 56. Addison-Wesley, 2008.
- Java Language Specification: 6.1. Declarations.
- Java SE Documentation: 9 - Naming Conventions.

Use uppercase letters throughout a constant name

Category: Advisory > Naming (1)

Description: A static, final field name that contains lowercase letters decreases readability.

A static, final field name that contains lowercase letters does not follow standard naming conventions, which decreases code readability. For example, `Min_Width`.

Recommendation

Use uppercase letters throughout a static, final field name, and use underscores to separate words within the field name. For example, `MIN_WIDTH`.

References

- J. Bloch, *Effective Java (second edition)*, Item 56. Addison-Wesley, 2008.
- Java Language Specification: 6.1. Declarations.
- Java SE Documentation: 9 - Naming Conventions.

Statements

- Avoid writing more than one statement per line
- Include a 'default' case in a 'switch' statement
- Include a terminating 'else' clause in an 'if-else-if' statement

Avoid writing more than one statement per line

Category: Advisory > Statements

Description: More than one statement per line decreases readability.

Code where each statement is defined on a separate line is much easier for programmers to read than code where multiple statements are defined on the same line.

Recommendation

Separate statements by a newline character.

References

- Java SE Documentation: 7.1 Simple Statements.

Include a 'default' case in a 'switch' statement

Category: Advisory > Statements

Description: A 'switch' statement that is based on a non-enumerated type and that does not have a 'default' case may allow execution to 'fall through' silently.

A `switch` statement without a `default` case may allow execution to 'fall through' silently, if no cases are matched.

Recommendation

In a `switch` statement that is based on a variable of a non-enumerated type, include a `default` case to prevent execution from falling through silently when no cases are matched. If the `default` case is intended to be unreachable code, it is advisable that it throws a `RuntimeException` to alert the user of an internal error.

Example

In the following example, the `switch` statement outputs the menu choice that the user has made. However, if the user does not choose 1, 2, or 3, execution falls through silently.

```
1  int menuChoice;
2
3  // ...
4
5  switch (menuChoice) {
6      case 1:
7          System.out.println("You chose number 1.");
8          break;
9      case 2:
10          System.out.println("You chose number 2.");
11          break;
12      case 3:
13          System.out.println("You chose number 3.");
14          break;
15      // BAD: No 'default' case
16  }
```

In the following modified example, the `switch` statement includes a `default` case, to allow for the user making an invalid menu choice.

```
1  int menuChoice;
2
3  // ...
4
5  switch (menuChoice) {
6      case 1:
7          System.out.println("You chose number 1.");
8          break;
9      case 2:
10          System.out.println("You chose number 2.");
11          break;
12      case 3:
13          System.out.println("You chose number 3.");
14          break;
15      default:  // GOOD: 'default' case for invalid choices
16          System.out.println("Sorry, you made an invalid choice.");
17          break;
18  }
```

References

- Java SE Documentation: 7.8 switch Statements.

Include a terminating 'else' clause in an 'if-else-if' statement

Category: Advisory > Statements

Description: An 'if-else-if' statement without a terminating 'else' clause may allow execution to 'fall through' silently.

An `if-else-if` statement without a terminating `else` clause may allow execution to 'fall through' silently, if none of the `if` clauses are matched.

Recommendation

Include a terminating `else` clause to `if-else-if` statements to prevent execution from falling through silently. If the terminating `else` clause is intended to be unreachable code, it is advisable that it throws a `RuntimeException` to alert the user of an internal error.

Example

In the following example, the `if` statement outputs the grade that is achieved depending on the test score. However, if the score is less than 60, execution falls through silently.

```
1  int score;
2  char grade;
3
4  // ...
5
6  if (score >= 90) {
7      grade = 'A';
8  } else if (score >= 80) {
9      grade = 'B';
10 } else if (score >= 70) {
11     grade = 'C';
12 } else if (score >= 60) {
13     grade = 'D';
14   // BAD: No terminating 'else' clause
15 }
16 System.out.println("Grade = " + grade);
```

In the following modified example, the `if` statement includes a terminating `else` clause, to allow for scores that are less than 60.

```
1  int score;
2  char grade;
3
4  // ...
5
6  if (score >= 90) {
7      grade = 'A';
8  } else if (score >= 80) {
9      grade = 'B';
10 } else if (score >= 70) {
11     grade = 'C';
12 } else if (score >= 60) {
13     grade = 'D';
14 } else {  // GOOD: Terminating 'else' clause for all other scores
15     grade = 'F';
16 }
17 System.out.println("Grade = " + grade);
```

References

- Java SE Documentation: 7.4 if, if-else, if else-if else Statements.

