**Clean Code:**

**There are many aspects to writing clean code, some of which are outlined below. You should adopt good practices and stick with them. Writing clean code is critical from a code readability, reliability and security perspective.**

* **Logic before coding (work out problem, write pseudocode)**
* **Naming convention (See below)**
* **Small tasks (Avoid large functions, each method should achieve a small task)**
* **Comment your code (comments relating to methods e.g. Javadoc style)**
* **Indentation (You learned this in first year)**
* **Testing all functionality (Unit testing…….driver programs)**

**Code**

* Use plenty of white space to clarify code.
* Indent properly to show structure, and be consistent with your indentation throughout the entire program.
* Put statements ending with semicolons and { } on separate lines (except for loops).

**for (i=o;i<5;i++)**

**{ cout<<i\*j; j=j-3; }…………..poor practice**

**for (i = 0; i < 5; i++){**

**cout << i \* j;**

**j = j - 3;**

**} //end for……………………..good practice**

**Variables / Constants**

* Chose variable names and constants that are self-documenting. Use only single letter variables for counting.

*Example:* **c = a - b;** vs. **score = total\_shots - total\_missed;**

* Constants should generally be all capitalized, and variables should generally be all lower case (camel-casing is also acceptable for variables; just be consistent).

*Example:* **const int NAME\_LEN = 4;** and **int num\_of\_students = 20**; **// or numOfStudents**

* Use constants for all “magic numbers” or whenever a particular value is used more than once.

**Documentation**

* Always place your name, date, and program description at the top of the program.
* Write comments that express more of the *why* than the *how*. Document the code’s intent.

*Example:* **// Divide total grades by total students**

vs. **// Compute student grade average**

* Make your comments say something about the code that the code can’t say about itself.

*Example:* **level++; // Add 1 to level**

vs. **level++; // Advance to next level before processing remaining lives**

* Typically a comment should refer to a logical grouping of lines rather than a single line of code.
* Avoid all but the most common abbreviations in comments so they’re easy to read.
* Clearly separate comments from code.
* Use a commenting style that is not overly tedious, time-consuming, or a maintenance monster.
* Outline your code with comments before you write it rather than doing all of the documentation at the end. You’ll save yourself time from having to figure out tricky places or forgetting details, assumptions, and subtleties of design.

**Methods**

* method names should usually consist of a verb and a noun which describe the function’s purpose. *Example:* **cms()** vs. **calcManagerSalary()**
* A method should have a single purpose. *Example:* Figuring an average, printing a student list, and getting input in one method vs. three functions which perform each of those actions.
* Use method to reduce redundancy in your programs. If you have three code segments that do nearly the exact the same thing, that code is a good candidate for a method.

**http://www.horstmann.com/bigj/style.html  
Appendix A1  
Java Language Coding Guidelines**

**A1.1  Introduction**

This coding style guide is a simplified version of one that has been used with good success both in industrial practice and for college courses.

A style guide is a set of mandatory requirements for layout an d formatting. Uniform style makes it easier for you to read code from your instructor and classmates. You will really appreciate that if you do a team project. It is also easier for your instructor and your grader to grasp the essence of your programs quickly.

A style guide makes you a more productive programmer because it *reduces gratuitous choice.* If you don't have to make choices about trivial matters, you can spend your energy on the solution of real problems.

In these guidelines, several constructs are plainly outlawed. That doesn't mean that programmers using them are evil or incompetent. It does mean that the constructs are not essential and can be expressed just as well or even better with other language constructs.

If you already have programming experience, in Java or another language, you may be initially uncomfortable at giving up some fond habits. However, it is a sign of professionalism to set aside personal preferences in minor matters and to compromise for the benefit of your group.

These guidelines are necessarily somewhat dull. They also mention features that you may not yet have seen in class. Here are the most important highlights:

* Tabs are set every three spaces.
* Variable and method names are lowercase, with occasional upperCase characters in the middle.
* Class names start with an Uppercase letter
* Constant names are UPPERCASE, with an occasional UNDER\_SCORE.
* There are spaces after keywords and surrounding binary operators.
* Braces must line up horizontally or vertically.
* No magic numbers may be used.
* Every method, except for main and overridden library methods, must have a comment.
* At most 30 lines of code may be used per method.
* No continue or break is allowed.
* All non-final variables must be private.

