Java Coding Standards

Applied Intelligence

Application Services Practice

By Jack Dexter, Steven Bilotta, Peter Moore and Richard Molland

03 October 2014

Document Reference : BAES AI-JavaCodeStd-1.2

68 pages including cover

Applied Intelligence

BAE SYSTEMS
INSPIRED WORK

Version history

Version	Date	Author	Action
1.0	24 Sep 2013	JD	Issued.
1.1	25 Jul 2014	RJCM	Revised for issue to Steve Cummins (Cambridge University)
1.2	03 Oct 2014	RJCM	Incorporate feedback from Alastair Beresford
1.2.1	13/10/2015	SAC	Identified optional items on annotated copy for Cambridge University assessment.

Copyright statement

Executive summary

This document contains Java coding standards to be followed by all BAE Systems Applied Intelligence Java projects.

The code standards can be summarised as:

1. Code defensively – you must check all parameters to ensure they conform to the expected (handled values).
2. Handle all errors – do not swallow exceptions.
3. Think about security – defend your code from attack.
4. Think about performance – think how the code will cope with real datasets.
5. Consider the supportability of the code – log what's going on, put in good comments, think about those developers who will work with code after you.
6. Be consistent in code style – it doesn't actually matter where the curly braces go, as long as they are consistent across the project.

References

Mnemonic	Document Details
[1]	Title: Oracle Java Code Conventions Doc Ref: http://www.oracle.com/technetwork/java/codeconv-138413.html
[2]	Title: Geosoft Code Coding Standards Doc Ref: http://geosoft.no/development/javastyle.html
[3]	Title: Effective Java ™, Prentice Hall Doc Ref: http://my.safaribooksonline.com/book/programming/java/9780137150021
[4]	Title: Exception Handling Best Practices in Java Programming Doc Ref: http://javarevisited.blogspot.co.uk/2013/03/0-exception-handling-best-practices-in-Java-Programming.html
[5]	Title: Exception Handling Best Practices Doc Ref: http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html
[6]	Title: Secure Coding Guidelines for the Java Programming Language Doc Ref: http://www.oracle.com/technetwork/java/seccodeguide-139067.html
[7]	Title: CERT Secure Programming Doc Ref: https://www.securecoding.cert.org/confluence/display/java/Introduction
[8]	Title: Java performance tuning, O'Reilly Media Doc Ref: http://shop.oreilly.com/product/9780596000158.do
[9]	Title: Code Conventions for the Java TM Programming Language Doc Ref: http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html#367

List of contents

1. Introduction

This document lists Java coding recommendations common in the Java development community.

The recommendations are based on established standards collected from a number of sources, individual experience, business requirements/needs, and suggestions given in [1] and [2].

There are several reasons for introducing guidelines rather than just referring to the ones above. The main reason is that these guides are far too general in their scope and that more specific rules need to be established.

Developers and code reviewers should be pragmatic in applying coding standards. This guide cannot possibly cover every code eventuality, so ultimately developers should ensure that their code is written in a way that maximises its readability and maintainability.

1.1 Purpose

The purpose of this document is to capture the distilled wisdom that senior developers in Applied Intelligence have amassed over the years. They have made plenty of mistakes in their time and would prefer it if you didn't do the same. They want you to be creative and invent some nice new mistakes of your own and don't just repeat the ones they made.

More formally, the document purpose is to:

- Help programmers reduce probability of runtime errors in code.
- Improve readability and maintainability of code.
- Deliver consistent, professional source code.

1.2 How to read the document

The following code standards are a guideline, a steer in the right direction. They are not an absolute concrete rulebook that must be followed. The main reason to have these coding standards is so that the whole company has a common starting ground, and we fully understand that certain teams/projects will need to advance, amend or drop certain standards for historic, technology or preference reasons. Please feel free to do so, but what we do ask is that you are consistent throughout your project and that you share any standards or better ways of working with the wider community.

1.3 Document Layout

The document is big. You won't read it all. To help ensure you read the bits that you should, the document is broken into 3 chapters and 3 appendixes. The main document provides a summary and a set of Golden Rules to follow. The, perhaps less exciting, detail is in the Appendices.

The document structure is as follows:

- Chapter 1 is this introduction;
- Chapter 2 is a practical summary with the 'Coding Standards on a Page (or three)';
- Chapter 3 gives the Golden Rules of successful, professional programming;
- Appendix A details some common Java specific mistakes to avoid;
- Appendix B details general coding standards;
- Appendix C details code style and naming conventions you could/should use.

In the appendix the standards are grouped by topic and each standard is numbered for ease of reference.

Layout for each standard is as follows:

Ref#. Standard short description
`Code Example`
Motivation, background and additional information.

1.4 Document control

This document is owned and controlled by the BAE Systems Applied Intelligence Application Services Practice.

The Practice welcomes review comments and updates. Please contact the Head of the Practice.

1.5 Recommended further reading

In addition to this code standard, 'Effective Java: Second Edition' by Joshua Bloch is highly recommended reading and provides guidelines for writing efficient, well designed Java programs.

2. Coding Standards on a Page (or three)

Opt - Indicates Optional Rule for Cambridge Courses.

```
/**
* MyJavaCodingStandards.java - © BAESystems Detica 2013
*
* Not Protectively Marked
*/

import java.util.List;
import java.util.ArrayList;

/**
 *Does cool things. Positively chilly in fact. My kids would say
 * the functionality is "epic". Probably with an American accent.
 * Thanks TV.
 */
class MyJavaCodingStandards {
    Logger log = LoggerFactory.getLogger(MyJavaCodingStandards.class);

    public static final int RANGE_MIN = 0;
    public static final int RANGE_MAX = 100;

    private AnotherClass classVariable;

    MyJavaCodingStandards(){
        classVariable = new AnotherClass();
    }

    public void randomNumber(Integer x){
        if( null == x ){
            throw new IllegalArgumentException("x cannot be null");
        }

        MyObject obj = classVariable.getDodgyObject();

        if( null == obj ){
            throw new DodgyObjectException();
        }

        if( x > RANGE_MIN && x < RANGE_MAX ){
            ...
        }
    }


    public void avoidAutoboxing(int x){
        Integer intToAutoBox = x;

        int i = intToAutoBox;

    public static float methodExample(int x){
        int [] a = new int[20];

        float floatValue = (float) x;

        return floatValue;
    }
```

All code files must have a header with copy-right and protective markings Opt

Find More Detail in Appendix C.1.5

Imported classes should listed explicitly, avoid java.util.*

C.7.3

All code files must have a description Opt

C.1.5

CamelCase Class names

Constant—All caps with "_" separators. Avoid magic numbers, declare named constants.

B.5.3

C.2.7

Member variables should be declared private. Use getter and setter methods.

C.11.4

C.5.3

Avoid TAB, Use 4 Spaces Opt

All input parameters shall be checked for null values. Opt
Similarly, if you call a method that returns an object.

B.1.1

Code Defensively!

All input parameters shall be checked for range validity Opt

B.1.3

Be wary of automatic features such as autoboxing. Null is probably equivalent to 0 in most cases. But the designers of java cannot know that for sure so it is set to null. Autoboxing will throw an NullPointerException if 'x' is null as it will call x.intValue();

