CSE142

Style and Commenting Guide

Michelle Yun

Last updated: March 28, 2017

Contents

DISCLAIMER: This guide is not all-inclusive and does not substitute for reading through the feedback that your TA leaves on your homework assignments.

Suggested Reading

IMPORTANT: This document covers the entire quarter, so it is not meant to be something that you read through in one sitting. It is a reference guide that you will hopefully be able to go back to as new topics are addressed. I have tried to make notes of approximately when certain style issues become relevant next to section headings, but this may vary depending on the quarter/professor. Here is a (linked!) table of suggested readings that you can go through before you turn in each homework assignment. Note that these suggested readings stack on top of each other – ie: assignment 4's suggested readings include assignment 3's, which include assignment 2's, etc.

assignment #	suggested readings
1	indentation naming conventions (ignore variables and class constants) required comments: class comment (template) required comments: method comments (ignore parameters and returns)
2	loop zen: getting the most out of your loops naming conventions printing variable names: loop variable names spacing: expressions
3	long lines required comments: method comments (ignore returns)
4	conditional execution main methods objects: object creation scoping types: `int` vs `double` variable names required comments: method comments (template) comments to avoid: implementation details
5	boolean zen loop zen: only repeated tasks types: `int` vs `boolean` required comments: inline comments
6	loop zen: choosing the right loop
7	arrays
8	objects: fields

Style

2.1 arrays (∼ assignment 7)

2.1.1 data that goes together

Arrays should only be used to store logically cohesive information. They should not be used as "junk drawers" that cram a bunch of unrelated information together, and they should not be used as a way to get around not being able to return more than one value from a method. For example, say that you find yourself needing to return two different Strings from the same method. It's bad style to make and return a String array of length 2 that has the first String in the first index and the second String in the second index to get around the fact that you cannot return both (if you ever find yourself needing to return two different things from the same method, this is likely an issue with your method decomposition).

Take a look at the example below. In the `collectData` method, it doesn't really make sense to put temperature and rain values together in the same array because these values are logically disparate.

```java
public static void main(String[] args) {
   Scanner console = new Scanner(System.in);
   int[] weatherData = collectData(console);
   ...
}

public static int[] collectData(Scanner console) {
   int[] results = new int[11];
   System.out.println("enter your city's temperatures for the last 10 days.");
   for (int i = 1; i <= 10; i++) {
      System.out.print("day " + i + "'s temperature? ");
      int nextTemp = console.nextInt();
      results[i - 1] = nextTemp; // subtract 1 since we started at i = 1
   }
   System.out.print("how many days did it rain last month? ");
   int daysOfRain = console.nextInt();
   results[results.length - 1] = daysOfRain; // last index = results.length - 1
   return results;
}
```

Every number being stored in the `results` array is an `int`, but the data is conceptually split into two chunks: the first ten indexes hold temperatures, and the last index holds the number of days of rain in the last month. You could argue that all of the numbers have something to do with the weather, but this is still a pretty weak relationship. By cramming both types of data into the same data structure, we create additional work for ourselves. When we want to go through the temperatures, we have to remember to end our loop one index before the end of the array. When we want to access the days of rain, we have to remember exactly where it is in the array. We also have to document the fact that the last value in the array isn't a temperature to avoid confusing anyone else who might be reading our code. All of these issues indicate that the data should have just been kept separate in the first place.

```java
public static void main(String[] args) {
   Scanner console = new Scanner(System.in);
   int[] tempData = collectData(console);
   System.out.print("how many days did it rain last month? ");
   int daysOfRain = console.nextInt();
   ...
}

public static int[] collectData(Scanner console) {
   int[] results = new int[10];
   System.out.println("enter your city's temperatures for the last 10 days.");
   for (int i = 1; i <= 10; i++) {
      System.out.print("day " + i + "'s temperature? ");
      int nextTemp = console.nextInt();
      results[i - 1] = nextTemp; // subtract 1 since we started at i = 1
   }
   return results;
}
```

This fixed version has actually moved the code to prompt/store the rain data out of `collectData` and into main, where `daysOfRain` is now just being kept as a separate variable. It's important to realize that this is just one improved version of the original program. Depending on the exact structuring of the rest of this fictional program, other approaches could work, too.

2.1.2 unrolling

Whenever you are going through multiple values in an array, you should always use a loop to process that data rather than manually access each individual index (which we call "unrolling"). Here is what unrolling may look like for an array called `data` that has 3 values.

```java
int val1 = data[0];
int val2 = data[1];
int val3 = data[2];
int sum = val1 + val2 + val3;
```

Even when you know exactly how many values are in your array, using a loop is stylistically better than unrolling because loops are much more flexible than manually accessing each index. If the array being processed ever changed in length and you were unrolling, your code would no longer work (for example, what would happen if `data` had a length of 2 and the above code was run on it?). Using a loop allows your code to accomodate arrays of any length.

```java
int sum = 0;
for (int i = 0; i < data.length; i++) {
   sum += data[i];
}
```

2.2 boolean zen (∼ assignment 5)

boolean zen is all about using boolean values efficiently and concisely, and is best described through a few examples.