*Note to the instructor:* Of course, many programmers and organizations have strong feelings about coding style. If this style guide is incompatible with your own preferences or with local custom, please feel free to modify it. For that purpose, this coding style guide is available in electronic form from the author.

**A1.2  Source Files**

Each Java program is a collection of one or more source files. The executable program is obtained by compiling these files. Organize the material in each file as follows:

* package statement, if appropriate
* import statements
* A comment explaining the purpose of this file
* A public class
* Other classes, if appropriate

The comment explaining the purpose of this file should be in the format recognized by the javadoc utility. Start with a /\*\*, and use the @author and @version tags:

|  |  |
| --- | --- |
|  | /\*\*  COPYRIGHT (C) 1997 Harry Hacker. All Rights Reserved.  Classes to manipulate widgets.  Solves CS101 homework assignment #3  @author Harry Hacker  @version 1.01 1997-02-15  \*/ |

**A1.3  Classes**

Each class should be preceded by a class comment explaining the purpose of the class.

First list all public features, then all private features.

Within the public and private section, use the following order:

1. Constructors
2. Instance Methods
3. Static Methods
4. Instance Fields
5. Static Fields
6. Inner classes

Leave a blank line after every method.

All non-final variables must be private. (However, instance variables of a private inner class may be public.) Methods and final variables can be either public or private, as appropriate.

All features must be tagged public or private. Do not use the default visibility (that is, package visibility) or the protected attribute.

Avoid static variables (except final ones) whenever possible. In the rare instance that you need static variables, you are permitted one static variable per class.

**A1.4  Methods**

Every method (except for main) starts with a comment in javadoc format.

|  |  |
| --- | --- |
|  | /\*\*  Convert calendar date into Julian day.  Note: This algorithm is from Press et al., Numerical Recipes  in C, 2nd ed., Cambridge University Press, 1992  @param day day of the date to be converted  @param month month of the date to be converted  @param year year of the date to be converted  @return the Julian day number that begins at noon of the  given calendar date.  \*/  public static int dat2jul(int day, int month, int year)  {  . . .  } |

Methods must have at most 30 lines of code. The method signature, comments, blank lines, and lines containing only braces are not included in this count. This rule forces you to break up complex computations into separate methods.

**A1.5  Variables and Constants**

Do not define all variables at the beginning of a block:

|  |  |
| --- | --- |
|  | {  double xold; // Don't  double xnew;  boolean more;  . . .  } |

Define each variable just before it is used for the first time:

|  |  |
| --- | --- |
|  | {  . . .  double xold = Integer.parseInt(input);  boolean more = false;  while (more)  {  double xnew = (xold + a / xold) / 2; // OK  . . .  }  . . .  } |

Do not define two variables on the same line:

|  |  |
| --- | --- |
|  | int dimes = 0, nickels = 0; // Don't |

Instead, use two separate definitions:

|  |  |
| --- | --- |
|  | int dimes = 0; // OK  int nickels = 0; |

In Java, constants must be defined with the keyword final. If the constant is used by multiple methods, declare it as static final. It is a good idea to define static final variables as private if no other class has an interest in them.

Do not use *magic numbers!* A magic number is a numeric constant embedded in code, without a constant definition. Any number except -1, 0, 1, and 2 is considered magic:

|  |  |
| --- | --- |
|  | if (p.getX() < 300) // Don't |

Use final variables instead:

|  |  |
| --- | --- |
|  | final double WINDOW\_WIDTH = 300;  . . .  if (p.getX() < WINDOW\_WIDTH) // OK |

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

|  |  |
| --- | --- |
|  | public static final int DAYS\_PER\_YEAR = 365; |

so that you can easily produce a Martian version without trying to find all the 365s, 364s, 366s, 367s, and so on, in your code.

