Java Style Guidelines

Coding style is important because it increases readability and maintainability of your code. Most companies will have some level of style guidelines for their employees, and INFDEV01-3 follow this practice by imposing style guidelines for students. By following the style guidelines, you will enable the instructors to help you solve coding problems more quickly since they will not waste time figuring out the style of your code.

1. Formatting Code

These guidelines are for improving the readability of your code. They involve issues of indenting and spacing as well as how to choose good names for your variables and constants. While you do have limited choices, do not violate these guidelines unless it cannot be helped. Good formatting not only makes your code more readable, it is also beneficial during debugging. Finding missing or misplaced braces and parentheses is easier if your code is well formatted. You are advised to get into the habit of writing well formatted code early on so that it becomes second nature.

1.1. Indent nested code

The best way to make your code readable is to indent nested code. Nested code means code that is "inside" other code. One easy heuristic is to indent every time you go inside braces. Some code, such as if statements and for loops, use braces optionally. Even if you do not use the braces for these statement, consider the braces invisible and indent according to this rule anyway. You must indent using a certain number of spaces; we recommend 2, 3, or 4 spaces. The number of spaces for indentation must be consistent throughout your code. The length of tabs can vary depending on your text editor, which is why you must use spaces instead of tabs for indentation.

```
public class Example {
...void someFunction (int arg) {
......if (arg > 10)
.........System.out.println("big");
......for (int i=0; i < arg; i++) {
.........System.out.println(i);
......}
...}
}
```

Indent after labels in switch statements even though the code after the label is not technically a separate block. This makes the labels easier to distinguish.

```
void someFunction (int arg) {
...switch (arg) {
......case 1:
.........System.out.println("one");
.........break;
......case 2:
.........System.out.println("two");
...}
}
```

1.2. Use braces

Many syntactic structures such as class definitions and methods require braces. However other structures, such as if statements and for loops, do not require braces if the inner code is just one line long. It is best to use braces even when they are not required. This makes it easier to identify what the body of the statement is and also makes it easier to add more lines of code to the inner block later on. Using braces at all times is an optional style consideration.

```
if (classSize > 140)
   return false;     // Braces unnecessary and not used.

if (classSize > 140) {
   return false;     // Braces unnecessary but used anyway.
}
if (classSize > 140) {
   System.out.println("Too many students!");
   return false;       // Braces necessary.
}

if (classSize > 140)
   System.out.println("Too many students!");
   return false;       // Logic error. Return is not in the if body.
```

1.3. Locate braces appropriately

Locate the opening brace '{' of each block statement either as the last character on the statement line or on the next line by itself. If you place the brace on the next line, do not begin the inside code on the same line. In either situation, locate the closing brace '}' on a line by itself. Indent the closing brace to align with the first character of the block statement.

`public class Example {`
`  …`
`}`
	`public class Example`
	`{`
	`  …`
	`}`

`void someFunction (int arg) {`
`  …`
`}`
	`void someFunction (int arg)`
	`{`
	`  …`
	`}`

`for (int i = 0; i < 10; i++) {`
`  …`
`}`
	`for (int i = 0; i < 10; i++)`
	`{`
	`  …`
	`}`

`switch (arg) {`
`  case 1:`
`  …`
`  case 2:`
`  …`
	`switch (arg)`
	`{`
	`  case 1:`
	`  …`
	`  case 2:`

} 	… }

1.4. Use white space

White space consists of characters such as space and tab and newline that are invisible in the text editor but take up space nonetheless. You should embed white space between operators and operands in arithmetic and logic expressions to increase the readability of your code.

```
// Bad style:
double d=(-b-Math.sqrt(b*b-4*a*c))/(2*a);

// Better style (added spaces):
double d = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);

// Bad style:
if (x>LEFT&&x<RIGHT&&y>TOP&&y<BOTTOM)
   return true;

// Better style (added spaces):
if (x > LEFT && x < RIGHT && y > TOP && y < BOTTOM)
   return true;
```

1.5. Break up long lines

If you are writing a statement that is too long, the editor might cause the line to wrap around to the next. Some text editors allow lines to be indefinitely long, so you are able to see the entire line only by scrolling horizontally. However, if the line is longer than 100 characters, a printer can either force the line to wrap or cut off the end of the line. Keep your lines under 100 characters in length.

