# Coding Conventions and Styles

## Introduction

Employing good and consistent coding conventions is very important. It helps make the programs readable and improves productivity (not to mention the fact that it is very helpful for your TAs). For these reasons, it is in your best interest that all assignments meet the standards described below. Following the proper style and conventions listed in this document will help you immensely. Don't worry if you are not sure what the terms below mean; you will soon get to know them thoroughly. After reading this document, make sure you keep it by your side when you code to answer your questions about format and style.

## Names

Using the following naming conventions will keep code ambiguities to a minimum.

### Class Names: `SlinkyDog`

Classes should be named using nouns which describe what the class models. Class names should always begin with a capital letter. All new words within the name should start with a capital letter.

### Method Names: `fly()` or `goToTheStore()`

These usually perform some kind of action and/or return a value, so we use verbs to indicate what they do. Method names should always begin with a lowercase letter, and all new words within the name should start with a capital letter.

### Accessor Methods: `getToyOwnerName()`

Methods that retrieve information from an instance variable are often called accessor methods. Accessors that retrieve the value of an instance are named after the attribute they are going to retrieve and prefixed with the word "get".

### Mutator Methods: `setToyOwnerName(String ownerName)`

A method whose purpose is to change the value of some attribute is named after the attribute it is going to change and is prefixed with the word "set". These methods are sometimes called mutators. Most of the time mutators take the new value of the variable they are changing as a parameter.

### Instance and Local Variables: `_numGreenArmyMen` and `numToys`

Instance and local variables store state information about the object. Their names should be comprised of nouns describing the information they store. A variable name such as `numSodas` is better than `stuff`. All instance and local variable names begin with a lowercase letter and each new word begins with a capital letter. . This allows you to easily differentiate between instance and local variables, which is very handy. If you have one local variable of a certain type in a method, it is acceptable to name it the same as its class name, e.g. `Button button = new Button()`. However, if you have more than one variable of a type, it is preferable to be as descriptive as possible, e.g. `quitButton, resetButton,` as opposed to `button1, button2`.

Note: You should generally avoid using one letter variable names, as they are not descriptive and will make your code more difficult to read. The only exceptions are if the usage agrees with an established naming convention, such as the use of `i` or `j` for a simple loop index, or `x` and `y` for coordinates are acceptable. Don't

feel constrained by traditional naming conventions, though. You can always make up some more descriptive names for loop variables, especially with multiple and nested loops. The code below is a good example of this.

```
int tableSum = 0;
for (int row = 0; row < table.getNumRows(); row++) {
     for (int column = 0; column < table.getNumColumns(); column++) {
          tableSum += table.getValueAt(row, column);
     }
}
```

**Constants: `DORYS_IQ`**

Constants should have names consisting of nouns describing their content. In Java, constants are just instance variables of classes. You will learn how to make a constant in the first few weeks of class. In order to distinguish them from just plain instance variables, we use only capital letters in their names and separate different words with underscores. In addition, a constant declaration should be `static final`, e.g.

```
public static final int NUM_HTAS = 4
```

**Using Predefined Classes: `course.prj.LiteBrite.Palette`**

When you use predefined classes (like classes from course.prj) we prefer that you do not import the whole package, but instead import each class that you're using as necessary. This should minimize name clashes in your code. (If you don't know what this means, don't worry about it.) That is, instead of having a line like

```
import course.prj.LiteBrite.*;
```

You should import each class you're using like so:

```
import course.prj.LiteBrite.LiteBox;
import course.prj.LiteBrite.ColorPalette;
```

## Indentation, WhiteSpace, and Skipping Lines

### Indentation

Java consists of nested blocks of code. For example, if statements can be contained in for loops, which are contained in methods, which are contained in classes. One way to show this nesting is by indentation. This allows the person reading your code to easily see the major sections of code and main flow-of-control in methods. As you edit your program, however, you may need to re-indent your code. The following is the course's indentation conventions.

Nested code should be indented 4 spaces from the previous line.

```
Claw claw = new Claw();
for (int i = 0; i < _clawMachine.getNumAliens(); i++) {
     Alien currentAlien = claw.getAlien(i);
     if (currentAlien.hasBeenChosen()) {
          currentAlien.sayFarewell();
          currentAlien.goOffToABetterPlace();
```

```
    }
}
```

**Braces**

Braces indicate the beginning and end of blocks of code. Class definitions, methods, and loops all have braces around them. The opening brace should be on the same line as the beginning of the block; i.e., if the braces are around the class, the opening brace is on the same line as the name of the class; if the braces are around the method, the opening brace is on the same line as the name of the method, etc. The closing brace should be aligned with the beginning of the line with the opening brace. Make sure to leave a space before the opening brace, so instead of

```
public void buildTreeHouse(){ ...,
```
write

```
public void buildTreeHouse() { ...
```

The below is a good example of how to use braces.

```
public class Alien {
     private Eye _leftEye;
     private Eye _centerEye;
     private Eye _rightEye;
     public void blink() {
          _leftEye.blink();
          _centerEye.blink();
          _rightEye.blink();
     }
}
```

**Line Limits**

Lines of code should not exceed 80 characters, as anything longer than that becomes less and less readable. Such lines should be broken up in what's called "line breaking." Below are a couple special cases for how to wrap long lines. The one exception to this rule is import statements.