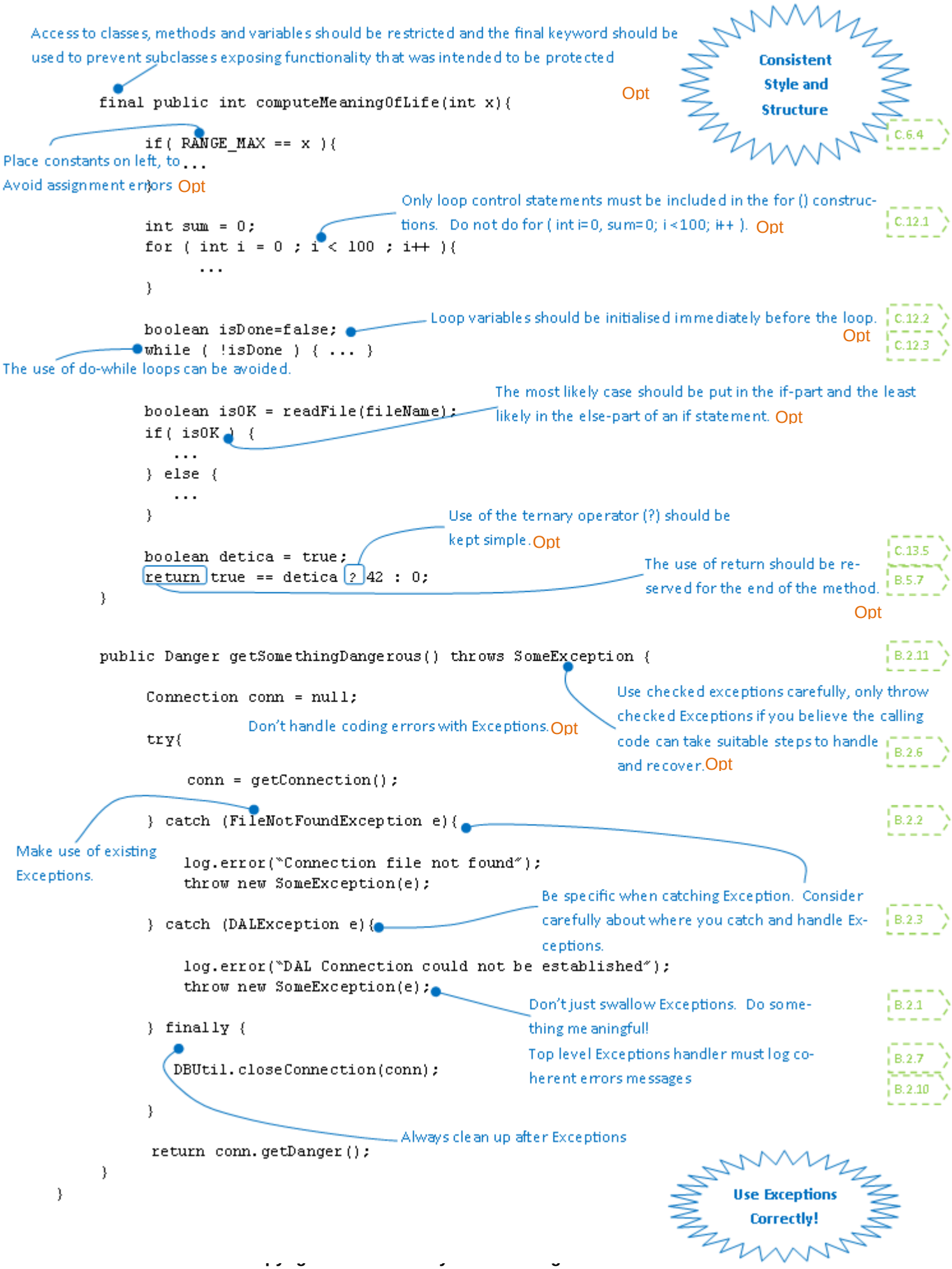
B.1.2

Use this method modifier order, avoid 'static public float'

Array specifiers must be attached to the type not the variable. Opt

C.10.2

Type conversions must always be done explicitly. Never rely on implicit type conversion.



```
import java.util.List;
import java.util.ArrayList;

public class DatabaseStuff extends Database implements Cloneable, Serializable {

    public executeQuery(String dodgyUserInput){
        ...
        Statement st = null;
        rs = st.exectueQuery("Select * from Table where x =" + dodgyUserInput);
        String result = fetchOneRowOnly(rs);
        ...
    }

    /**
     * Return lateral location of the specified position.
     * If the position is unset, NAN is returned
     *
     * @param x    X coordinate
     * @param y    Y coordinate
     * @param zone Zone of position
     * @return Lateral location
     * @throws IllegalArguementException If zone is <= 0
     */
    private double computeLocation(double x, double y, int zone)
                                    throws IllegalArguementException{
        switch (condition){
            case 1 :
                statements;
                /* Falls through */
            case 2 :
                statements;
                break;
            default :
                statements;
                break;
        }

        Matrix4x4 matrix;
        double    cosAngle;
        double    sinAngle;

        // Create a new identity matrix
        matrix = new Matrix4x4();

        // Precompute angles for efficiency
        cosAngle = Math.cos(angle);
        sinAngle = Math.sin(angle);

        // Specify matrix as a rotation transformation
        matrix.setElement(1, 1, cosAngle);
        return 0.0;
    }
}
```

The class and interface declarations should have the following form.

C.2.15

Use prepared statements. Allowing the execution of any String passed into the method leave the Database completely vulnerable to exploit.

Opt

Think about Security

B.4

Think about Performance

Think about Performance, think how the code will perform on the real datasets. For Example, don't select all then filter, use correct SQL.

Opt

C.4.3

JavaDoc comments should have the following form.

Opt

B.2.9

C.5.7

The switch statement should have the following form.

A comment should be added when no break is include.

Every switch statement should include a default case.

Opt

Variables in declarations can be left aligned.
Variable names must be mixed case starting with lowercase.

C.6.4

C.6.2

Logical units within a block should be separated by one blank line and commented correctly.

Opt

C.4.5

C.4.6

Use // for all non-JavaDoc comments, including Multi-line comments. The comments should be indented relative to their position in the code.

Opt

Remove all commented out code

C.1.1

C.1.4

Files should not exceed 1000 lines in length. Do not include more than one class per file. File content must be kept within 120 columns.

3. The Golden Rules

These are the golden rules for successful, happy coding. Follow them!

Or, at least, these are the rules that give you a good chance to write robust, secure, functionally and non-functionally correct code, which is nearly as good.

Some of these rules result in code that is “a bit boring” as it is not compact, complicated and clever. Not boring, “good.” Good code is well documented, well laid out and easy to understand for all. Good code is easy to review and easy to test. Writing good code reduces defects, reduces security vulnerabilities and makes support and re-use easier. Good code is good! Compact and complicated code is for special occasions only.

3.1 Code defensively

Our goal is to make robust, secure, functionally and non-functionally correct code. The key to the first three objectives is ensuring that the code responds correctly to its inputs. One of the keys to responding correctly is checking all parameters that are passed into the method are valid and correct.

This is known as coding defensively; checking all inputs to ensure they are valid and in the appropriate range. Inputs may be passed in via a method call, or returned from a function called within the method, or even a variable declared outside of the scope of the method. In each case, you should follow the cliché of “trust no-one.” Like driving, the rules may say the parameter or variable should not, for example, be null, but unless the compiler can guarantee this is true, you should not believe it.

It is worth noting the fourth objective, to be non-functionally correct, is partly helped by coding defensively and partly hindered. Defensive coding is critical to secure code. However, excessive checking of parameters may have a performance impact. This is a balance that needs to be considered. However, as Applied Intelligence is a company that specialises in secure code, you should err on the side of security and robustness.

All input parameters shall be checked for null values

All input parameters shall be checked for range validity

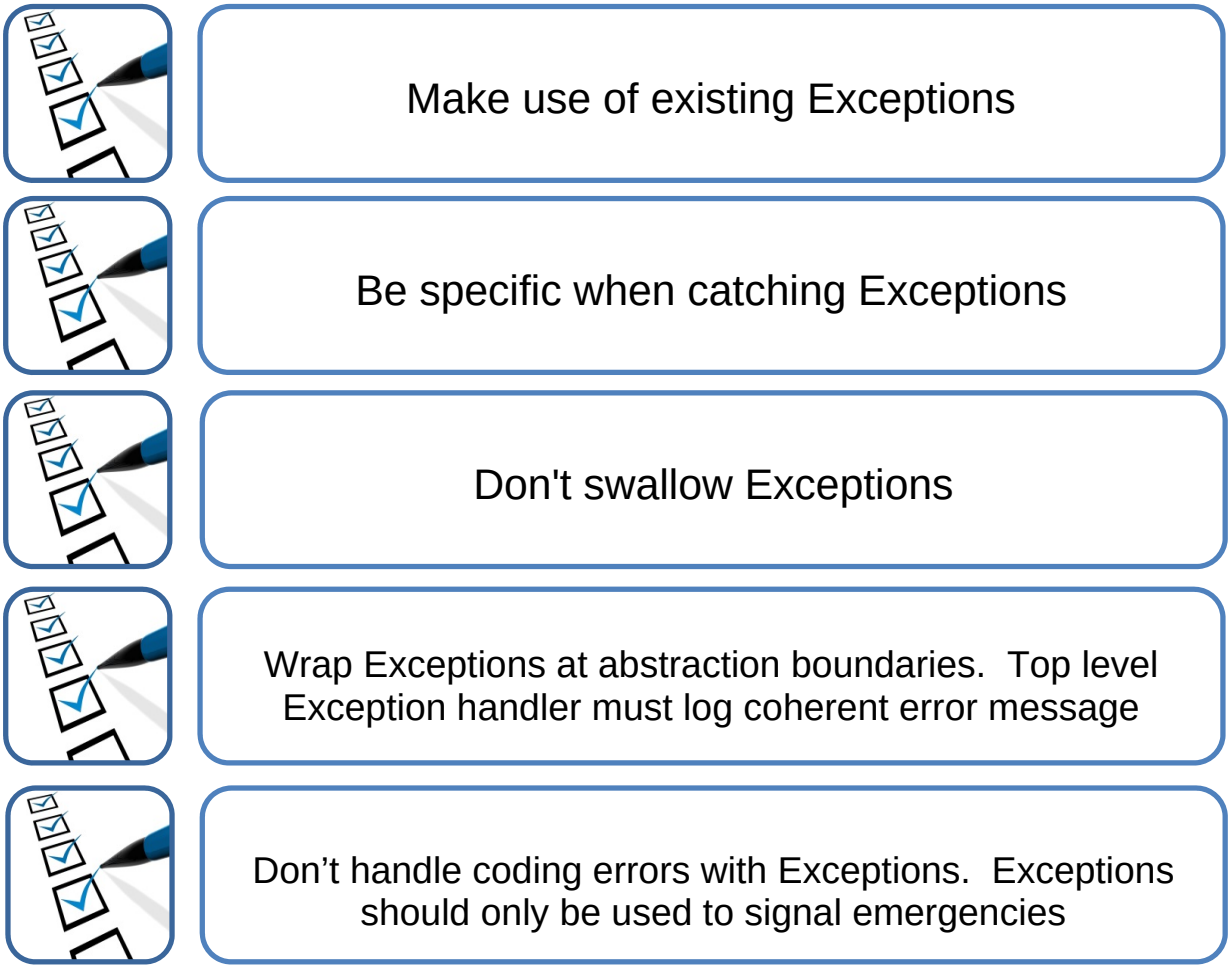
Appendix A. For a full break down of the Defensive Coding standards refer to Appendix F Code Defensively.