Let's say that you have a boolean value called test and that you want to execute a chunk of code only when test is true. Here is a tempting solution:

```java
if (test == true) {
   // do some stuff
}
```

However, note that test itself *is* a boolean value. When it is true, we are asking whether true == true. true == true will evaluate to true, but remember that the if branch will execute as long as what is in its parentheses evaluates to true. So we can actually use test directly:

```java
if (test) {
   // do some stuff
}
```

Maybe we instead want to execute the if branch when test is false. Here is an initial solution:

```java
if (test == false) {
   // do some stuff
}
```

But this looks very similar to the previous example's poor boolean zen, so we know we can probably do better. As stated before, the if branch will execute as long as what is in the if's parentheses evaluates to true. Before, we could just directly put test inside of the parentheses, but now we have to take a false value and turn it into a true value. We can do this by negating test with !.

```java
if (!test) {
   // do some stuff
}
```

Now let's say that, instead of executing a chunk of code when our boolean value test is true or false, we instead are at the end of some method and want to return true when test is true and false otherwise. Here's an initial attempt at a solution using what we just learned:

```java
if (test) {
   return true;
} else {
   return false;
}
```

There is actually a much more concise way to do this. If we want to return `true` when `test` is `true` and `false` when `test` is `false`, then we actually just want to return whatever `test` is.

```java
return test;
```

As a last example, consider how `boolean` zen can be used when assigning values to `boolean` variables. In the below code, say we want to set the variable `legalAdult` to `true` if the given age is greater than or equal to 18 and `false` otherwise. Here is an initial attempt:

```java
Scanner console = new Scanner(System.in);
System.out.print("how old are you? ");
int age = console.nextInt();
boolean legalAdult;
if (age >= 18) {
   legalAdult = true;
} else {
   legalAdult = false;
}
```

Using `boolean` zen, we can make the above code more concise. Note that we set `legalAdult` to `true` whenever the test `age >= 18` evaluates to `true`, and `false` whenever `age >= 18` evaluates to `false`. In other words, we can just set `legalAdult` equal to whatever the test `age >= 18` ends up evaluating to.

```java
Scanner console = new Scanner(System.in);
System.out.print("how old are you? ");
int age = console.nextInt();
boolean legalAdult = age >= 18;
```

2.3 conditional execution (∼ assignment 4)

2.3.1 choosing the appropriate combination

There are three different combinations of `if`s, `else if`s, and `else`s that we will be using throughout the quarter: the `if`/`if`/`if`, the `if`/`else if`/`else if`, and the `if`/`else if`/`else`. Each one has a different minimum number of branches that could execute and maxiumum number of branches that could execute.

combination	min	max
`if/if/if`	0	3
`if/else if/else if`	0	1
`if/else if/else`	1	1

When using conditional execution in your program, you should choose which combination would be best based on the minimum/maximum number of branches that could execute. For example, considering the minimum number of branches that could execute, we know that it would not be appropriate to use an `if/else if/else if` in the below code.

```java
Scanner console = new Scanner(System.in);
System.out.print("what is your favorite number? ");
int favNum = console.nextInt();
if (favNum < 4) {
   System.out.println("your favorite number is less than mine!");
} else if (favNum == 4) {
   System.out.println("you have the same favorite number as me!");
} else if (favNum > 4) {
   System.out.println("your favorite number is greater than mine!");
}
```

There are only 3 possibilities here, and exactly one of them must be true every time; if `favNum` is not less than 4 or equal to 4, then we automatically know that it must be greater than 4, so we can just end in an `else` branch.

```java
Scanner console = new Scanner(System.in);
System.out.print("what is your favorite number? ");
int favNum = console.nextInt();
if (favNum < 4) {
   System.out.println("your favorite number is less than mine!");
} else if (favNum == 4) {
   System.out.println("you have the same favorite number as me!");
} else { // favNum > 4
   System.out.println("your favorite number is greater than mine!");
}
```

It's worth noting that the code using the `if/else if/else if` and the code using the `if/else if/else` would work the same externally. However, using an `if/else if/else` is stylistically better because using an `if/else if/else if` implies (to human readers of our code) that `favNum` could fall into none of the branches, and we know that this is impossible.

