Software Technology 151/501, CRICOS number: 00301J

ST151/501 Java Coding Standard
Department Of Computi ng,
Curtin University of Technol ogy

This document was last modified on 3rd July 2002

1.0 Copyright Notices

The material was modified from *C++ Coding Guidelines* which can de downloaded from the IPE152/502 web area. Special acknowledgement is given to Michael Borck who wrote the C++ coding standards document.

2.0 Referenced Documents

Please see the C++ coding standards document for a list of references used to generate that document. This document was modifed from the C++ document.

3.0 Contribut ions

This document was produced with contributions from a number of academic staff members within the department as well as being loosely based upon the GNU coding standard.

4.0 Definiti ons

The term *method* is used in this document to mean a callable block of code that performs one or more actions. This is a general definition and makes no distinction whether or not the ``block of code'' returns a value. If this distinction is important it will be explicitly stated. Based on this definition the term *method* is a synonym for a *function*, *procedure or subroutine*.

5.0 Introduct ion

Coding standards define a consistent look and feel in your code. These guidelines are a set of rules describing how your code should look, which features of the programming language you will use and how. The important thing isn't which set of conventions is better, but rather to have a standard and use it consistently.

These guidelines will not only help you in reading code from your instructor and classmates [1] but will help your tutors as they read (and grade!) your code. These guidelines should make you a more productive programmer because you do not have to make decisions about trivial matters. You can spend your energy on the solution of real problems. It is also important to

1 This is when you *help (i.e. critise/review)* not *copy* someone else's code as that would be plagiarism.

remember that most software firms follow strict coding standards, and this is one of the ways to simulate the real world as closely as possible.

You are expected to conform to these Coding Guidelines in all your programming assignments. Deviations from this standard are acceptable if they enhance readability and code maintainability. Major deviations required a explanatory comment at each point of departure so the person assessing your code will know that you didn't make a mistake, but purposefully are doing a local variation for a good cause. If you ignore these guidelines then your code will be branded as *unacceptable* and will not be marked.

Although the Java language is used throughout this document, every attempt has been made to make the guideline as generic as possible. Most of the concepts mentioned here are valid for and can be applied to many programming languages. There is a matching C++ coding standard which can be found on the IPE152/502 web area.

6. Summary of Guidel ines.

These guidelines may mention features that you have not seen in the class. Here are the most important highlights (see relevant section for details):

- tab are set every four spaces.
- variable names are lowercase.
- class names are capitalised (e.g. public class ThisIsAClass).
- method and variable names are capitalised but always start with a lowercase letter (e.g. public void thisIsAMethod()).
- constants are all uppercase.
- use a space after keywords, after every comma.
- there are spaces between binary operators.
- paired braces must line up.
- no literal numbers may be used, use symbolic constants instead (e.g not 3.14159 but Math.PI).
- every method must have a set of comments.
- methods should be at most 30-50 lines in length.
- no goto, continue, break (except in switch statements) or multiple returns are allowed.
- for loop indexes should be declared as local to the for loop.
- lines should not exceed 78 characters in length.

7.0 Lexical Issues

7.1 Naming Conventions

One way to improve the readability of your code is to follow certain naming conventions that show what category an identifier is in. You should be able to tell at first glance whether an identifier is a constant or a variable. Following the Java conventions aids this task but the names that you select for identifiers must also reflect this.

Generally all names must be written in English. Avoid names that don't mean anything (e.g. *foo*) and single character variable names, the exception being loop counters which may use variable names such as *i, j*, and *k*. Any numbers in a name must be written using letters, not numbers, (e.g. *ValueOne*) unless there is a good reason not to. The following rules specify when to use upper- and lowercase letters in identifier names.

Files: Each .java file should contain one class definition and its name should consist of the class name followed by the file extension *.java*.

Methods: Methods can be made up from several words, the first letter of each word in uppercase and the rest in lowercase. The first word must be all lowercase. Acronyms should not be used.

Example:
```
public void setText( const QString &text )
```

Classes: Class names can be made up from several words, the first letter of each word in uppercase and the rest in lowercase.

Examples:
```
public class Dialog;
public class ModuleManager;
```

Constants: Constants are written in uppercase, multiple words divided by underscore.

Examples:
```
public static final double PI = 3.1415;
public static final int MEANING_OF_LIFE = 42;
```

Variables: Variables are named in the same way as methods.