When declaring array variables, group the [] with the type, not the variable.

|  |  |
| --- | --- |
|  | int[] values; // OK  int values[]; // Ugh--this is an ugly holdover from C |

**A1.6  Control Flow**

**A1.6.1—The if Statement**

Avoid the "if ... if ... else" trap. The code

|  |  |
| --- | --- |
|  | if ( ... )  if ( ... ) ...;  else ...; |

will not do what the indentation level suggests, and it can take hours to find such a bug. Always use an extra pair of { ... } when dealing with "if ... if ... else":

|  |  |
| --- | --- |
|  | if ( ... )  {  if ( ... ) ...;  } // {...} are necessary  else ...;  if ( ... )  {  if ( ... ) ...;  else ...;  } // {...} not necessary, but they keep you out of trouble |

**A1.6.2—The for Statement**

Use for loops only when a variable runs from somewhere to somewhere with some constant increment/decrement:

|  |  |
| --- | --- |
|  | for (int i = 0; i < a.length; i++)  System.out.println(a[i]); |

Do not use the for loop for weird constructs such as

|  |  |
| --- | --- |
|  | for (a = a / 2; count < ITERATIONS; System.out.println(xnew))  // Don't |

Make such a loop into a while loop. That way, the sequence of instructions is much clearer.

|  |  |
| --- | --- |
|  | a = a / 2;  while (count < ITERATIONS) // OK  { . . .  System.out.println(xnew);  } |

**A1.6.3—Nonlinear Control Flow**

Avoid the switch statement, because it is easy to fall through accidentally to an unwanted case. Use if/else instead.

Avoid the break or continue statements. Use another boolean variable to control the execution flow.

**A1.6.4—Exceptions**

Do not tag a method with an overly general exception specification:

|  |  |
| --- | --- |
|  | Widget readWidget(Reader in)  throws Exception // Bad |

Instead, specifically declare any checked exceptions that your method may throw:

|  |  |
| --- | --- |
|  | Widget readWidget(Reader in)  throws IOException, MalformedWidgetException // Good |

Do not "squelch" exceptions:

|  |  |
| --- | --- |
|  | try  {  double price = in.readDouble();  }  catch (Exception e)  {} // Bad |

Beginners often make this mistake "to keep the compiler happy". If the current method is not appropriate for handling the exception, simply use a throws specification and let one of its callers handle it.

**A1.7  Lexical Issues**

**A1.7.1—Naming Convention**

The following rules specify when to use upper- and lowercase letters in identifier names.

* All variable and method names and all data fields of classes are in lowercase (maybe with an occasional upperCase in the middle); for example, firstPlayer.
* All constants are in uppercase (maybe with an occasional UNDER\_SCORE); for example, CLOCK\_RADIUS.
* All class and interface names start with uppercase and are followed by lowercase letters (maybe with an occasional UpperCase letter); for example, BankTeller.

Names must be reasonably long and descriptive. Use firstPlayer instead of fp. No drppng f vwls. Local variables that are fairly routine can be short (ch, i) as long as they are really just boring holders for an input character, a loop counter, and so on. Also, do not use ctr, c, cntr, cnt, c2 for variables in your method. Surely these variables all have specific purposes and can be named to remind the reader of them (for example, current, next, previous, result, . . . ).

**A1.7.2—Indentation and White Space**

Use tab stops every three columns. That means you will need to change the tab stop setting in your editor!

Use blank lines freely to separate parts of a method that are logically distinct.

Use a blank space around every binary operator:

|  |  |
| --- | --- |
|  | x1 = (-b - Math.sqrt(b \* b - 4 \* a \* c)) / (2 \* a); // Good  x1=(-b-Math.sqrt(b\*b-4\*a\*c))/(2\*a);//Bad |

Leave a blank space after (and not before) each comma or semicolon. Do not leave a space before or after a parenthesis or bracket in an expression. Leave spaces around the ( . . . ) part of an if, while, for, or catch statement.

|  |  |
| --- | --- |
|  | if (x == 0) y = 0;  f(a, b[i]); |

Every line must fit on 80 columns. If you must break a statement, add an indentation level for the continuation:

|  |  |
| --- | --- |
|  | a[n] = ..................................................  + .................; |

Start the indented line with an operator (if possible).

  If the condition in an if or while statement must be broken, be sure to brace the body in, *even if it consists of only one statement:*

|  |  |
| --- | --- |
|  | if ( .........................................................  && ..................  || .......... )  {  . . .  } |

If it weren't for the braces, it would be hard to separate the continuation of the condition visually from the statement to be executed.