Some lines of code can be broken up into several individual statements.

```
   double length = Math.sqrt(Math.pow(Math.random(), 2.0) + Math.pow( Math.random(), 2.0));
// Too long!

   // Better style:
   double xSquared = Math.pow(Math.random(), 2.0);
   double ySquared = Math.pow(Math.random(), 2.0);
   double length = Math.sqrt(xSquared + ySquared);
```

Another strategy is to break the long line in strategic positions. If a method call is long and has several parameters, you can break the line at the commas between parameters. You can break relational statements such as if statements with many conditions just after the `||` and `&&` operators.

```
    double finalSolution = functionThatComputesTheAnswer (argument1, argument2, argument3);
// Too long!

    // Better styles:
    double finalSolution = functionThatComputesTheAnswer (argument1,
                                                          argument2,
                                                          argument3);

    double finalSolution = functionThatComputesTheAnswer (argument1, argument2,
                                                          argument3);

    // Bad style
    if (positionX >= LEFT_BOUNDARY && positionX <= RIGHT_BOUNDARY && positionY >=
TOP_BOUNDARY && positionY <= BOTTOM_BOUNDARY) {   // Too long!
        return true;
    }

    // Better style:
    if (positionX >= LEFT_BOUNDARY &&
        positionX <= RIGHT_BOUNDARY &&
        positionY >= TOP_BOUNDARY &&
        positionY <= BOTTOM_BOUNDARY) {
       return true;
    }
```

1.6. Use meaningful names

When declaring variables and constants, use names that accurately describe what the data is going to be used for. Class names should describe what the instances of the class will do or represent. Method names should describe what the methods do. Do not let a name get too long a meaningful name that is too long is hard to read.

Variables and Methods: Use multiple words to describe the meanings of your variables and methods if a single word would be ambiguous. Begin variable and method names with lowercase letters. If you use more than one word in a name, capitalize the first letter of each word except the first word. Do not use underscores '_'.

```
    int heightOfSearsTower;
    int heightOfSpaceNeedle;

    void computeVelocityAtImpact (double height) {
        …
    }
```

One exception to the meaningful names rule is for variables used to iterate through loops. In the following example, the variable i is created only for iterating though the loop.

```
for (int i=0; i < 10; i++);
```

Avoid abbreviations in your variable and method names. The exception to this rule is for common words such as “number” and “seconds”, which could be abbreviated as “num” and “sec”. If you abbreviate common words, make sure you abbreviate consistently.

Constants: Constant names should be all uppercase, with words separated by underscores.

```
final int NUM_LINES = 5;
```

Classes: Capitalize the first letter of class names.

```
public class Car {
   …
}

public class RaceCar extends Car {
   …
}

public class Formula1RaceCar extends RaceCar {
   …
}
```

1.7. Avoid magic numbers

According to the textbook (S. Reges and M. Stepp, Building Java Programs: A Back to Basics Approach, 3rd Edition), we should use class constants to make our "programs more readable and adaptable" and avoid the use of magic numbers, which are hard coded literals. However, it is fine to use the numbers 0, 1, 2, and 100 in your programs when they are used for counting (such as loop indexes) or mathematical operations (calculating a percentage, dividing a quantity in half, determining whether a number is even or odd). It is also OK to use numbers in some formulas, such as the formula for the volume of a sphere, $V = 4/3\pi r3$. Otherwise you should define class constants rather than use magic numbers in your programs. There are three situations when you should use a named, class constant rather than just typing a value into the code that uses it.

1. If the value requires some explanation, a magic number may hide the explanation that a named constant could provide. For example, code like a = x * 1.579786; might leave the reader confused, but a = x * HALF_PI; probably makes a lot more sense. Here, 1.579786 would be a magic number.

2. If the value represents an adjustable parameter, a named constant provides a standard way to advertise the parameter to other programmers. For example, code like score += 150; might hide a modifiable feature of the program, while code like score += ALIEN_SHIP_SCORE; would do a better job of highlighting it.
3. If the value occurs in more than one place in our program and every occurrence has to agree, a named constant can force them to agree. For example, in the following program fragment, the named constant OUTPUT_WIDTH makes it clear that these two loops are supposed to run the same number of iterations. If you were to use a literal value like 18 in both of these loops, then 18 would be considered a magic number.

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

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

It is fine to use numeric values when the value requires no additional comments and could not realistically be a tunable parameter. For example, the constants 1, 2, 4.0 and 3.0 are all appropriate in the following code fragments.

```
// Visit every odd-numbered column

for ( int i = 1; i <= width; i += 2 ) {
   ;
}

// Compute volume of the sphere.
double volume = 4.0 / 3.0 * Math.PI * Math.pow( r, 3.0 );
```

2. Documentation and Commenting

Documentation and comments should explain (to yourself and others) exactly what your code is trying to accomplish and how you are approaching the problem. Writing comments before you write code often helps you organize your thoughts and make programming easier. Mostly, documentation is for the benefit of others who will read your code after it has been completed.

