Comments

Block Comments

Block comments will have the following structure, with a leading asterisk, or a leading space followed by an asterisk, on each line

```
/*
* Block comment text starts here…
* and more here…
*/
```

or

```
/*
 * Block comment text starts here…
 * and more here…
 */
```

Header Comment

All files will contain a file header block comment that contains, at a minimum:

Name of the author
Date of Creation
Functional Description of the Module

In-line Comments

In-line comments that run on multiple lines should line up, for example

```
…
for (i = 0; i < 42; ++i)       // Iterate over the meaning of existence
{
   Count += bobsYourUncle();   // Count number of uncles named Bob
   j *= judysYourAunte(Count); // Not sure what Judy does with this…
}
```

Functions, Methods and Classes

All functions, methods and classes will have a block comment that describes the operation and limitations of the function. Special conditions, such as argument value requirements, must be included in this comment.

Functions, Methods, Classes, if, do, while, switch, etc.

The closing curly-bracket in any code block shall have an in-line comment that repeats the reason for the block. This comment is required if the block is longer than ten (1) lines, otherwise it is optional. Example:

```
while (itemCount > 0)
{
… A fairly long block of code is in here
…
…
```

```
…
…
…
…
… and why are we in here in the first place?
… Oh yes, look at the next line!
} // while (itemCount > 0)
```

Formatting

Line Length

The code should be displayable and not result in word-wrapping problems! Limiting the line length to 132 characters is generally a good idea.

Naming Conventions

Functions, methods and variables

Functions, methods and variables shall start with a lower case letter and can then be a mixture of lower and upper case letters. The use of the underscore character is to be avoided, except in those rare cases where it increases readability. The variable names i, j, and k should only be used as loop index counters. Names should be meaningful and should indicate the purpose of the function, variable or class. Use of the letters l (lowercase L) and O (uppercase o) should be avoided because they look too much like one and zero. Exceptions are, of course, when they are used in a word-like name such as `loopCounter` or `pullFromStack()`. Examples:

```
int iVal, myInteger, yetAnotherVariableName, n;
void myFunctionThatisforIllistrationPurposesOnly(void);
```

Constants

Constants shall be in upper case letters. Constants are any value that is set once and then never changes. The underscore character should be used to improve readability. Examples:

Java
```
static final int SEC_PER_DAY = 3600;
```

C and C++
```
#define SEC_PER_DAY ((int)3600)
```

Magic Numbers

Special values (aka *Magic Numbers*) shall not be used in the code base unless they are defined as constants and are clearly documented. Magic numbers in your code will cost you your A+.

Classes

Classes shall start with an upper case letter and can then be a mixture of lower and upper case letters. The use of the underscore character is to be avoided, except in those rare cases where it increases readability. Example:

```
public class Bicycle
{
   … All the stuff you find in a class
} // public class Bicycle
```

Indenting and alignment

Tabs shall not be used for indenting in order to facilitate portability. The standard indent shall be three (3) spaces. Tabs in your code will cost you your A+, or worse if they are pervasive.

The curly-brackets shall appear below function and method names as shown below:

```
void myFunctionThatDoesNothing()
{
   return;
}
```

The same convention shall be used for `while, for`, `if` and `case` constructs. For example

```
if (a == 0)
{
   // do some stuff in here…
}
```

White Space

White space is to be used to increase readability. Spaces should be used in conditional and loop statements to improved readability, Blank lines should precede and follow in-line declarations and the first line of executable code after the initial block of declarations. It should also be used to distinguish between different functional blocks (initializations, declarations, loops, etc...). For example:

```
static final int CENTURY = 100;
int i, j, k;
double x, y, z;

x = 0.;

int[] a = new int[CENTURY];   // A 100 year array

for (i = 0; i < CENTURY; ++i) // Fill the array with pointless stuff
{
   a[i] = i;
}
```

is much more readable than

```
static final int CENTURY=100;
int i,j,k;
double x,y,z;
x=0.;
int[] a=new int[CENTURY]; // A 100 year array
for(i=0;i<CENTURY;++i)    // Fill the array with pointless stuff
{
   a[i]=i;
}
```

Mixed Mode Arithmetic

Never mix arithmetic types when doing a calculation. The number 1 is some kind of integer. The number 1.0 is some kind of floating point value. Consider the following calculations: "`A = 1/2 * B`" and "`A = 1.0/2.0 * B`". What will be the results? The outcome depends on the types of `A` and `B` and how the compiler handles arithmetic precedence as well as automatic casting (never assume that the compiler or interpreter will follow the rules you learned in elementary school). The first example will likely be zero in many cases, even if the variable `B` is not an integer! Unless you wrote the compiler, you really don't know.

Flow Control

The `break` statement shall ONLY be used within a `switch/case` construct. It shall NEVER be used to exit a loop construct. The `continue` statement can be used if it improves readability such as when the entire body of the loop would be enclosed in an if-statement.

FOR and WHILE Loops: The iterator (the loop counter) in a for-loop shall not be modified within the body of the loop. Additionally, a for-loop shall not be used to simulate a while-construct. The for-loop construct shall always execute a predetermined fixed number of iterations.

All functions and methods shall have a `return` statement as their last executable line. There is an exception for void methods in Java, where the statement is optional. In general, there shall be one AND ONLY ONE return statement in a function or method. Note that exceptions can be made if the code is easier to understand if it uses multiple return statements. Exceptions can be made for very small functions, but only if clarity is not sacrificed. For example:

```
/*
* sign function:
* returns -1 if x < 0,  0 if x = 0, 1 if x > 0
*/
int sign(int x)
{
   if (x < 0) return(-1); else if (x > 0) return (1); else return(0);
}
```

is at least as readable as…

```
int sign(int x)
{
   int iRet = 0; // return value default is zero

   if (x < 0)
      iRet = -1;
   else if (x > 0)
      iRet = 1;

   return(iRet);
}
```

Note that the following “clever” code using the ternary operator is NOT as readable as the above code listings:

```
int sign(int x)
{
   (x < 0) ? return(-1) : (x > 0) ? return(1) : return(0);
}
```