2.3.2 `if/else` factoring

The whole point of conditional execution is to separate out different chunks of code. If you have common code being repeated between branches, then this is saying that you want to execute that code regardless of what you are testing for. Notice the redundant code in the below example.

```java
public static int getAge(Scanner console) {
   System.out.print("how old are you? ");
   int age = console.nextInt();
   if (age >= 18) {
      System.out.println("you are " + age + " years old");
      System.out.println("you are a legal adult!");
      return age;
   } else {
      System.out.println("you are " + age + " years old");
      System.out.println("you are not a legal adult yet.");
      return age;
   }
}
```

In both the `if` and `else` branches, we print out the user's age and return it. Since we want to do this in both cases, we should factor this code to above and below the `if/else`.

```java
public static int getAge(Scanner console) {
   System.out.print("how old are you? ");
   int age = console.nextInt();
   System.out.println("you are " + age + " years old");
   if (age >= 18) {
      System.out.println("you are a legal adult!");
   } else {
      System.out.println("you are not a legal adult yet.");
   }
   return age;
}
```

2.3.3 avoiding useless branches

Avoid `if`, `else if`, or `else` branches that are not necessary. Remember that you don't *always* need an `else if` or `else` branch to follow up an `if` branch – only use these if they make sense.

```
 1 Scanner console = new Scanner(System.in);
 2 int min = 10000; // let's assume we won't get a number > 10000
 3 for (int i = 0; i < 10; i++) {
 4    System.out.print("next number? ");
 5    int next = console.nextInt();
 6    if (next < min) {
 7       min = next;
 8    } else {
 9       min = min;
10    }
11 }
```

Notice how the `else` branch on line 8 is pretty useless because it's just setting `min` to the value that it already has. Sometimes students write even more extreme versions of this code where the unnecessary branch is completely empty.

```
 1 Scanner console = new Scanner(System.in);
 2 int min = 10000; // let's assume we won't get a number > 10000
 3 for (int i = 0; i < 10; i++) {
 4    System.out.print("next number? ");
 5    int next = console.nextInt();
 6    if (next < min) {
 7       min = next;
 8    } else {
 9
10    }
11 }
```

Instead, we should realize that the `else` branch isn't contributing anything to the program – so we can just leave it out entirely and keep the `if` branch by itself.

```
1 Scanner console = new Scanner(System.in);
2 int min = 10000; // let's assume we won't get a number > 10000
3 for (int i = 0; i < 10; i++) {
4    System.out.print("next number? ");
5    int next = console.nextInt();
6    if (next < min) {
7       min = next;
8    }
9 }
```

Here is an example where the fix is a little more complex.

```java
Scanner console = new Scanner(System.in);
System.out.print("how many dogs do you have? ");
int dogs = console.nextInt();
if (dogs < 3) {

} else {
   System.out.println("wow! you have a lot of dogs!");
}
```

When you have an empty `if` branch followed by an `else` branch, you can turn this into just a stand alone `if` branch by negating the `if`'s test and putting the code that was originally in the `else` branch in your new `if`.

```java
Scanner console = new Scanner(System.in);
System.out.print("how many dogs do you have? ");
int dogs = console.nextInt();
if (dogs >= 3) {
   System.out.println("wow! you have a lot of dogs!");
}
```

Note how the original `if`'s test, `dogs < 3`, was negated to `dogs >= 3` and how the println was moved to the new `if` branch in the fixed code.

2.4 indentation (∼ assignment 1)

The indentation of a program should increase by one tab every time a curly brace ({) is opened and decrease by one tab every time a curly brace (}) is closed. The number of spaces per tab doesn't matter too much (usually 3 or 4 is good). The below code is harder to read because it is indented poorly.

```java
public class SomeClass {
   public static void main(String[] args) {
System.out.println("hello");
   System.out.println("how are you?");
      System.out.println("programming is cool!");
}
}
```

But with proper indentation, the same code becomes much more readable.

```java
public class SomeClass {
   public static void main(String[] args) {
      System.out.println("hello");
      System.out.println("how are you?");
      System.out.println("programming is cool!");
   }
}
```

2.5 long lines (∼ assignment 3)

Any line greater than or equal to 100 columns long should be broken up (ideally, lines should actually be kept $<=$ 80 columns wide). This keeps code nice and compact (don't have to scroll horizontally to read). Note that columns include whitespace such as tabs. If you are using jGrasp, an easy way to check your line length is to click on the end of a line, then look in the bottom right corner. You should see "Col:" followed by a number. That is the number of columns in your line.

To break up long comments, simply break your line into two or more lines. Where exactly you decide to break a comment doesn't matter too much, but it's usually best to not break them when you're in the middle of a word! ☺

```java
// This program will do all of these things: a b c d e f g h i j k l and more
```

```java
// This program will do all of these things:
// a b c d e f g h i j k l and more
```

For long lines of code, you should first choose where you want to break your line. This can be a little bit arbitrary, but usually it is best to break lines *after* things like commas and operators such as plus signs and minus signs. After you have chosen where to break the line, you will need to introduce some type of hanging indentation to show that the second line is actually a continuation of the previous one. There are two conventions that you can use:

1. Leave a hanging indent of two tabs relative to the start of the first line.

   ```java
   1 public static void useless(int num) {
   2    System.out.println("here's the number passed into this method " +
   3          num + "!");
   4 }
   ```

   Line 3 has been indented by two tabs relative to line 2.