### Operator Wrapping

This

```
int x = getSomeNumber() + getSomeNumber() + getSomeNumber() + getSomeNumber() +
getSomeNumber();
```

should be changed to

```
int x = getSomeNumber()
+ getSomeNumber()
+ getSomeNumber()
+ getSomeNumber()
+ getSomeNumber();
```

### Skipping Lines

Another way of making major sections of your code easy to read is by skipping lines. You should skip one line between methods and other sections within a class. You should also separate logical chunks of code within a method with one or two blank lines.

## Code Organization

### Classes

Each class should live in its own file with a matching file name, e.g. “Fish.java” for a class named “Fish”.

### Private Classes

Private classes should be declared at the very bottom of their containing classes.

```
public class BigClass {
     //methods, members elided

     private class SmallClass {
          //methods, members elided
     }
}
```

Note that the brace closing SmallClass immediately precedes the brace closing BigClass.

## Internal Documentation

“Internal documentation” simply refers to the comments in your program. No matter how simple a program is to its author, it is almost always confusing to another person reading it (i.e, the TAs who will be grading your programs). Comment well, and you will have a happy TA. Comment poorly, and the TA will have trouble understanding what you are doing (and you may not get the benefit of the doubt). There are two ways to make a comment in Java. Comments that only span one line are indicated by a double slash ( //).

```
// This is a one line comment.
```

This style of comments is useful when you want to add a comment to a line that already has code on it. Typically, a comment would be on the line directly above the line to which it applies. Example:

```
// Check to see if the toy is owned by Andy
if (owner.equals("Andy")) {
```

Comments that span several lines are enclosed by /* */.

```
/*
* Comments that are several lines long are called 'block comments', and you
* should make sure to leave at least one empty line before block comments. Such
* comments are commonly used to describe large chunks of code like the purpose
* of a large loop or some code that isn't immediately obvious and takes a
* couple lines to explain.
*/
```

Note that the opening '/*' and closing '*/' should be on their own lines.

Comments are used for adding information to your code. Your comments should not just repeat the code that you have written, as such comments are useless. Comments should be added whenever an explanation or clarification is needed for some section or line of code, such as a difficult sequence of commands, an unusual or atypical design pattern, or a very brief synopsis of what a particular method call might do or return. Comment as you go along, do not leave all your commenting until the end of the project. Commenting done later takes more time because you have to remember or figure out what the code is doing instead of just writing down what you are already thinking about.

**Header Comments**

Header comments describe methods and classes. They consist of a block of information placed before the code they explain. Use multiline commenting style with an extra asterisk ( /** */) for header comments. You should describe each class and its design in the header comment.

Every method should have a header which contains a comment describing its purpose, a high level description of how it achieves its purpose, including the arguments it takes in and why. If the method returns a value, the importance of this value should be described in the header as well. Bad Example of a Method Header:

```
/**
* This method adds a graphic.
*/
public boolean addGraphic(ShapeInterface s) { ... }
```

Good Example of a Method Header:

```
/**
* This method adds an object which implements the ShapeInterface
* to the GraphicsCollection. This will cause this shape to be
* displayed the next time the graphical frame is rendered.
* If the object is already in the GraphicsCollection, the item is not
* added, and the method returns false. Otherwise, returns true.
*/
public boolean addGraphic(ShapeInterface s) { ... }
```

**Inline Comments**

Inline comments appear in code to clarify its workings. Use // commenting style for inline comments. Lines should be commented to describe what they are doing. Note that this comment should not just repeat what the Java code does, but should elaborate on the code. This should be done for every section and logical chunk of code.

Here is an example of a good use of inline comments:

```
// iterate through first half of the list, individually swap with other half
for(int i = 0; i < list.length / 2; i++) {
      String temp = list[i];
      // swap with the corresponding element on the other half
      list[i] = list[list.length - i - 1];
      list[list.length - i - 1] = temp;
}
```

Here is an example of an excess of inline comments:

```
// iterate through first half of the list, individually swap with other half
for(int i = 0; i < list.length / 2; i++) { // a for-loop
      // store original element
      String temp = list[i];
      // swap with the corresponding element on the other half
      list[i] = list[list.length - i - 1];
      // put the overwritten element back into the list
      list[list.length - i - 1] = temp;
}
```

Note: Inline comments can be helpful while you are coding so that you can better understand what you are writing while you are coding. Feel free to use as many comments as you like for that purpose, just be sure to slim down those comments to what is necessary when you hand-in your assignment.

**Misc. Conventions**

Using ‘This’

Please use the ‘this’ keyword when calling methods in a class that are defined elsewhere in the same class! For example, instead of

```
class Horse {
      public void move() {
            gallop();
      }
}
```

Write

```
class Horse {
      public void move() {
            this.gallop();
      }
}
```