Recommended reading: Effective Java ™, Prentice Hall
http://my.safaribooksonline.com/book/programming/java/9780137150021

3.2 Use Exceptions Correctly

The correct use of Exceptions is key to ensuring the software behaves as intended and is easy to maintain and debug. Java provides many existing Exception classes for a large proportion of the error conditions that are likely to occur; these should be reused where possible. Deal with exceptions at the correct level, avoid just catching Exception or RuntimeException and consider carefully about where to catch and handle your Exceptions.

Exceptions are there to signal emergencies and not to control logic or coding errors, such as ArrayOutofBoundsException. Ensure that the exceptions are documented correctly in the JavaDoc. If an exception is caught don't just swallow it, deal and resolve the exception



For a full break down of the Exception standards refer to A.2

Appendix B. ExceptionsCode Defensively. Additional reading can be found at:

http://javarevisited.blogspot.co.uk/2013/03/0-exception-handling-best-practices-in-Java-Programming.html

http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html?page=2

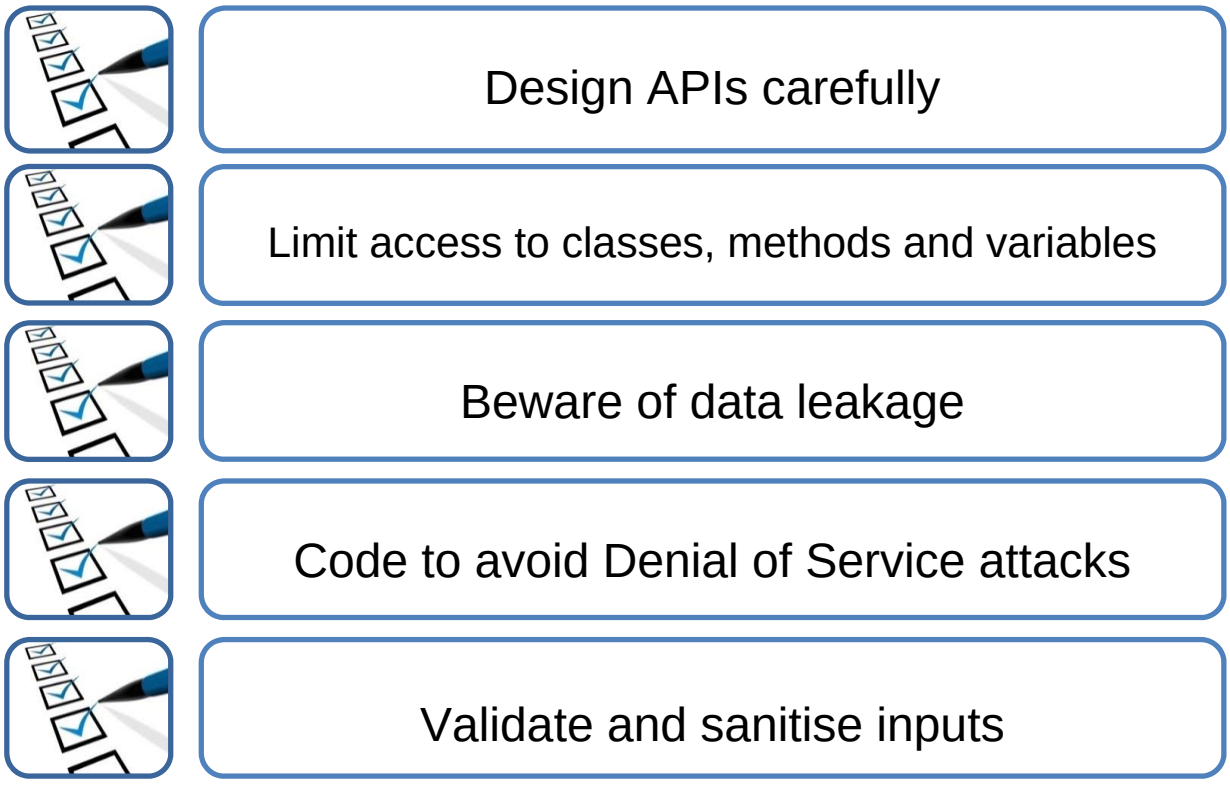
3.3 Focus on security

We should be writing code that is secure and does not contain vulnerabilities that can be exploited by malicious users. The Java virtual machine is designed with security in mind however holes can be opened up via implementation errors. We work with clients that demand the highest levels of security and it is important that we write code that can be trusted.

APIs should be designed with security in mind. Access to classes, methods and variables should be restricted and the final keyword should be used to prevent subclasses exposing functionality that was intended to be protected. It is all too easy to leak sensitive data though poorly designed APIs, excessive logging or Exceptions.

Resource intensive code exposes applications to the possibility of denial of service attacks. Beware of activities that use large amounts of resources, particularly those working with the file system.

Inputs should always be validated to ensure they fall within the expected range handled by the method. Maliciously crafted inputs have the potential to cause a lot of damage e.g. SQL injection. When you cannot trust the caller of your code you must code with security in mind.



The following links provide detailed information on how to code securely and are recommended reading:

http://www.oracle.com/technetwork/java/seccodeguide-139067.html

https://www.securecoding.cert.org/confluence/display/java/Introduction

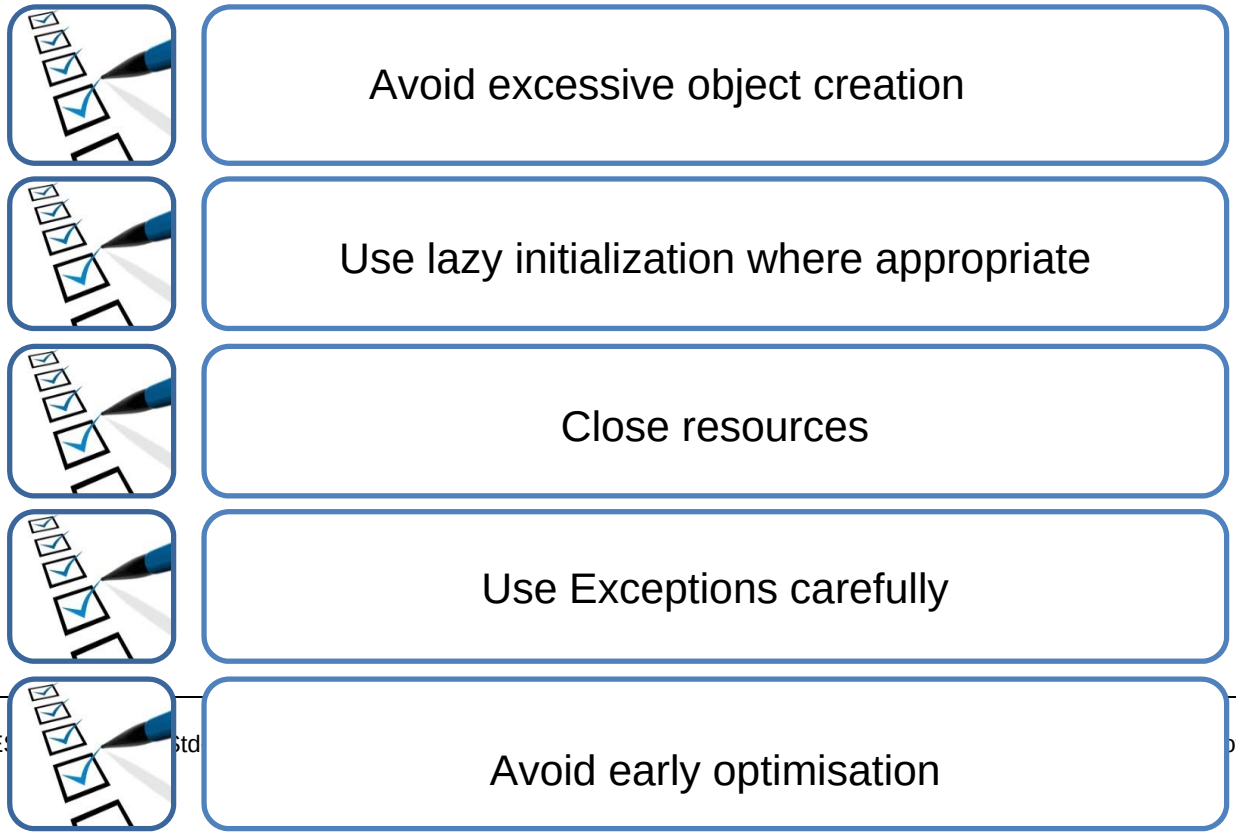
https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide

3.4 Consider performance in all the code you write

Users are quick to identify areas of an application that are slow. A delivery can fail even if it is functionally correct because it suffers with performance problems. Modern processors are quick and the majority of the code we write will not suffer with poor performance. However, it is good to have an understanding of the key causes of issues, be they related to processing requirements, memory usage or IO. Following some simple guidelines, which avoid the most common mistakes, can ensure we write efficient code first time. Many performance problems are associated with design errors where the true volumes of data / network latencies have not been considered up front.

Only optimise the areas of the code that will benefit from it. Optimised code often suffers from decreased readability which leads to increased chances of defects. If the 10% performance improvement does not make a big difference to the end user then choose to keep the code clean simple and correct.

There are numerous ways to increase the performance of a Java application, many of which are specific to particular frameworks e.g. Hibernate, Spring, JSF. These coding standards do not cover these.