Examples:
```
int i, index, x, y;
int xAdd;
```

7.2 Parentheses, Braces and Comma

Use a space after every comma. Use a space after a begin parenthesis, and a space before an end parenthesis. Do not use such spaces for square brackets. Use spaces between binary operators, do not use spaces with unary operators.

Put begin and end braces {} in the same column, on separate lines directly before and after the block.

For example:

```
private void meaningOfLife( int age, int year )
    {
    if ( age == year )
        {
          ...
        }
    else
        {
          ...
        }
    }
```

7.3 White Space and Indentation

An important way of making your code easy to read is to space it out in a neat and orderly manner.

Use empty lines to group variables and code lines that logically belong together. Variable names that are grouped this way must be on the same column. Empty lines must be empty, containing no *TAB*s or spaces. Allow one blank line between a block comment and the method it belongs to. Allow four to six lines at the end of a method before the next block comment. Separate logical chunks of code within a method with blank lines.

Use four character indentation, this is commonly used and allows for easy recognition of the indentation level. Use spaces instead of TABs, to assure the code looks the same everywhere.

Lines should not exceed 78 characters as this will ensure that printed output will match the screen on all terminals and printers.

8.0 Statements

8.1 Flow Control

if, else, while, for, and *do* should be followed by a block (i.e. must use braces { }). If you have several consecutive *if* and *else if* statements, consider using *case* statements instead.

if statements, *while*, *do/while* and *for* loops can only execute boolean expressions, they should never assign or change any variables. The ONLY exception to this is the *for* loop when it modifies the loop index.

8.2 Return Statement

Having a single return statement at the end of a method has the advantage that there is a single, known point which is passed through at the termination of execution of the method. If forces you to write structured code, e.g. using *if-then-else* statements *properly*.

Never use multiple returns. It will not be tolerated. Think about how these multiple returns would be represented as in your algorithm (i.e. *goto*!).

Poor Coding:

```
for ( int i = 0 ; i < max ; i++ )
    {
    if ( record[i] == key )
        {
        return TRUE;
        }
   }
return FALSE;
```

Good Coding:

```
int i = 0;
found = FALSE;
while ( ( i < max ) && ( !found ) )
    {
    if ( record[i] == key )
        {
        found = TRUE;
        }
    i++
    }
return found;
```

The *GOOD EXAMPLE* has the advantage that if we later decide that there is more to do after the search, we do not need to modify the *while* loop but just place the new code between the while loop and the return statement.

8.3 Switch

In a *switch* statement each *case* should be terminated with a *break* statement. All *switch* statements should have a default case. All *drop through cases* should be clearly documented in a comment before the switch statement. This technique should only be used when it makes the code simpler and clearer.

8.4 Goto, continue, and break

Goto, continue and break statements should not be used. The only exception is using the *break* keyword within a switch statement (see section 8.3 above).

8.5 For loop indexes

For loop indexes should always be declared as local to the for loop (see previous coding example for the syntax). This ensures they can never be referenced outside the for loop. Remember that one of the properties of a FOR loop is that the value of the loop index after the loop is undefined.

9.0 Variables

9.1 Numeric Literals

Avoid the use of numeric literal values. A numeric literal is a constant used in source code without a constant definition. Constants should be declared as both public and global to the class. For example:

Poor Coding:

```
    if ( customer == 16 )
        {
        processTicket();
        }
    else if ( customer == 7 )
        {
        processRefund();
        }
    else
        {
        opps("How did I get here?");
        }
```

Good Coding:

```
    public static final int PAID       = 16;
    public static final int COMPLAINED = 7;

    if ( customer == PAID )
        {
        processTicket();
        }
    else if ( customer == COMPLAINED )
        {
        processRefund();
        }
    else
        {
        opps("How did I get here?");
        }
```

In the above example what do ``16" and ``7" mean? If there was a number change, or the numbers were just plain wrong, how would you know? Instead of numbers, use a real name that means something, as shown in the *good coding example.*

9.2 Initialisation

Variables should be declared with the smallest possible scope. Ensure every variable is given a value before use. Wherever possible initialise the variable close to where the decision is made about what its value will be, or where it is used, rather than where it is declared. This means that every class field should be initialised in all of the constructors for the class. If this is not the case for a particular class field, ask yourself whether that item of data should be represented by a class field or declared as a variable which is local to a particular method.