**A1.7.3—Braces**

Opening and closing braces must line up, either horizontally or vertically:

|  |  |
| --- | --- |
|  | while (i < n) { System.out.println(a[i]); i++; }  while (i < n)  {  System.out.println(a[i]);  i++;  } |

Some programmers don't line up vertical braces but place the { behind the key word:

|  |  |
| --- | --- |
|  | while (i < n) { // DON'T  System.out.println(a[i]);  i++;  } |

Doing so makes it hard to check that the braces match.

**A1.7.4—Unstable Layout**

Some programmers take great pride in lining up certain columns in their code:

|  |  |
| --- | --- |
|  | firstRecord = other.firstRecord;  lastRecord = other.lastRecord;  cutoff = other.cutoff; |

This is undeniably neat, but the layout is not *stable* under change. A new variable name that is longer than the preallotted number of columns requires that you move *all* entries around:

|  |  |
| --- | --- |
|  | firstRecord = other.firstRecord;  lastRecord = other.lastRecord;  cutoff = other.cutoff;  marginalFudgeFactor = other.marginalFudgeFactor; |

This is just the kind of trap that makes you decide to use a short variable name like mff instead.

Do not use // comments for comments that extend for more than two lines. You don't want to have to move the // around when you edit the comment.

|  |  |
| --- | --- |
|  | // comment — don't do this  // more comment  // more comment |

Use /\* ... \*/ comments instead. When using /\* ... \*/ comments, don't "beautify" them with additional asterisks:

|  |  |
| --- | --- |
|  | /\* comment—don't do this  \* more comment  \* more comment  \*/ |

It looks neat, but it is a major disincentive to update the comment. Some people have text editors that lay out comments. But even if you do, you don't know whether the next person who maintains your code has such an editor.

Instead, format long comments like this:

|  |  |
| --- | --- |
|  | /\*  comment  more comment  more comment  \*/ |

or this:

|  |  |
| --- | --- |
|  | /\*  comment  more comment  more comment  \*/ |

These comments are easier to maintain as your program changes. If you have to choose between pretty but unmaintained comments and ugly comments that are up to date, truth wins over beauty.

Java Coding Guidelines

https://www.securecoding.cert.org/confluence/display/java/Java+Coding+Guidelines

**Naming Convention:**

**What Is a Naming Convention?**

A naming convention is a rule to follow as you decide what to name your identifiers (e.g. class, package, variable, method, etc..).

**Why Use Naming Conventions?**

Different Java programmers can have different styles and approaches to the way they program. By using standard Java naming conventions they make their code easier to read for themselves and for other programmers. Readability of Java code is important because it means less time is spent trying to figure out what the code does, leaving more time to fix or modify it.

To illustrate the point it's worth mentioning that most software companies will have a document that outlines the naming conventions they want their programmers to follow. A new programmer who becomes familiar with those rules will be able to understand code written by a programmer who might have left the company many years before hand.

**Picking a Name for Your Identifier**

When choosing a name for an identifier make sure it's meaningful. For instance, if your program deals with customer accounts then choose names that make sense to dealing with customers and their accounts (e.g., customerName, accountDetails). Don't worry about the length of the name. A longer name that sums up the identifier perfectly is preferable to a shorter name that might be quick to type but ambiguous.

**A Few Words About Cases**

Using the right letter case is the key to following a naming convention:

* **Lowercase** is where all the letters in a word are written without any capitalization (e.g., while, if, mypackage).
* **Uppercase** is where all the letters in a word are written in capitals. When there are more than two words in the name use underscores to separate them (e.g., MAX\_HOURS, FIRST\_DAY\_OF\_WEEK).
* **CamelCase** (also known as Upper CamelCase) is where each new word begins with a capital letter (e.g., CamelCase, CustomerAccount, PlayingCard).
* **Mixed case** (also known as Lower CamelCase) is the same as CamelCase except the first letter of the name is in lowercase (e.g., hasChildren, customerFirstName, customerLastName).