Recommended reading:

Java performance tuning, O'Reilly Media http://shop.oreilly.com/product/9780596000158.do

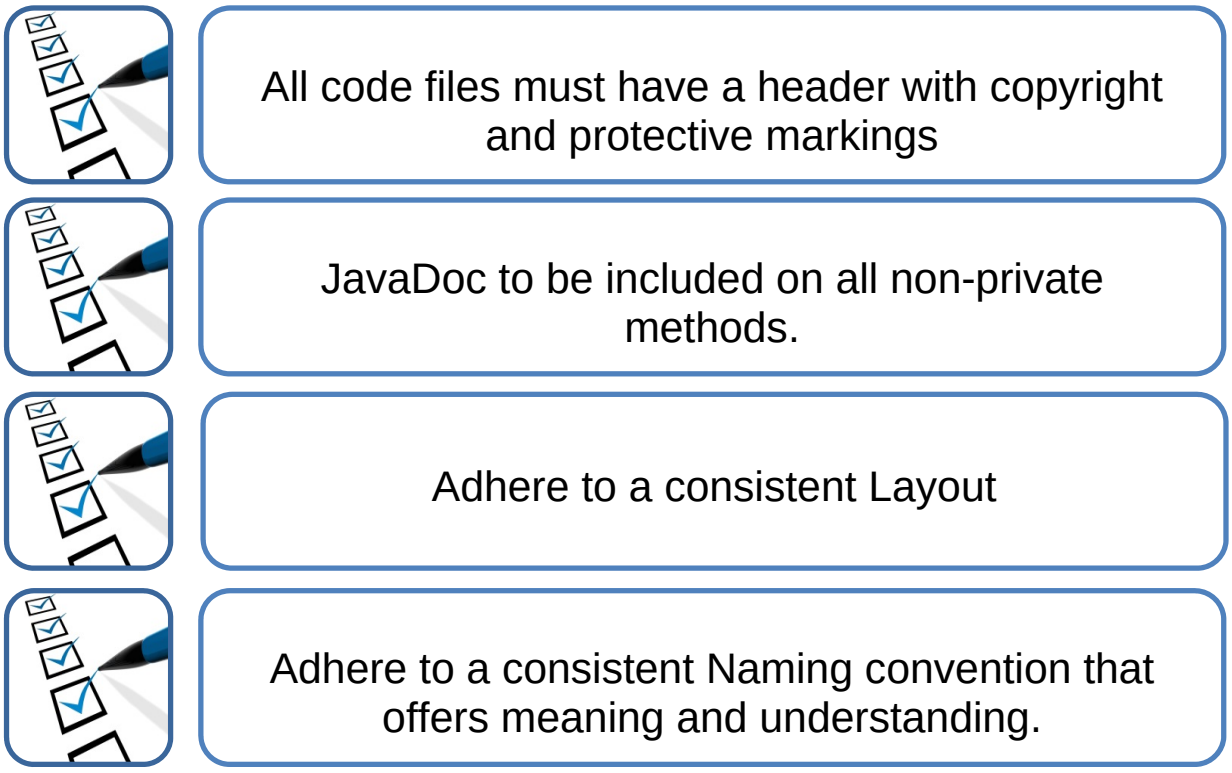
3.5 Consistency in Coding Style and Naming Conventions

The aim here is to be consistent! The code should be JavaDoc'd and commented correctly and laid out in a clear way so that is can be easily read and maintained by others.

Naming conventions define how classes, methods and variables are named.

Naming conventions can be contentious and arbitrary. It actually not important that you adhere to a particular set of conventions, but it is important that they are consistent across the project code base. This document outlines sensible general principles you should follow, unless you have good reason to do otherwise.

There are tools that can assist with the layout of the code. Theses should be used by all members of the development team, automatically enforcing consistency.



Appendix C. For a full break down of the Coding Style and Naming Conventions standards refer to **Error! Reference source not found. Error! Reference source not found.**, C.2 Naming Conventions, C.4 Comments, C.5 LayoutCode Defensively. Additional reading can be found at:

http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html#367

http://geosoft.no/development/javastyle.html

3.6 Make use of Tool Support

Automatic tools are helpful and, once configured, high value for low effort (admittedly, getting the configured can be frustrating and painful, so may not be worth it). You should use static analysis tools to find bugs in your code, check style tools to format your code, dynamic analysis tools to profile your code (looking for performance and memory issues), build tools to ensure clean consistent compilation and test, test tools to run tests, mock dependencies and check test coverage, and supporting frameworks to cover common patterns like Inversion of Control (hello Spring!) and database usage.

Suggested useful tools (be aware that recommended tool choices change regularly, so you should do some research on what tools are considered “best at the moment”):

Tools type	**Recommended examples**
Static Analysis	FindBugs
Format Checking	Checkstyle
Dynamic Analysis	YourKit
Build	Ant, Maven
Continuous Integration	Jenkins
Test Frameworks	TestNG
Mock Frameworks	Mockito
Test Coverage	Covertura, Emma
IDE	Eclipse
Supporting Frameworks	Spring, Hibernate

Appendix D. Common Java mistakes

This Appendix details some common mistakes made by Java programmers that everyone should be aware of.

D.1 Forgetting to override hashCode() and equals()

If you are going to be comparing objects or are working with Collections it is important to override hashCode() and equals().

The hashCode and equals method should take into account the same properties of the class. If two objects are equal by the equals method then their hash codes should also be the same.

It is also important to add the @Override annotation. This will force the compiler to check that you are actually overriding a method you think you are, and it makes the code easier to read because it is obvious when methods are overwritten.

D.2 Treating Strings as mutable

Strings do not change their value. Calling String.trim() does not modify the existing String but returns a new String e.g.

```
String a = “Test   ”
a.trim()
```

In this case a will still have trailing white space. You must reassign the result of the method call to the variable

```
String a = “Test   ”
a = a.trim()
```

D.3 Not checking Objects for NULL

Although new objects are guaranteed to be NOT NULL, you should not assume that all objects are valid. Particularly objects that are passed into your method via parameters, or via a call to another function. Unless you are absolutely sure the object cannot be NULL, make sure your code is NULL safe.

Risky code (don't do it!)

```
void myFunc(Object o1) {
  o1.doStuff(); //Potential NPE!
  o1.getThing().doOtherStuff(); //More potential NPE!
  String suspicious = DodgyClass.getString();
  if(suspicious.equals("trouble!"){ // Potential NPE!
  ..
  }
}
```

Much better

```
void myFunc(Object o1) {
  if(o2 != null) {
    o1.doStuff(); //Potential NPE!
    Object otherStuff = o1.getThing().doOtherStuff();
    if(otherStuff != null) {
    ..
    }
  }
  String suspicious = DodgyClass.getString();
  if("trouble!".equals(suspicious)){
    ..
  }
}
```

Note the NULL safe comparison form. This can be used wherever you have a constant object value (such as a string). E.g.

```
while(MY_CONST.equals(variable1)) {
```

```
..
}
```

D.4 Using == rather than .equals()

When comparing objects you should always use .equals(). The == operator simply compares that two references are equal where in fact the majority of time the developer would like to know if the two objects are actually equal in value.

Also Strings present a special class of pain where == can sometimes work and sometimes not depending on whether the string is in the literal pool.

D.5 Forgetting java is 0 index

Java, unlike some languages, is 0 indexed. This is important when working with arrays

```
Car[] cars = new Car[5];
.
.
cars[5].drive()
```

This would result in a runtime error. The 5th element in the array should be referred to as cars[4] with cars[0] being the first (not cars[1] as you may expect)

D.6 Importing the wrong class

In Java it is possible, and quite common to have classes with the same name but live in different packages. It is a common mistake, particularly when using an IDE to import the incorrect class. In some cases they may even share common method names and you will not be aware of the problem until strange behaviour at runtime. Always double check that your IDE has imported the expected version of the class.

D.7 Missing break statement in switch

Without break statements a switch statement will fall through to the case below. In the below example if foo is 0 then doSomething(), doSomethingElse() and doSomeOtherThing() will all be called. The author probably only expects doSomething() to be executed.

```
switch (foo) {
    case 0:
        doSomething();
    case 1:
        doSomethingElse();
    default:
        doSomeOtherThing();
}
```

This is prevented by using break statements.

```
switch (foo) {
```

```
    case 0:
        doSomething();
        break;
    case 1:
        doSomethingElse();
        break;
    default:
        doSomeOtherThing();
        break;
}
```

D.8 Mistyped overridden method name

It's easy to mistype a method name when overriding from a super class (particularly the capitalization) without noticing.

E.g.

```
public boolean Equal(…)
```

rather than

```
public boolean equals(…)
```

These types of errors can often take a long time to track down. Java is case sensitive so this problem can creep in though the code.

It is important to use the @Override annotation above methods that are being overridden. This will force a compiler error avoiding the mistyped method name issue.

Appendix E. Coding Standards

B.1 Code Defensively

B.2.1 All input parameters shall be checked for null values.

Where a parameter should never be null a check should be included at the beginning of the method.

```
public void randomNumber(Integer x) {
    if(null == x) {
        throw new
            IllegalArgumentException("x cannot be null");
    }
}
```

Similarly, if you call a method that returns an object.

```
public void doSomething() {
    MyObject obj = classVariable.getDodgyObject();
    if (null == obj) {
         throw new DodgyObjectException();
    }
}
```

But what about that class variable? It's ok if it's guaranteed to have been initialised. In the case below, the object itself cannot be initialised if the class variable fails as Java guarantees to either create the object, or throw an exception.

```
class CoolClass {
    MyObjectCreator classVariable = null;
    public CoolClass() {
        classVariable = new MyObjectCreator();
    }
    public void doSomething() {
        MyObject obj = classVariable.getMyObject();
        if (null == obj) {
             throw new DodgyObjectException();
        }
    }
```

```
}
```

But note that if someone changes the class, they could move the initialisation (to fix a bug where the class never initialises) and expose your method! Be paranoid.