This makes it clear that you have considered all paths through the code and the variable is initialised correctly, ensures that variables are not unnecessarily allocated and reduces the risk of a variable being inadvertently hidden.

Poor Coding:

```
int max = 0;
   .
   .
   .
for ( int i=0 ; i < n ; i++ )
    {
    if ( age[i] > max )
        {
        max = age[i];
        }
    }
```

Good Coding:

```
int max;
   .
   .
   .
max = 0;
for ( int i=0 ; i < n ; i++ )
    {
    if ( age[i] > max )
        {
        max = age[i];
        }
    }
```

The initialisation of max is located close to where it is algorithmically significant in the Good Coding example whereas it is harder to notice in the Bad Coding example (given that there could be a large number of lines of code between the declaration of max and the for loop).

10.0 Methods

10.1 Method Length

Methods should be short and do one task. The length of the method depends on how complex it is. The more complex the method is, the shorter it should be. Too long methods should be split into several methods each doing a minor task. A method should not be longer than 30-50 lines. If similar chunks of code appear multiple time in your code, you should probably write a method to handle that operations. Note that good use of stepwise refinement (i.e. top down decomposition) will result in achieving the above requirement as a natural part of the design process.

10.2 Method Comments

For every method, write a comment that explains:

- what it does.
- pre- and post-conditions.
- the import/export information.
- any other general comments

For example:

```
// NAME:     DateToJulian
// PURPOSE:  Convert calendar date into Julian date
// IMPORTS:  d, m, y the day, month and year
// EXPORTS:  The Julian day number that begins at noon of the given
//           calendar date
// Assertions:
//     Pre:  d, m and y all represent a valid date.
//     Post  Julian date will be day number will be valid.
// REMARKS:  This algorithm is translated from Press et all.,
//           Numerical Recipes in C, 2nd ed., Cambridge University
//           Press, 1992.
```

10.3 Parameters

If a method has many parameters it might be wise to split the parameter line into several lines, placing the parameters with a close relation to each other on the same line. Many parameters is often a sign that your method is too big (i.e. high coupling, low cohesion) or that an object is required.

11.0 Classes

A Class should only represent one thing. Split it into subclasses if it's too big.

All class fields must be declared as private. When access to member variables is required, use accessor/mutator methods; never use public or protected variables. In other words each class should have a clearly defined public interface and everything else should be private.

All methods which are not part of the public interface for the class should be declared private.

12.0 Exceptions

Use exceptions to control things that don't work as they were expected to, like parameters having values they should never have. Don't use exceptions as a way of leaving loops or nested method calls. Remember that exceptions are used for handling exceptional circumstances and usually involve terminating the program is a controlled manner. They should not be used for anything else. Methods that use exceptions should explicitly state so in the documentation.

13.0 File Organisati on

It is important that, within each source code file you keep a consistent ordering of code components.

13.1 Introductory Comment

Every file must be documented with an introductory comment that provides:
- information on the file;
- author, user name and unit;
- its purpose;
- references;
- general comments;
- other required files;
- date of last modification;

For example:

```
// FILE:       Lotto.java
// AUTHOR:     Jon Post
// USERNAME:   postj
// UNIT:       ST151
// PURPOSE:    Provides a class for the generation of between
//             1 and 18 sets of lotto numbers.
// REFERENCE:  None.
// COMMENTS:   The algorithm uses simple random number generation,
//             to generate the numbers.
// REQUIRES:   Makes use of LottoSet.java which contains a class
//             used to contain of set of six lotto numbers.
// Last Mod:   3rd March 2001
```

13.2 Ordering of Java Code:

The suggested order of sections for a program file is as follows:
- Introductory Comment;
- Declaration of class fields;
- Constructors;
- Methods which form the class interface (i.e. accessors/ mutators etc);

Private methods;

14.0 The Readme File

A *README* file should accompany each of your programs. This is a *text* file that explains the overall design of your project. When presenting your design you should discuss why you made the design choices you did. Anyone should be able to explain and defend your design by simply reading your *README* (without looking at your code). You should also discuss all known bugs, speculate about their cause, and suggest how they could be fixed. Finally you should include a description of any functionality that goes beyond the required specifications (i.e all the bells and whistles you added).