Types (1)

- Provide type parameters in call to a constructor of a generic type
- Provide type parameters to generic types
- Use a parameterized instance of a generic type for a method return type

Provide type parameters in call to a constructor of a generic type

Category: Advisory > Types (1)

Description: Parameterizing a call to a constructor of a generic type increases type safety and code readability.

The use of generics in Java improves compile-time type safety and code readability. Users of a class or interface that has been designed using generic types should therefore make use of parameterized instances in variable declarations, method return types, and constructor calls.

Recommendation

Provide type parameters to generic classes and interfaces where possible.

Note that converting legacy code to use generics may have to be done carefully in order to preserve the existing functionality of an API; for detailed guidance, see the references.

Example

The following example is poorly written because it uses raw types. This makes it more error prone because the compiler is less able to perform type checks.

```
1 public List constructRawList(Object o) {
2     List list;  // Raw variable declaration
3     list = new ArrayList();  // Raw constructor call
4     list.add(o);
5     return list;  // Raw method return type (see signature above)
6 }
```

A parameterized version can be easily made and is much safer.

```
1 public <T> List<T> constructParameterizedList(T o) {
2     List<T> list;  // Parameterized variable declaration
3     list = new ArrayList<T>();  // Parameterized constructor call
4     list.add(o);
5     return list;  // Parameterized method return type (see signature above)
6 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 23. Addison-Wesley, 2008.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- The Java Tutorials: Generics, Converting Legacy Code to Use Generics.

Provide type parameters to generic types

Category: Advisory > Types (1)

Description: Declaring a field, parameter, or local variable as a parameterized type increases type safety and code readability.

The use of generics in Java improves compile-time type safety and code readability. Users of a class or interface that has been designed using generic types should therefore make use of parameterized instances in variable declarations, method return types, and constructor calls.

Recommendation

Provide type parameters to generic classes and interfaces where possible.

Note that converting legacy code to use generics may have to be done carefully in order to preserve the existing functionality of an API; for detailed guidance, see the references.

Example

The following example is poorly written because it uses raw types. This makes it more error prone because the compiler is less able to perform type checks.

```
1 public List constructRawList(Object o) {
2     List list;  // Raw variable declaration
3     list = new ArrayList();  // Raw constructor call
4     list.add(o);
5     return list;  // Raw method return type (see signature above)
6 }
```

A parameterized version can be easily made and is much safer.

```
1 public <T> List<T> constructParameterizedList(T o) {
2     List<T> list;  // Parameterized variable declaration
3     list = new ArrayList<T>();  // Parameterized constructor call
4     list.add(o);
5     return list;  // Parameterized method return type (see signature above)
6 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 23. Addison-Wesley, 2008.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- The Java Tutorials: Generics, Converting Legacy Code to Use Generics.

Use a parameterized instance of a generic type for a method return type

Category: Advisory > Types (1)

Description: Using a parameterized instance of a generic type for a method return type increases type safety and code readability.

The use of generics in Java improves compile-time type safety and code readability. Users of a class or interface that has been designed using generic types should therefore make use of parameterized instances in variable declarations, method return types, and constructor calls.

Recommendation

Provide type parameters to generic classes and interfaces where possible.

Note that converting legacy code to use generics may have to be done carefully in order to preserve the existing functionality of an API; for detailed guidance, see the references.

Example

The following example is poorly written because it uses raw types. This makes it more error prone because the compiler is less able to perform type checks.

```
1 public List constructRawList(Object o) {
2     List list;  // Raw variable declaration
3     list = new ArrayList();  // Raw constructor call
4     list.add(o);
5     return list;  // Raw method return type (see signature above)
6 }
```

A parameterized version can be easily made and is much safer.

```
1 public <T> List<T> constructParameterizedList(T o) {
2     List<T> list;  // Parameterized variable declaration
3     list = new ArrayList<T>();  // Parameterized constructor call
4     list.add(o);
5     return list;  // Parameterized method return type (see signature above)
6 }
```

References

- J. Bloch, *Effective Java (second edition)*, Item 23. Addison-Wesley, 2008.
- Help - Eclipse Platform: Java Compiler Errors/Warnings Preferences.
- The Java Tutorials: Generics, Converting Legacy Code to Use Generics.