B.2.2 Autoboxing is bad

```
public void randomNumber(Integer x) {
    Integer intToAutoBox = null;
    // Uh-oh
    int i = intToAutoBox;
}
```

In Java 5 and later this is translated to

```
public void randomNumber(Integer x) {
    Integer intToAutoBox = null;
    // Uh-oh
    int i = intToAutoBox.intValue();
}
```

Be wary of automatic features such as autoboxing. Null is probably equivalent to 0 in most cases, but the designers of Java cannot know that for sure. So faced with a nasty null pointer exception, or an insidious hidden error (making an assumption that null is the same as 0, when in fact, it's not), they went with the NPE. They are correct, but autoboxing should be avoided.

B.2.3 All input parameters shall be checked for range validity

Values of input parameters should be checked to ensure that the supplied value is within the range that the method is expecting. As with nulls, the same is true of a returned value or a class level variable.

```
static int RANGE_MIN = 0;
static int RANGE_MAX = 100;
public void randomNumber(int x) {
    // Range check
```

```
    if(x < 0) {
        throw new
            IllegalArgumentException("x cannot be less than 0");
    }

    int value = someFunction.getValue();
    if (value > RANGE_MIN && value < RANGE_MAX) {
        ...
    }
}
```

Checking the range validity of a parameter is vital.

Don’t forget overflows and underflows, etc. At a functional level, this could cause all sorts of problems.

```
public void randomNumber(int x) {
    // Range check
    if(x < 0) {
        throw new
            IllegalArgumentException("x cannot be less than 0");
    }

    // What happens when x = Integer.MAX?
    // Or Integer.MAX + 2?
    float value = 100 / x + 1;
}
```

Overflow happens, resulting in the denominator going to 0 and a divide by zero exception.

B.2 Exceptions

B.2.1 Don't swallow Exceptions

Don't catch Exceptions and then do nothing

```
public void doSomething() {
    try {
```

```
        ..code that throws Exceptions
    } catch (Exception ex) {
        ex.printStacktrace();
    }
}
```

Even worse is an empty catch block

```
public void doSomething(){
    try {
        ..code that throws Exceptions
    } catch (Exception ex) {

    }
}
```

The code will simply continue and nobody will be aware that the Exception occurred unless they are looking at the logs. The empty block is a nightmare bug waiting to happen.

If you really must ignore an exception, it can happen that an API insists on throwing an exception when doing nothing would have better, ensure you at least put a comment to show the swallowing is intentional. If possible, log the exception (although this may be hard if the exception happens a lot).

```
public void doSomething(){
    try {
        ..code that throws Exceptions
    } catch (StupidException ex) {
        // This does not matter! We are going to ignore it.
        Logger.getInstance().Log(DEBUG,
            “Ignoring StupidException in doSomething()”);
    }
}
```

B.2.2 Make use of existing Exceptions

The Java API contains many existing Exception classes suitable for a large proportion of the error conditions that are likely to occur. These should be reused where possible. E.g.

FileNotFoundException. Java developers are already familiar with these and comfortable handling them.
Enhances readability. Be specific when catching Exceptions

B.2.3 Be specific when catching Exception

It rarely makes sense to catch just Exception. You should always catch the specific Exceptions thrown by the method you are invoking. This will encourage you to think about how each possible error should be handled. Also do not catch top-level exceptions, Unchecked exceptions inherit from the RuntimeException class, which in turn inherits from Exception. By catching the Exception class, you are also catching RuntimeException.

```
File file = new File(filename);

try{
    readFile(file);
} catch (FileNotFoundException e) {
    // specific code to handle this Exception
} catch (EOFException e) {
    // specific code to handle this Exception
} catch (ObjectStreamException e) {
    // specific code to handle this Exception
} catch (IOException e) {
    // specific code to handle this Exception
}

// Or use the Java 7 and later Exception Handler

try {
    readFile(file);
} catch (FileNotFoundException | EOFExection |
         ObjectStreamException | IOException ex) {
     // specific code to handle this Exception
}
```

B.2.4 Catch Exceptions late

Consider

```
public void readFile(String filename) throws
FileNotFoundException {

    InputStream in = null;
    in = new FileInputStream(filename);
    in.read(...);
}
```

Instead of

```
public void readFile(String filename){
    //...

    InputStream in = null;

    // DO NOT DO THIS!!!
    try {
        in = new FileInputStream(filename);
        in.read(...);
    } catch (FileNotFoundException e) {
        logger.log(e);
    }

    //...
}
```

You should consider carefully about where to catch and handle your Exceptions. If you do this too early it will not be possible to provide meaningful feedback to the user. Exceptions should be propagated up the stack to the layer that is able to act upon the Exception. In the

following case it would probably have been better to propagate the FileNotFoundException to the caller to let it handle it appropriately.

B.2.5 Wrap Exceptions at abstraction boundaries

When propagating an error ensure you wrap the Exception as it crosses an abstraction boundary. E.g. a dao should not be throwing a HibernateException back up to the service layer. It should be wrapped as a DataAccessException hiding the underlying implementation details. The original Exception must be wrapped in the new Exception to preserve the underlying cause.

B.2.6 Don’t handle coding errors with Exceptions

If you find yourself catching an ArrayOutOfBoundsException or DivisionByZero then it indicates that you have some coding errors. These types of Exceptions are usually better handled by checking arguments rather than catching exceptions. For example, an array index should be checked to determine it is less than the array size and greater than or equal to zero.

B.2.7 Provide context with Exception

When throwing an Exception consider if there is any context information that will be useful with debugging. E.g. An error message, parameter values, or even detailed information allowing the calling code to correctly recover. In this case create a new Class, extending Exception with fields and methods to provide access to the specific context details.

B.2.8 Exceptions should only be used to signal emergencies

Exceptions should not be used to control the flow of the code. The following shows an example of some **bad code**.

```
public void increaseToFiveThousand() {
    try {
        while (true) {
```

```
            increaseCount();
        }
    } catch (FiveThousandReachedException ex) {
        Logger.getInstance().Log(DEBUG, “Reached 5000”);
    }
    //Continue execution
}

public void increaseCount()
    throws FiveThousandReachedException {
    if (count >= 5000)
        throw new FiveThousandReachedException();
}
```

Exception handling is expensive, you want to avoid if possible.

Exceptions, conceptually, are for exceptional circumstances that you want to alert the program to. They are not for routine program flow control. Don’t abuse the concept of an exception, keep it simple (stupid). Not least because you can become unstuck if you rely on an exception that can be generated in more than one way.

Don’t be afraid of exceptions though. Use them in circumstances that would be deemed exceptional for that method, such as an illegal input value.

```
/**
 * @param seed The random number seed. This must be greater
 *             than 0.
 */
public void randomNumber(int seed) {
    if (x < 0) {
        throw new
            IllegalArgumentException("x must be > 0");
    }
}
```

B.2.9 Ensure checked Exceptions are documented

Checked Exceptions should be documented using @throws

B.2.10 Always clean up after Exceptions

Always use a finally block to close down any resources initialised prior to the Exception being thrown. Note: If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catch is interrupted of killed, the finally block may not execute even though the application as a whole continues.

```
public void findEmployee() throws SQLException {
    Connection conn = null;
    try {
        conn = getConnection();
        //Code that throws Exception
    } finally {
        DBUtil.closeConnection(conn);
    }
}
```

B.2.11 Use checked Exceptions carefully

Only throw checked Exceptions if you believe the calling code will be able to take suitable steps to handle and recover from the Exception. If not use an unchecked exception. Fewer checked Exceptions simplify the code base with fewer try / catch blocks.

B.2.12 Top level Exception handler must log coherent error message

All Exceptions must be logged by a top level Exception handler if they cannot be successfully handled lower down the stack.

B.3 Logging

This section contains standards for how we should be logging debug and warning messages.

B.3.1 Do not use System.out.println

You should not use System.out.println for logging. There is no guarantee where this will be logged to and the logging could be missed.

B.3.2 Use appropriate logging levels

- Use TRACE level for detailed/diagnostic logging
- Use DEBUG level for things an application developer would need to know
- Use INFO level for things an administrator would need to know
- Use WARN level for things indicating an application or transient problem
- Use ERROR level for things indicating a problem with the server itself

B.3.3 Use slf4j rather than Log4j directly

Allows logging implementations to be easily switched

B.3.4 Do not log throwables

This should be left to the caller. This will prevent multiple instances of the same stack trace appearing in the logs

B.3.5 Check log level is enabled before calling

Ensure that performance is not affected by calls to logger. E.g. in the following case (using log4j) if we did not have the check then we would be evaluating object.toString() even when debug logging is not enabled.

```
if(logger.isDebugEnabled()) {
    logger.debug("returned the following " +
        object.toString())
}
```

B.4 Performance

B.5.8 Avoid excessive object creation

Creating and garbage collecting java objects is expensive. By using appropriate classes, particularly for String manipulation this can be avoided and the performance improvements are significant.

E.g. use StringBuffer to construct a String rather than +

```
String s = “”;
for(int i=0; i<100; i++) {
    s = s + “1”;
}
```

```
StringBuffer sb = new StringBuffer();

for(int i=0; i<100; i++) {
    sb.append(“1”);
}
```

You should use primitive types instead of Objects where appropriate to reduce the number of objects created and destroyed. Particular care to object creation should be paid when working with loops. Where possible declare objects outside of the loop and reuse them.

Techniques such as Object pooling and caching can be used to further reduce the number of object created, improving memory consumption and increasing speed.