2. Or, if you are breaking up something within parentheses (such parameters in a method header), you can align the code inside of the parentheses.

```java
public static void reallyReallyLong(int one, int two, int three, int four,
                                    int five, int six, int seven) {
```

Notice that the beginning `i` in `int one` and the beginning `i` in `int five` have been lined up.

2.6 loop zen (∼ assignment 2)

2.6.1 choosing the right loop

First, you should consider whether you really need a loop in the first place. Remember that loops should only be used for *repeated* tasks; if you only need to do a task once, then there's no need to introduce a loop in the first place.

If you are sure that you need a loop, the next step is to decide what kind of loop to use. Remember that you should use `for` loops for repeated tasks where you know the number of repetitions in advance. If you don't know this in advance, or if it would be too difficult/inefficient to determine this in advance, you should use a `while` loop.

2.6.2 getting the most out of your loops

Loops are a good way to do as little work as possible, so make sure that you're letting your loop do as much of the heavy lifting as it can. For example, in the below code, consider whether or not line 1 is really necessary.

```java
1 System.out.print("*");
2 for (int i = 0; i < 4; i++) {
3    System.out.print("*");
4 }
```

Instead, why not just run the loop one extra time?

```java
for (int i = 0; i < 5; i++) {
   System.out.print("*");
}
```

2.6.3 only repeated tasks

It's key to remember that loops (both `for` and `while` loops) should only be used for *repeated* tasks. As already mentioned earlier, there's no point in having a loop that just runs once. Similarly, if you have something that only happens once (maybe at the end of a bunch of repeated tasks), then you probably don't need to put code for it inside of your loop.

```java
1 for (int i = 0; i < 4; i++) {
2    System.out.println("yay!");
3    if (i == 3) {
4       System.out.println("all done!");
5    }
6 }
```

In the above example, note that the `if` branch on line 3 will only ever execute once (on the very last iteration of the `for` loop). Since loops should only contain repeated tasks and we know exactly when we want the "all done!" message to be printed, this code should be pulled out to below the `for` loop.

```java
for (int i = 0; i < 4; i++) {
   System.out.println("yay!");
}
System.out.println("all done!");
```

2.7 main (∼ assignment 4)

There are no hard and fast rules that dictate how long or short your main method should be. On one hand, we don't want main to be too cluttered and detailed, because this prevents people from being able to quickly understand your program by reading your main method. On the other hand, it is also not good for main to be too concise. Main should read like the back cover of a novel or a table of contents – enough detail to give a good description of what the program is about, but not so much detail that everything is given away. Having a main that is too short, with just one or two method calls, would be like turning to the back cover of a book and seeing "story"; having a main that is too long would be like turning to the back cover of a book and seeing the entire story.

It is important to realize that having less lines in main does not necessarily mean that your main method is "better". Take a look at the below example.

```java
public static void main(String[] args) {
   intro();
   runProgram();
}

public static void runProgram() {
   method1();
   method2();
   method3();
   method4();
}
```

We know from reading main that there is definitely an intro being printed out in the program, but that's about it. To actually see what is happening in the rest of the program, we would have to do some further digging, and this is not ideal. We should be able to look at *just* main and get a good sense of what the program is doing. In essence, `runProgram` has become a second main method, because it has taken over main's job as the conductor of the program (once main calls `runProgram`, main no longer has control over what the program is doing). There's no need to introduce a second main method – we can just use the one that we started with. Here is an improved main method that is much more descriptive of the program.

```java
public static void main(String[] args) {
   intro();
   method1();
   method2();
   method3();
   method4();
}
```

This main method is longer than the first, but it also clearly shows the structure of the program. We can clearly see each crucial method that is being called.

To summarize, here are some questions that you can ask yourself when evaluating your main method:

1. Do I ensure that main is a concise summary by factoring out code like printing statements when appropriate?
2. Could somebody read through just my main method and come away with a general idea of what my program does?
3. Are all of the crucial steps or parts of my program well represented in my main method?

If you answered "no" to any of the above questions, then you may want to consider restructuring your main method.

2.8 methods (∼ assignment 4)

You could certainly write an entire program by putting all of your code in main, but it is good practice to instead factor different chunks of code into different methods. Code could be put into a separate method for many reasons, but here are two important ones.