**Standard Java Naming Conventions**

The below list outlines the standard Java naming conventions for each identifier type:

* **Projects/Packages:** Names should be in lowercase. With small projects that only have a few packages it's okay to just give them simple (but meaningful!) names:
* package pokeranalyzer

package mycalculator

In software companies and large projects where the packages might be imported into other classes, the names will normally be subdivided. Typically this will start with the company domain before being split into layers or features:

package com.mycompany.utilities

package org.bobscompany.application.userinterface

* **Classes:** Names should be in CamelCase. Try to use nouns because a class is normally representing something in the real world:
* class Customer

class Account

* **Interfaces:** Names should be in CamelCase. They tend to have a name that describes an operation that a class can do:
* interface Comparable

interface Enumerable

Note that some programmers like to distinguish interfaces by beginning the name with an "I":

interface IComparable

interface IEnumerable

* **Methods:** Names should be in mixed case. Use verbs to describe what the method does:
* void calculateTax()

string getSurname()

* **Variables:** Names should be in mixed case. The names should represent what the value of the variable represents:
* string firstName

int orderNumber

Only use very short names when the variables are short lived, such as in for loops:

for (int i=0; i<20;i++)

{

//i only lives in here

}

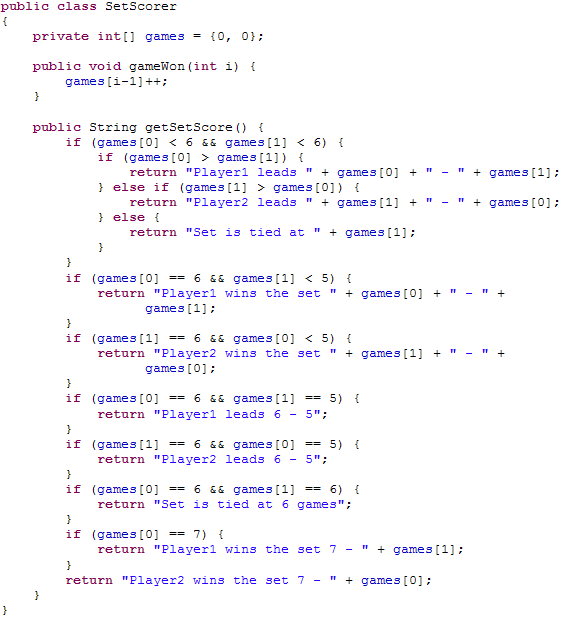
* **Constants:** Names should be in uppercase.
* static final int DEFAULT\_WIDTH

static final int MAX\_HEIGHT

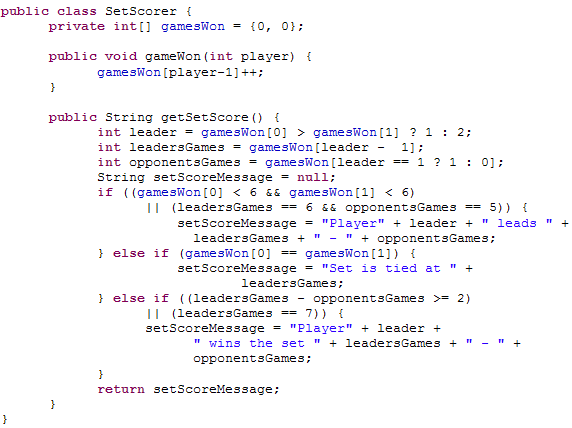
**Elegant Code (Extract from IBM)**

There are many things that contribute to clean code. Some are universal and apply to any type of programming language or problem domain for which you are developing software. Some of the properties of clean code depend upon the specific programming language.

Let's take a quick look at two short pieces of code that do the same thing. This is code that scores sets in tennis. Each one has a method to record a game won by a player and another method to return a message containing the score. Both assume that all inputs are valid and no checks for invalid state are made. They are written in a design-by-contract method, where the burden on the correct use of the class is on the client. They both pass the same set of unit tests. Now, take a look at the two listings below. Which do you think is "cleaner," Listing 1 or Listing 2?



**Listing 1: First tennis set scoring example**



**Listing 2: Second tennis set scoring example**

Listing 1 is written in a naive style that a beginning programmer might use with code that seems to repeat itself. It's not necessarily complex, but seems cluttered to me and quite a bit inefficient. Listing 2 has more complex conditions, but if you understand Java, it reads quite well. The only part that you might question is the first part of the condition in the last *else if*. It turns out that when you get to this clause, one player has won.

Neither implementation is wrong. In fact, they are both small enough that they're not much more than a example, so talking about how clean the code isn't a very useful discussion to have about these listings, except to whet your appetite about what makes one implementation cleaner than another.

Source : http://www.ibm.com/developerworks/rational/library/nov06/pollice/