Note: Since Java version 1.5 StringBuilder has been available which is faster, however StringBuffer is designed to be thread-safe and all public methods in StringBuffer are synchronized. StringBuilder does not handle thread-safety issues and none of its methods are synchronized.

B.5.9 Don't optimise to soon

Code should only be optimised when there are performance problems. If a piece of code is likely to be executed many times then it is worth spending time on considering how to ensure it runs efficiently or alternatively reduce the number of times it needs to be called, however where performance is unlikely to be an issue it is more important to write clean, easy to read code that is functionally correct.

B.5.10 Use Exceptions carefully

Exception handling in Java is an expensive operation and involves executing several hundred lines of code. Particular attention should be paid to try and catch blocks within loops and if possible these should be moved to outside of the loop.

B.5.11 Close resources manually

You should ensure that you manually close resources once you have finished with them. Do not rely on finalizer methods to close things for you as they can be a considerable delay before the finalizer is called, if indeed it is called at all during the lifetime of the process.

B.5.12 Choose the right sort of Collection

Be aware of the differences between the different types of Collections provided. Make sure you are using the most appropriate Collection for the operations you are performing. Some are good at random access, others are good for insertion and deletion.

Ensure you provide an appropriate initial size for ArrayLists and Vectors in order to gain significant performance improvements.

Only use a Thread safe collection if you really need to. At the expense of code simplicity you can use arrays for even faster performance.

B.5.13 Avoid synchronisation where possible

Calling a synchronised method is considerably slower than calling a non synchronised version.

Where possible you should use non synchronised classes as opposed to any thread safe alternatives e.g. ArrayList vs Vector

B.5.14 Use lazy evaluation

Large Java applications often contain a large amount of static data / initialised objects. These are often initialised using static blocks at the beginning of the class. These work well however can cause the application to take a long time to initialise, this may not always be appropriate, so a design choice is needs to be based on whether load time or fail-fast is more important. The start up time can be decreased by lazy initializing in getter methods.

B.5 Miscellaneous

B.5.1 No unused code

There should be no unused code in the production code base.

B.5.2 80% of code covered by unit tests

It is best practice to achieve 80% of the code covered by a unit test.

B.5.3 The use of magic numbers in the code should be avoided. Numbers other than 0 and 1 can be considered declared as named constants instead.

```
private static final int TEAM_SIZE = 11;
```

```
Player[] players = new Player[TEAM_SIZE];


As opposed to:
Player[] players = new Player[11];
```

If the number does not have an obvious meaning by itself, the readability is enhanced by introducing a named constant instead.

If the value could change then it should be externalised and read from a properties file.

B.5.4 Floating point numbers should always be written with decimal point and at least one decimal.

```
double total = 0.0; // NOT: double total = 0;
double speed = 3.0e8; // NOT: double speed = 3e8;
```

Enhances readability. This highlights the different nature of integer and floating point numbers.

B.5.5 Floating point constants should always be written with a digit before the decimal point.

```
double total = 0.5; // NOT: double total = .5;
```

Enhances readability.

B.5.6 Static variables or methods must always be referred to through the class name and never through an instance variable.

```
Thread.sleep(HEART_BEAT_SECONDS);
// NOT: thread.sleep(HEART_BEAT_SECONDS);
// Also remember to avoid Magic Numbers such as 1000 use a
descriptive constant.
```

This emphasize that the element references is static and independent of any particular instance. For the same reason the class name should also be included when a variable or method is accessed from within the same class.

B.5.7 Never use return in the middle of a method

The use of return should be reserved for the end of a method. Including a return half way through makes refactoring the method into smaller chunks harder.

B.5.8 Do not compound increment / decrement operators

Compound statements can make code complicated to read:

```
foo( x++ );
```

It is clearer if you separate out the statements, like so:

```
foo( x ) ; x++ ;
```

Appendix G. Code Style Standards

C.1 Code Files

This section contains standards related to the files in which we store Java code

C.1.1 File content must be kept within 120 columns.

120 columns is the common dimension for editors, terminal emulators, printers and debuggers, and files that are shared between several developers should keep within these constraints. It improves readability when unintentional line breaks are avoided when passing a file between programmers.

C.1.2 The incompleteness of split lines must be made obvious [1].

Split lines occurs when a statement exceed the 120 column limit given above. It is difficult to give rigid rules for how lines should be split, but in general:

- Break after a comma.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

C.1.3 Only one class per file.

Do not include more than one class per file (unless using inner classes).

C.1.4 Files should not exceed 2000 lines in length

Long files become difficult to manage. Any file over 2000 lines should be refactored into smaller files.

C.1.5 All code files must have a header with copyright and protective markings

```
/**
 * CoolCode.java – (C) BAESystems Detica 2013
 * NPM
 *
 */

import java.util.List;
import java.util.List;

/**
 * Does cool things.  Positively chilly in fact.  My kids would say the functionality is
 * “epic”.  Probably with an American accent.  Thanks TV.
 */
class  MyJavaCodingStandards  {
```

A header at the top of the file should contain the copyright and protective markings details. The class description should be included above the class declaration.

C.2 Naming Conventions

Naming conventions define how classes, methods and variables are named.

Naming conventions can be contentions and arbitrary. It actually not important that you adhere to a particular set of conventions, but it is important that they **must be consistent** across the project. The following are sensible general principles you should follow, unless you have good reason to do otherwise.

C.2.1 Names representing packages should be in all lower case.

```
mypackage
com.detica.application.dao
```

Package naming convention used by Sun for the Java core packages.

C.2.2 Package names should start with com.detica.<modulename>

```
com.detica.modulea
```

This will reduce the chance of clashes of class names.

C.2.3 Names representing types must be nouns and written in mixed case starting with upper case.

```
Car
TransitVan
```

Type naming convention used by Sun for the Java core packages.

C.2.4 Don’t reuse class names from Java API

```
com.detica.File
com.detica.ArrayList
```

Can cause confusion.

C.2.5 Variable names must be mixed case starting with lowercase.

```
accountStatus
canProceed
```

Variable naming convention used by Sun for the Java core packages. Ensures variables can be distinguished from types.

C.2.6 Variable names should differ more than just case

```
private String Foo
private String foo //These are too similar
```

Increases the readability of the code.

C.2.7 Names representing constants (final variables) must be all uppercase using underscore to separate words.

```
CLOSED_STATUS, NUMBER_OF_PLAYERS
```

Constants can be made public and do not need getters/setters to access them. There use should be clear in their name.

C.2.8 Method names must be verbs and written in mixed case starting with lowercase.

```
getClosedStatus()
calculateVolume()
```

Method naming convention used by Sun for the Java core packages.

C.2.9 Acronyms and abbreviations should not be uppercase when used as a name.

```
importXmlFile()
generateHtml()
```

Follows previous naming standards and improves readability.

C.2.10 Generic variables (including method parameter names) should have the same name as their type.

```
Car car;
```

as opposed to..

```
Car vehicle;
```

```
private void connectJms(JMSConnection jmsConnection) {...}
```

as opposed to..

```
private void connectJms(JMSConnection aJmsConnection) {...}
```

This reduces the complexity and number of different names used.

C.2.11 Don’t reuse variable names

For example if you have a class attribute named firstName do not create a local variable called firstName

Increases readability of code and reduced chance of coding errors

C.2.12 All names should be written in English.

English should be used for international development.

C.2.13 The name of the object is implicit and should be avoided in a method name.

```
car.getMaxSpeed();
```

as opposed to...

```
car.getCarMaxSpeed();
```

Including the object name in a method name is superfluous.

C.2.14 Interface names should be in camel case. They should be named after the operation that they do

```
Comparable
Enumerable
```

This makes the behaviour of any class that implements the interface clear.

C.2.15 By convention, type parameter names are single, uppercase letters

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

Makes it clear what is a Type and what is a Class.

C.3 Specific Naming Conventions

C.4.18 The terms get /set must be used where an attribute is accessed directly.

```
car.getMaxSpeed();
account.setBar();
```

Convention used by Sun for the Java core packages.

C.4.19 is prefix should be used for Boolean variables and methods.

```
private boolean isOverdrawn() {...}
private boolean isComplete;
```

Convention used by Sun for the Java core packages.

has, can and should can also be used where appropriate e.g.

```
private boolean hasOverdraft;
private boolean canAuthorise;
private boolean shouldAbort;
```

C.4.20 The term compute can be used in methods where something is computed.

```
lifeInsurance.computeRisk();
particle.computeEnergy();
```

Useful for identifying potentially time consuming and CPU intensive operations and is a candidate for caching the result if used repeatedly. Consistent use will enhance readability.

C.4.21 The term find can be used in methods where something is looked up.

```
dictionary.findEntry(String entry);
node.findShortestPath(Node destinationNode);
```

Useful for identifying simple look up method with minimum of computations involved. Consistent use will enhance readability.

C.4.22 UI variables should be suffixed by the element type.

```
postcodeTextField
addressesScrollbar
```

```
offersPanel
fileToggle
```

Enhances readability and gives the user an immediate clue of the type of the variable and the operations that can be performed on the object.

C.4.23 Plural form should be used on names representing collections.

```
Collection<Address> addresses;
int[] scores;
```

Enhances readability and gives the user an immediate clue of the type of the variable and the operations that can be performed on its elements.

C.4.24 Iterator variables should be called I, j, k etc.

```
for (Iterator i= addresses.iterator(); i.hasNext();) {
...
}

for (int i=0; i < nNames; i++) {
...
}
```

Mathematical convention for indicating iterators.

Variables *j, k* etc. should be used for nested loops.