1. **Reducing redundancy**. Putting repeated code into its own method allows you to easily make changes (you will only have to modify code in one place as opposed to two or more places). Remember that copying and pasting code should almost always be a red flag.
2. **Keeping main concise**. In some cases, even when code is not repeated, it may still be a good idea to put it into its own method if that code represents a discrete task. For example, let's say that you need to print an intro at the beginning of your program. The intro is something that will only ever need to be printed once in the run time of the program. However, it would still be a good idea to put that code in its own method because this prevents main from being too unnecessarily cluttered with details (if we just want an overview of the program, it's sufficient to know that the program has an intro; we don't need to know exactly what the intro says).

Here are some other guidelines to keep in mind when writing your methods.

1. **Each method should represent one concrete task**. If you were to describe your method to someone else, would you be able to give a succinct summary of its purpose, or would you find yourself listing out multiple tasks that the method is in charge of? Methods that have a clear purpose are much easier to reuse. For example, imagine that I have three tasks: task A, task B, and task C. I could potentially put all three tasks into one method. However, what if, in the future, I want to write another method that uses task A but not task B or C? I can't use the method that I have already written because that method performs all three tasks. A better approach would be to separate out the three tasks into their own methods. This allows me to easily reuse the code that I have already written.

2. **Avoid trivial methods**. Trivial methods are methods that are so short that they don't improve the structure of the program. In general, avoid methods that just have one `System.out.println` or `System.out.print`, and avoid methods that just contain one method call. It is sometimes okay to have one-line methods (for example, if you are writing a method to calculate a formula), but you should definitely think about whether putting so few lines of code into a method is worthwhile.

3. **Avoid useless returns and parameters**. A method should only return something if you intend to capture and use that value in the rest of your program. Otherwise, just leave it `void`. Similarly, any parameters that are passed into a method should actually be used. Sometimes, however, it is not enough to just make sure that parameters are used in the method. Some parameters can be considered "useless" even if they are manipulated in the method. For example, consider the below method's parameter `sum`.

```java
1 public static void someMethod(int repetitions, int sum) {
2    sum = 0;
3    for (int i = 0; i < repetitions; i++) {
4       sum += 7;
5    }
6    System.out.println("sum = " + sum);
7 }
```

   `sum` is definitely being used in this method, but note that its value is set to 0 on line 2. The original value of `sum` that was passed into the method is therefore irrelevant – whether it was 4, 56, or 100000 doesn't matter because it will always be immediately overwritten. Remember that parameters are used to pass information into methods. If we always immediately reset our variable to 0, then we are showing that we don't care about what information was passed into the method in the first place, and this is a strong indication that we should just keep the variable local to the method.

```java
public static void someMethod(int repetitions) {
   int sum = 0;
   for (int i = 0; i < repetitions; i++) {
      sum += 7;
   }
   System.out.println("sum = " + sum);
}
```

2.9 naming conventions (∼ assignment 1)

In Java, we commonly use camelCasing for names, which means that we capitalize each separate word in the name without using spaces, hyphens, or any other characters for separation. Here are the exact rules:

- **Class names**: should be camelCased and start with an uppercase letter. Example:
  - `public class SomeClass`
- **Method names**: should be camelCased and start with a lowercase letter. Example:
  - `public static void thisIsAMethod`
- **Variable names** (assignment 3): same as method names. Example:
  - `int hereIsAnInteger`
- **Class constants** (assignment 2): should be in all caps with underscores to separate words. Examples:
  - `public static final int DAYS_OF_THE_WEEK = 7;`
  - `public static final int WIDTH = 50;`

2.10 objects (∼ assignment 4/8)

2.10.1 fields (∼ assignment 8)

There are two important things to remember when working with the fields of your object.

1. **Encapsulation**. Without exception, you *must* make all of the fields of your object `private`. This ensures that the values of our fields can't be messed with by outside code.
2. **Initializing in the constructor**. It may be easy to take for granted at this point in the quarter, but remember that a line of code like `int someNum = 1;` is actually doing two things: first, it is *declaring* (the `int someNum` part), and second, it is *initializing* (the `someNum = 1` part). When

working with fields, you must declare them right underneath the class header and initialize them in the constructor. It is not good style to initialize at declaration.

```java
public class SomeClass {
   private int counter = 0;
   ...
}
```

Instead, separate the `private int counter` and `counter = 0` parts.

```java
public class SomeClass {
   private int counter;

   public SomeClass() {
      counter = 0;
   }
   ...
}
```

2.10.2 object creation (∼ assignment 4)