You can document during programming or after programming. However, you are advised to document your code as you write it – after writing a large program, it is often the case that you have forgotten what your earlier code does and how it works.

2.1. Understand comment types

Java supports three different types of comments: C-style comments (first introduced in the programming language C), in-line comments (first introduced in the programming language C++), and Javadoc (see S. Reges and M. Stepp, Building Java Programs: A Back to Basics Approach, 3rd Edition, Appendix B).

```
/* This is a single line C-style comment. */

/*
 * This is a multi-line
 * C-style comment.
 */

// This is a single line in-line comment.

// This is a multi-line
// in-line comment.

/**
 * This is a JavaDoc comment
 * @author Jane Doe
 */
```

Document each instance variable, constant, method, and class with Javadoc style comments. Use either C or in-line style comments inside method bodies to document what you intend the code to do when executed.

2.2. Document class headers

For each class in your code, including test classes, write javadoc comments that describe what the class does and what it will be used for plus the author's name. These javadoc comments must be directly above the class header. Any imports will be above the class documentation and header.

```
/**
 * The Circle class maintains information about the size, shape,
 * and area of a circle.
 * @author Alex Famous
 */
 public class Circle {
   …
}
```

2.3. Document method headers

For each method, write javadoc comments that describe the parameters that will be passed in, what the method will do with that data, and what results will be returned along with information about any exceptions thrown. These javadoc comments go immediately above the method header.

```
/**
 * Set the radius of the circle.
 * @param radius new radius of the circle
 */
public void setRadius (double radius) {
    …
}


/**
 * Get the radius of the circle.
 * @return radius of the circle
 */
public double getRadius () {
   …
}

/**
 *  Computes and returns the distance between this point
 *  and the given other point.
 *
 *  @param p the point to which the distance is computed
 *  @return the distance, computed as the square root of
 *          the sums of the squares of the differences
 *          between the two points' x-coordinates (dx)
```

```
     *          and between their y-coordinates (dy)
     *  @throws NullPointerException if p is null
     */
    public double distance(Point p) {
        int dx = x - p.x;
        int dy = y - p.y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    /**
     * Starts the program.
     * @param args command line arguments
     */
    public static void main(String[] args) {
        …
    }
```

Note that the @param tag is used only for method parameters. Do not include @param tags for variables used in a method.

2.4. Document variables

For each variable in your program, write a javadoc comment immediately above the variable declaration that describes what the variable is used for. Do this even when the name of the variable makes its use obvious.

```
/**
 * Radius of the circle
 */
private int radius;
```

OR

```
/** Radius of the circle */
private int radius;
```

2.5. Document constants

For each constant that is declared in your program, write a javadoc comment immediately above the constant declaration that describes what the constant is used for. Do this even when the name of the constant is obvious.

```
/** Constant representing number of spaces to indent */
public static final int NUM_SPACES = 4;
```

2.6. Provide internal comments for long methods

It is sometimes useful to break up the code of long methods into chunks that serve a similar purpose and then comment that purpose. The internal documentation should precede the grouping of statements and should describe what they will accomplish. Separate the groups by empty lines to improve readability.

```
public void HelloWorldGUI () {    // Initialize the window.
    super("Example");
    setLocation(X, Y);
    setSize(WIDTH, HEIGHT);

    // Initialize the label.
    lblHello = new JLabel("Hello, World!");
    lblHello.setFont(new Font("Helvetica", Font.BOLD, FONT_SIZE));

    // Install the label in the window.
    Container c = getContentPane();
    c.setBackground(Color.white);
    c.setForeground(Color.black);
    c.setLayout(new FlowLayout());
    c.add(lblHello);

    // Make sure the window is visible.
    setVisible(true);
}
```

2.7. Write end comments for highly nested code

If your code is highly nested, it may not be enough to merely line up the closing braces with the originating statement. This is especially true when blocks of nested code are longer than what you can see on the screen. In this case, even though the closing braces are indented properly, it may still be hard to tell which brace closes which block of code. When this happens, comment the closing brace to indicate which statement with which it is paired.

```
for (i=0; i < 10; i++) {
   for (j = 0; j < i; j++) {
      …
      if (j + i > SENTINEL) {
         …
         switch (j + i) {
            …
         } // switch
      } // if
    } // for j
} // for i
```