C.4.25 Complement names must be used for complement entities.

```
get/set, add/remove, create/destroy, start/stop,
insert/delete, increment/decrement, old/new, begin/end,
first/last, up/down, min/max, open/close, show/hide,
suspend/resume
```

Reduce complexity by symmetry.

C.4.26 Only well known abbreviations should be used

```
computeAverage(); //NOT compAvg();
copyAddress(); //NOT cpAddress();
```

Abbreviations can make the code difficult to understand to anyone not familiar with them. There are certain abbreviations that are sufficiently common to use e.g.

Application > App

However abbreviating things like

Database > dtbse

should be avoided.

When abbreviating be aware of services / tools such as Spring and Hibernate that rely on convention.

C.4.27 Negated Boolean variable names must be avoided.

```
boolean isError; //NOT isNotError
boolean isFound; //NOT isNotFound
```

Enhances readability. Problems occur when the logical not operator is used and a double negative arises. Not immediately clear what !isNotError means.

C.4.28 Associated constants (final variables) should be represented using an Enum

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY
}
```

```

```

Instead of

```
public final static String MONDAY = “MONDAY”;
public final static String TUESDAY = “TUESDAY”;
public final static String WEDNESDAY = “WEDNESDAY”;
...
```

If the days are represented as Strings then it would be possible to assign an incorrect value

```
String day = “MOONDAY”
```

Enums ensure that incorrect values cannot be used e.g.

```
Day day = Day.MOONDAY;
```

C.4.29 Exception classes should be suffixed with Exception.

```
class AccessException extends Exception { … }
```

Exception class naming convention used by Sun for the Java core packages.

C.4.30 Default interface implementations can be prefixed by Default.

```
class DefaultTableCellRenderer implements TableCellRenderer {
… }
```

It is not uncommon to create a simplistic class implementation of an interface providing default behaviour to the interface methods. The convention of prefixing these classes by ***Default*** has been adopted by Sun for the Java library.

C.4.31 Singleton classes should return their sole instance through method getInstance.

```
class UnitManager {
```

```
    private final static UnitManager instance = new
                                            UnitManager();
    private UnitManager() { ... }
    public static UnitManager getInstance() { // NOT:  get()
or instance() or unitManager() etc.
       return instance;
    }
}
```

Common practice in the Java community.

C.4.32 Classes that creates instances on behalf of others (factories) can do so through method new[ClassName]

```
class CarFactory { public Car newCar(...) { ... } }
```

Indicates that the instance is created by new inside the factory method and that the construct is a controlled replacement of new Car().

C.4.33 Functions (methods returning an object) should be named after what they return and procedures (void methods) after what they do.

```
public Car getCar() {…}
public void respray(int colour) {…}
```

Increase readability. Makes it clear what the unit should do and especially all the things it is *not* supposed to do. This again makes it easier to keep the code clean of side effects.

C.4.34 Unit tests should be named *Test.java

```
EmployeeDaoTest.java
```

Allows a class to be clearly identified as a Test class.

C.4 Comments

This section contains standards for how we should document our code. It is important that code is documented correctly in order to allow JavaDoc to be correctly produced and increase the maintainability of code.

C.4.35 Comments should be applied pragmatically

Comments should be used to explain tricky code or calculations. Classes should not be overly commented. Try to make the code as self-documenting as possible through appropriate name choices and logical code structure.

C.1.1 All comments should be written in English

English is the preferred language.

C.1.2 JavaDoc comments should have the following form:

```
/**
 * Return lateral location of the specified position.
 * If the position is unset, NaN is returned.
 *
 * @param x    X coordinate of position.
 * @param y    Y coordinate of position.
 * @param zone Zone of position.
 * @return Lateral location.
 * @throws IllegalArgumentException If zone is <= 0.
 */
public double computeLocation(double x, double y, int zone)
throws IllegalArgumentException { ... }
```

A readable form is important because this type of documentation is typically read more often *inside* the code than it is as processed text.

Note in particular:

- The opening /** on a separate line
- Subsequent * is aligned with the first one
- Space after each *
- Empty line between description and parameter section.
- Alignment of parameter descriptions.
- Punctuation behind each parameter description.
- No blank line between the documentation block and the method/class.
- JavaDoc of class members can be specified on a single line as follows:

```
  /** Number of connections to this database */
  private int nConnections;
```

C.1.3 There should be a space after the comment identifier.

```
// This is a comment

/**
 * This is a JavaDoc
 * comment
 */
```

Enhances readability.

C.1.4 Use // for all non-JavaDoc comments, including multi-line comments.

```
// Comment spanning
// more than one line.
```

Use // comments ensure that it is always possible to comment out entire sections of a file using /* */ for debugging purposes etc.

C.1.5 Comments should be indented relative to their position in the code[1].

```
while (true) {
    // Do something
    something();
}
```

Enhances readability.

C.1.6 All public classes and public and protected functions within public classes should be documented using the Java documentation (JavaDoc) conventions.

This makes it easy to keep up-to-date online code documentation.

C.1.7 There should not be TODOs in production code

There should be no TODOs in production code. Instead tasks should be raised.

C.1.8 There should not be commented out code in production code

All commented out code should be removed in production code.

C.5 Layout

This section contains standards for how Java code should be laid out. This can be checked automatically using Checkstyle. Positioning of braces etc. often causes debate. The key thing is that layout is consistent across the code base for a particular project.

C.5.1 Curly brace “{“ position should be consistent.

```
class MyClass {
    MyClass() {
    }

    public void myMethod() {
        for (int i=0; i<10; i++) {
            // Do something...
        }
    }
}
```

Or

```
class MyClass
{
    MyClass()
    {
    }

    public void myMethod()
    {
        for (int i=0; i<10; i++)
        {
            // Do something...
        }
    }
}
```

It doesn’t matter which one you use, curly brace on the line, or below it. What matters is that you are consistent across the project.

C.5.2 Four space indent preferred.

```
    public void myMethod() {
```

```
        for (int i=0; i<10; i++) {
            // Do something...
        }
    }
```

Rather than

```
  public void myMethod() {
      for (int i=0; i<10; i++) {
        // Do something...
      }
  }
```

You must indent code and do it consistently to ensure the code is readable.

It doesn't actually matter whether it's 2 or 4 spaces as long as you are consistent, but 4 spaces usually gives greater clarity.

C.5.3 Special characters like TAB and page break must be avoided

These characters are likely to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.

C.5.4 Always use braces for if statements

```
if (condition) {
    statement;
}

As opposed to:

if (condition)
    statement;
```

Enhances readability and maintainability.

C.5.5 The for statement should have the following form:

```
for (initialization; condition; update) {
    statements;
}

Or

for(variable : iterator) {
    statements;
}
```

Enhances readability and maintainability.

C.5.6 The while statement should have the following form:

```
while (condition) {
    statements;
}
```

Enhances readability and maintainability.

C.5.7 The switch statement should have the following form:

```
switch (condition) {
    case ABC :
        statements;
        /* Falls through*/
    case DEF :
        statements;
```

```
        break;
    case XYZ :
        statements;
        break;
    default :
        statements;
        break;
}
```

This differs slightly from the Java Sun recommendation with regard to indentation. Each case keyword is indented relative to the switch statement and there is an extra space before the : character.

The `/* falls through */` comment should be included whenever a break statement has been omitted and shows that it was intentional. Missing the break statement is a common error.

Every `switch` statement should include a `default` case. The `break` in the `default` case is redundant, but it prevents a fall through if another case is added later.

C.5.8 A try-catch statement should have the following form:

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}

try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
// Or use the Java 7 Exception Handler
```

```
try {
    statements;
} catch (FileNotFoundException | EOFExection |
         ObjectStreamException | IOException ex) {
     statements;
}
```

Enhances readability and maintainability.

C.6 White Space

C.6.1 - Operators should be surrounded by a space character.

- Java reserved words should be followed by a white space.
- Commas should be followed by a white space.
- Colons should be surrounded by white space.
- Semicolons in for statements should be followed by a space character.

```
a = (b + c) * d; // NOT: a=(b+c)*d

while (true) { // NOT: while(true) { ...

doSomething(a, b, c, d); // NOT: doSomething(a,b,c,d);

case 100 : // NOT: case 100:

for (i = 0; i < 10; i++) { // NOT: for(i=0;i<10;i++) {
```

Enhances readability. This is not a complete list and indicates the general intentions to improve readability.

C.6.2 Logical units within a block should be separated by one blank line.

```
// Create a new identity matrix
Matrix4x4 matrix = new Matrix4x4();

// Precompute angles for efficiency
Double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

// Specify matrix as a rotation transformation
matrix.setElement(1, 1, cosAngle);
matrix.setElement(1, 2, sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2, cosAngle);

// Apply rotation
transformation.multiply(matrix);
```

Enhances readability by introducing white space between logical units. Each block is often introduced by a comment as indicated in the example above.

C.6.3 Methods should be separated by three blank lines.

Enhances readability and ensures methods stand out within the class.

C.6.4 Variables in declarations can be left aligned.

```
TextFile file;
int      nPoints;
double   x;
```

Enhances readability. The variables are easier to spot from the types by alignment.

C.6.5 Statements should be aligned wherever this enhances readability.

```
minPosition     = computeDistance(min,     x, y, z);
averagePosition = computeDistance(average, x, y, z);
```

Alignment should be used to enhance readability but shouldn't violate the other code conventions.

C.7 Statements

This section contains standards for how various java constructs should be uses

C.7.1 All java files should belong to a specific package.

Ensuring files belong to a specified package (rather than the default package) ensures the project is structured and maintainable.