It is good practice to try to minimize the number of objects that your program creates. A lot of objects such as `Scanner consoles` and `Graphics gs` can be created once in the main method of your program and then passed as a parameter to any other methods that need them. This avoids having to create a new object for every method that needs access to one.

```java
public static void main(String[] args) {
   method1();
   method2();
}

public static void method1() {
   Scanner console = new Scanner(System.in);
   // do stuff with console
}

public static void method2() {
   Scanner console = new Scanner(System.in);
   // do stuff with console
}
```

```java
public static void main(String[] args) {
   Scanner console = new Scanner(System.in);
   method1(console);
   method2(console);
}

public static void method1(Scanner console) {
   // do stuff with console
}

public static void method2(Scanner console) {
   // do stuff with console
}
```

2.11 printing (∼ assignment 2)

A few guidelines for printing:

- When printing a blank line, remember that a `System.out.println()` with empty parentheses is preferred over a `System.out.println("")` with useless quotation marks.
- If you have a `System.out.print` that is directly followed by another `System.out.print` or a `System.out.println`, you can almost always combine the two to make your code more concise. For example, you would replace this:

```java
System.out.print("%");
System.out.print("%");
System.out.println();
```

  with this:

```java
System.out.println("%%");
```

- The escape sequence `\n` should only ever be used in `System.out.printf`s.

2.12 scoping (∼ assignment 4)

It is generally good practice to keep variables in the most local scope possible. Remember that the scope of a variable is the closest pair of curly braces that surround it. In the following example, the variable `nextScore` has been declared in the scope of the entire method, but note that declaring it in such a wide scope is actually not necessary.

```
public static void printTenScores(Scanner console) {
   int nextScore;
   for (int i = 1; i <= 10; i++) {
      System.out.print("next score?" );
      nextScore = console.nextInt();
      System.out.println("score " + i + " = " + nextScore);
   }
}
```

`nextScore` is only ever accessed inside of the `for` loop, so we can actually localize its scope to just the loop itself rather than the whole method.

```
public static void printTenScores(Scanner console) {
   for (int i = 1; i <= 10; i++) {
      System.out.print("next score?" );
      int nextScore = console.nextInt();
      System.out.println("score " + i + " = " + nextScore);
   }
}
```

2.13 spacing (∼ assignment 2)

Java is not a whitespace sensitive language, but maintaining good spacing throughout your program helps your code be more readable for human eyes. Most Java programmers follow the below spacing conventions for their programs.

- **Class headers**: `public class SomeClass {`
- **Method headers**: `public static void someMethod(int number1, int number2) {`
- **Method calls**: `someMethod(4, 3);`
- **Loops**:
  - `for (int i = 0; i < 4; i++) {`
  - `while (someTest) {`
- **Arrays**: `int[] someArray = new int[4];`
- **Variable initialization**: `int someNumber = 4;`

The spacing between methods also matters. Remember to leave a blank line between each method for readability.

```java
1 public static void method1() {
2     ...
3 }
4 public static void method2() {
5     ...
6 }
```

There should instead be a blank line after line 3, as shown below.

```java
1 public static void method1() {
2     ...
3 }
4 
5 public static void method2() {
6     ...
7 }
```

2.13.1 expressions

Spacing also becomes important when you have expressions in your program. Remember to keep a space to the left and right of operators such as +, -, *, and /. Notice how poor spacing makes expressions look squished and unreadable.

```java
int someExpression = x+4-y*3/2;
```

Introducing a bit of spacing makes a big difference.

```java
int someExpression = x + 4 - y * 3 / 2;
```

2.14 types (∼ assignment 4/5)

When declaring variables, one of the most important decisions you make is what type to assign to your variable. Choosing the type of your variable goes beyond just what works – given what the variable represents, you also want to consider what type is most appropriate.

2.14.1 `int` vs `double` (∼ assignment 4)

`int`s are whole number values such as 0, 1, 2, 3, ... , etc. `double`s, on the other hand, can have decimal places; for example, 1.23, 4.0, 9.0000001, etc. It is worth noting that, though we may think of them as equivalent in regular math, when programming, 4 is not the same as 4.0 (this is beyond the scope of the class, but this is because `int`s and `double`s are actually stored differently by your computer).

One of the advantages of using doubles, especially in calculations, is that you eliminate the risk of accidentally performing int division (when you divide an int by an int and get an int, possibly resulting in a loss of precision from the values after the decimal place that are truncated). Because of this, many students end up declaring many of their number values as doubles. However, declaring a variable as a double just for convenience is poor style. For example, if you had a variable numOfSiblings, it would be a misuse of type to make this variable a double because it doesn't make sense to talk about having 2.8 siblings; instead, this number should be declared as an int because the number of siblings that a person has will only ever take whole number values. On the other hand, if you have a variable weightInPounds, you may choose to make this number a double because it makes sense to talk about weights with a higher level of precision (ex: 200.8 pounds).