C.7.2 Import statements should be ordered, grouped with associated packages together and a blank line between groups.

```
Use the following order:
java
javax
org
com
```

Enhances readability and maintainability.

C.7.3 Imported classes should be listed explicitly.

```
import java.util.List; // NOT import java.util.*;
import java.util.ArrayList;
import java.util.HashSet;
```

Enhances readability and maintainability.

C.8 Classes and Interfaces

C.8.1 Class and Interface declarations should be organized in the following manner:

1. Class/Interface documentation.
2. class or interface statement.
3. Class (static) variables in the order public, protected, package (no access modifier), private.
4. Instance variables in the order public, protected, package (no access modifier), private.
5. Constructors.
6. Methods (no specific order).
7. Inner Class.

Enhances readability and maintainability.

C.9 Methods

C.10.3 Method modifiers should be given in the following order:

<access> static abstract synchronized <unusual> final native
The <access> modifier (if present) must be the first modifier.

```
public static double square(double a) {…}
```

as opposed to

```
static public double square(double a) {…}
```

Where <access> is public, protected or private.

Where <unusual> includes volatile and transient.

This improves readability and maintainability.

C.10.4 Avoid methods longer than 15 lines

Methods should be less than 15 lines long. Break up long methods into small methods. This promotes code reuse and allows for more combinations of methods. If the number of methods grows to be difficult to understand, then look at decomposing the class into more than one class.

C.10 Types

C.10.1 Type conversions must always be done explicitly. Never rely on implicit type conversion.

```
floatValue = (float) intValue; // NOT: floatValue = intValue;
```

By this, the programmer indicates that he is aware of the different types involved and that the mix is intentional.

C.10.2 Array specifiers must be attached to the type not the variable.

```
int[] a = new int[20]; // NOT: int a[] = new int[20]
```

The array is a feature of the base type, not the variable.

C.11 Variables

C.11.1 Variables should be initialised where they are declared and they should be declared in the smallest scope possible.

This ensures that variables are valid at any time. Sometimes it is impossible to initialise a variable to a valid value where it is declared, so it should be left uninitialized.

Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.

C.11.2 Variables must never have dual meaning.

Enhances readability and maintainability. Reduces chance of error by side effects.

C.11.3 Place constants on left side of evaluation expressions

```
if(”value”.equals(x)) {
..
}
```

This form provides protection against null pointer exceptions. For example, the following example will throw a NPE:

```
String x;
X = null;
if(x.equals(“1”) {
..
}
```

The constant on the left prevents this and returns the correct result:

```
String x;
X = null;
if(“1”.equals(x) {
..
}
```

C.11.4 Class variables should never be declared public.

The concept of Java information hiding and encapsulation is violated by public variables. Use private variables and access functions instead.

C.12 Loops

C.12.1 Only loop control statements must be included in the for()construction.

```
sum=0;  //NOT for (i=0,sum=0;i<100;i++)
for (i=0; i < 100; i++) {
 sum += values[i];
}
```

Enhance readability and maintainability. Separates loop control from loop content.

C.12.2 Loop variables should be initialised immediately before the loop.

```
isDone=false;
while (!isDone) {…}
```

Enhance readability and maintainability.

C.12.3 The use of do-while loops can be avoided.

do-while loops are less readable than *while* and *for* loops since the conditional is at the bottom of the loop.

Any *do-while* loop can be rewritten into a while or for loop thus reducing the number of constructs used to enhance readability and maintainability.

C.12.4 The use of break and continue in loops should be avoided.

Break should be avoided, unless it will improve performance and is clearly documented. Continue is less likely to offer this and should be avoided.

C.13 Conditionals

C.13.1 Complex conditional expressions must be avoided. Introduce temporary boolean variables instead [1].

```
boolean isFinished = (elementNo < 0) || (elementNo >
maxElement);
boolean isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) { … }
```

as opposed to

```
if ((elementNo < 0) || (elementNo > maxElement)|| elementNo
== lastElement) { … }
```

Enhance readability and maintainability.

C.13.2 The most likely case should be put in the if-part and the least likely in the else-part of an if statement [1].

```
boolean isOk = readFile(fileName);
if (isOk) { … }
else { … }
```

Enhances performance, readability and maintainability.

C.13.3 The conditional should be put on a separate line.

```
if (isDone) {
    doCleanup();
}

As opposed to:

if (isDone) doCleanup();
```

This aids debugging and makes it clear as to whether the check is true or not.

C.13.4 Executable statements in conditionals must be avoided.

```
InputStream stream = File.open(fileName, "w");
if (stream != null) { … }

As opposed to:

if (File.open(fileName, "w") != null)) { … }
```

Enhances readability and maintainability.

C.13.5 Use of the ternary operator (?) should be kept simple

Brackets should be used around the conditional

```
(a > b) ? 1 : 2;
```

Improves the clarity of the code

C.13.6 Nested ternary (?) operators can be confusing and should be avoided

```
(x<y && x<z) ? x : (y<x && y<z) ? y : (z<y && z<x) ? z
```

Improves the clarity of the code

Appendix H. Useful tools

This section outlines some useful tools and how to configure them into your build.

E.1 Find Bugs

Find bugs searches Java bytecode for potential issues and coding errors. A detailed description can be found here

http://findbugs.sourceforge.net/factSheet.html

Follow the instructions here to install FindBugs

http://findbugs.sourceforge.net/manual/installing.html#d0e98

3.6.1.1 Ant

Add the following task to your projects build.xml

```
<taskdef name="findbugs"
classname="edu.umd.cs.findbugs.anttask.FindBugsTask"/>    <property
name="findbugs.home" value="/export/home/work/findbugs" />
<target name="findbugs" depends="jar">
    <findbugs home="${findbugs.home}" output="xml" outputFile="projecta-
fb.xml" >
        <auxClasspath path="${basedir}/lib/Regex.jar" />
        <sourcePath path="${basedir}/src/java" />
        <class location="${basedir}/bin/projecta.jar" />
        <effort>max</effort>
        <threshold>low</threshold>
    </findbugs>
</target>
```

See http://findbugs.sourceforge.net/manual/anttask.html for more details

3.6.1.2 Maven

To use find bugs with Maven add the following plugin to the reporting section

```
      <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>findbugs-maven-plugin</artifactId>
                <version>2.5.2</version>
```

```
                <configuration>
                                <effort>max</effort>
                                <threshold>low</threshold>
                                <xmlOutput>true</xmlOutput>

                                <fork>true</fork>
                                <maxHeap>2048</maxHeap>
                <configuration>
</plugin>
```

http://mojo.codehaus.org/findbugs-maven-plugin/

3.6.1.3 Jenkins

It is highly recommend that FindBugs is run regularly, ideally as part of a continuous integration server such as Hudson or Jenkins. A plugin exists

http://wiki.hudson-ci.org/display/HUDSON/FindBugs+Plugin

which will provide a report for each build highlighting any issues

3.6.2 Checkstyle

Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. A detailed description can be found here

http://checkstyle.sourceforge.net/

3.6.2.1 Ant

Add the following to your project's build.xml

```
<taskdef resource="checkstyletask.properties"
classpath="/path/to/checkstyle-5.6-all.jar"/>  <checkstyle
config="docs/sun_checks.xml" file="Check.java"/>
```

Detailed instructions can be found here http://checkstyle.sourceforge.net/anttask.html

3.6.2.2 Maven

Simply add the following to the reporting section your project's POM

```
     <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
```

```
        <version>2.10</version>
      </plugin>
```

http://maven.apache.org/plugins/maven-checkstyle-plugin/

3.6.2.3 Jenkins

As with FindBugs the Checkstyle tasks should ideally be run by a continuous integration server such as Hudson or Jenkins. A good plugin exists that will produce an easily accessible report on any Checkstyle violations

http://wiki.hudson-ci.org/display/HUDSON/Checkstyle+Plugin

3.6.3 PMD

A third tool of use is PMD. PMD is a source code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth.

http://pmd.sourceforge.net/

3.6.4 SonarQube

SonarQube is an open platform to manage code quality, As such, it covers the 7 axes of code quality : Architecture and Design, Duplications, Unit Testing, Complexity, Potential bugs, Coding rules and Comments.

http://www.sonarqube.org

3.6.5 Cobertura

Cobertura provides code coverage for Java projects. It can provide percentages based on class coverage, conditional coverage and package coverage. It integrates nicely with Jenkins.

http://cobertura.github.io/cobertura/

3.6.5.1 Maven

```
<plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>cobertura-maven-plugin</artifactId>
                <version>2.6</version>
                <configuration>
                                <aggregate>true</aggregate>
                                <format> xml</format>
```

```
                                <maxmem>2048</maxmem>
                </configuration>
</plugin>
```

3.6.6 JavaNCSS

JavaNCSS (http://www.kclee.de/clemens/java/javancss/) stands for Java Non Commenting Source Statements, its purpose is to provide the number of SLOC within a project/module and break down those lines into java documentation, multiple line comments, single line comments and source code. When integrated within Jenkins you can use it to make a build as unstable/failed if the number of code comments drops above or below a threshold.

3.6.6.1 Ant

```
<taskdef name="javancss"
            classname="javancss.JavancssAntTask"
            classpath="/export/home/work/javancss"/>

<javancss srcdir="${basedir}/src/java"
            generateReport="true"
            outputfile="${test.results.root}/javancss_metrics.xml"
            format="xml"/>
```

3.6.6.2 Maven

```
<plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>javancss-maven-plugin</artifactId>
                <version>2.0-beta-2</version>
</plugin>
```