The bottom line is that you should declare numbers as what they are – and if you need to avoid things like int division, then you should cast when needed. Examples:

```java
// doesn't make sense to declare numOfSiblings1 and numOfSiblings2 as doubles
double numOfSiblings1 = 2.0;
double numOfSiblings2 = 1.0;
double averageSiblings = (numOfSiblings1 + numOfSiblings2) / 2;
```

```java
// instead, cast!
int numOfSiblings1 = 2;
int numOfSiblings2 = 1;
double averageSiblings = ((double) numOfSiblings1 + numOfSiblings2) / 2;
```

```java
// or, change the 2 to a 2.0
int numOfSiblings1 = 2;
int numOfSiblings2 = 1;
double averageSiblings = (numOfSiblings1 + numOfSiblings2) / 2.0;
```

2.14.2 int vs boolean (∼ assignment 5)

A possible misuse of ints occurs when students use them as though they are booleans. Usually, these int values will only take two possible values – of these values, one will be used to indicate true, and the other will indicate false. Take a look at the below example.

```java
Scanner console = new Scanner(System.in);
System.out.print("how many siblings do you have? ");
int numOfSiblings = console.nextInt();
int moreThanTwo;
if (numOfSiblings > 2) {
   moreThanTwo = 1;
} else {
   moreThanTwo = 0;
}
if (moreThanTwo == 1) {
   System.out.println("you have more than two siblings!");
}
```

In this example, the variable `moreThanTwo` is being used like a `boolean` because it is set to 1 to indicate `true` and 0 to indicate `false`. This is a poor use of type because you don't need an `int` to implement this behavior; if all you want is a value that will switch back and forth between two values, you should just use a boolean.

```java
Scanner console = new Scanner(System.in);
System.out.print("how many siblings do you have? ");
int numOfSiblings = console.nextInt();
boolean moreThanTwo = numOfSiblings > 2;
if (moreThanTwo) {
   System.out.println("you have more than two siblings!");
}
```

(If you need an explanation of the `boolean` zen being used in the above corrected example, head over to the `boolean` zen section of this style guide).

2.15 variable names (∼ assignment 2)

Using descriptive variable names greatly enhances the readability of your program. In general, non-descriptive one or two letter variable names should be avoided whenever possible, because `int a` is a lot less helpful to read than `int average`, for example. There are, however, a few exceptions.

2.15.1 acceptable one letter names

- `Graphics g` – because this is a convention that we will be using throughout the quarter.
- `int x`, `int y` – in the context of coordinates, these names are okay because, unlike other one letter variable names, they are not non-descriptive. Using the names `x` or `y` almost immediately brings coordinates to mind, so even though these names are extremely short, they still manage to describe the values well.

- Loop variable names – using the `i`, `j`, `k` convention in `for` loops is totally fine (keep reading for more on loop variable names).

2.15.2 loop variable names

When choosing names for the counter variables in `for` loops, you have two options.

1. **The `i`, `j`, `k` convention**. If you want to use one letter loop variable names, you should use this convention (as opposed to using other arbitrary one letter loop variable names such as `a`, `b`, `c`, etc.). This conventions says that all outer loops should use `i`, all once-nested loops should use `j`, and all twice-nested loops should use `k` (any further nesting beyond this point is highly unlikely). Here is an incorrect attempt at using the `i`, `j`, `k` convention:

```
for (int i = 0; i < 4; i++) {
   for (int k = 0; k < 3; k++) {
      ...
   }
}
for (int j = 0; j < 2; j++) {
   for (int k = 0; k < 10; k++) {
      for (int i = 0; i < 11; i++) {
         ...
      }
   }
}
```

   Here is the corrected code:

```
for (int i = 0; i < 4; i++) {
   for (int j = 0; j < 3; j++) {
      ...
   }
}
for (int i = 0; i < 2; i++) {
   for (int j = 0; j < 10; j++) {
      for (int k = 0; k < 11; k++) {
         ...
      }
   }
}
```

2. **Descriptive loop variable names**. If you don't want to use one letter loop variable names, you can instead name your loop variables descriptively (ie: just treat them like regular variables and try to give them fitting names based on how they are being used). Some examples:

```java
for (int row = 0; row < 4; row++) {
   for (int star = 0; star < 7; star++) {
      System.out.print("*");
   }
   System.out.println();
}
```

Either the `i`, `j`, `k` convention or using descriptive loop variable names is acceptable. Arguably, using descriptive loop variable names makes your program slightly more readable, but which one you use is up to you. It is, however, ideal that you pick one of these conventions and stick to it throughout your entire program. For example, this mixing would not be ideal:

```java
for (int i = 0; i < 4; i++) {
   for (int star = 0; star < 7; star++) {
      System.out.print("*");
   }
   System.out.println();
}
```

Commenting

Well-documented code is much easier to maintain, update, and understand – not only for other people, but also for yourself. Throughout the quarter, we will introduce you to more and more types of comments that you should include in your programs.

3.1 required comments

3.1.1 class comment (∼ assignment 1)

When writing an essay for an English class, you have probably included a header at the top of your paper with your name and other identifying information. We expect the same type of header to be included at the top of every `.java` file that you turn in. In addition, your class comment should include a brief description that gives an overview of what your program does. Here is an example of a class comment for a fictional program.

```java
// Michelle Yun
// 9/21/16
// TA: Bill Nye the Science Guy
// Assignment 42: TakeSurvey.java

// This program will give a survey to the user. At the end of the program,
// the user can choose to save their answers to a text file, and a brief report
// of the user's results will be printed to the console.
```

3.1.2 method comments (∼ assignment 1)

Every method that you write (except for main; main doesn't need a comment because your class description basically acts as main's description, too) must have a comment directly above it that describes the following:

1. What the method does
2. If applicable, what parameters the method accepts (∼ assignment 3)
3. If applicable, what the method returns (∼ assignment 4)

The exact formatting of the comment does not matter much as long as you address all of the above. Here is one example method that would need a comment.

```java
public static int magicNumberPrompter(Scanner console, int magicNumber) {
   System.out.print("what's the magic number? ");
   int nextTry = console.nextInt();
   int totalTries = 1;
   while (nextTry != magicNumber) {
      System.out.print("wrong! what's the magic number? ");
      nextTry = console.nextInt();
      totalTries++;
   }
   System.out.println("you needed " + totalTries +
                      " tries to get the magic number!");
   return totalTries;
}
```

And here are three differently formatted comments for the above method. All three would be acceptable. Note that the third is written in Javadoc, but you are *not* required to learn Javadoc for this class.

```java
// Given a console Scanner and desired number (magicNumber), continues to
// prompt the user for numbers until the user types in the magic number and
// returns the number of tries that the user took.
public static int magicNumberPrompter(Scanner console, int magicNumber) {
```

```java
// Continues to prompt the user for numbers until the user types in
// the magic number. Returns the number of tries that the user took.
// parameters needed:
//    console     = to prompt for numbers
//    magicNumber = desired number
public static int magicNumberPrompter(Scanner console, int magicNumber) {
```

```java
/**
 * Continues to prompt the user for numbers until the user types in
 * the magic number.
 *
 * @param console to prompt for numbers
 * @param magicNumber the desired number
 * @return the number of tries that the user took
 */
public static int magicNumberPrompter(Scanner console, int magicNumber) {
```

3.1.3 inline comments (∼ assignment 4)

Class and method comments appear above headers. Sometimes, it is helpful to include comments that are right next to lines of code. These are called inline comments. Inline comments should be included whenever you have line(s) of code where the logic is not immediately apparent. Try imagining that you are someone else and looking through your code – is there anything potentially confusing? If so, an inline comment or two may be needed. Here are some common places where inline comments may be useful (not all-inclusive):

- Initializing variables to values besides 0 or 1
- Priming `while` loops
- Complex expressions

Inline comments can appear right next to the line of code that you would like to comment on.

```java
// Given a Random object, prints a random number (1-5 of them) of even random
// numbers in the range 2-100
public static void printRandomNumbers(Random r) {
   int repetitions = r.nextInt(5) + 1; // want repetitions in the range of 1-5
   for (int i = 0; i < repetitions; i++) {
      int nextRandom = (r.nextInt(50) + 1) * 2; // even numbers between 2-100
      System.out.println(nextRandom);
   }
}
```

Or, if that makes your line too long, above the line of code.

```java
// Given a Random object, prints a random number (1-5 of them) of even random
// numbers in the range 2-100
public static void printRandomNumbers(Random r) {
   // want repetitions in the range of 1-5
   int repetitions = r.nextInt(5) + 1;
   for (int i = 0; i < repetitions; i++) {
      // even numbers between 2-100
      int nextRandom = (r.nextInt(50) + 1) * 2;
      System.out.println(nextRandom);
   }
}
```

3.2 templates

3.2.1 class comment

```
// Your name
// Today's date
// TA: Your TA's name
// Assignment #_: _______

// This program...
```

3.2.2 method comments

Note that the text surrounded by the "<" and ">"s should be replaced with the appropriate descriptions/names.

```
// <Description of method + returns (if applicable)>
// parameter(s) needed:
//    <parameter1> = <description of parameter1...>
//    <parameter2> = <description of parameter2...>
//    <parameter3> = <description of parameter3...>
```

3.3 comments to avoid

3.3.1 implementation details (∼ assignment 4)

In general, it is way harder to comment too much than to comment too little, but there are some types of comments that you should avoid using in your program. When writing class and method comments, you should avoid implementation details, which give away internal information about how your code works. Implementation details are usually phrased in terms such as "does x y z by ..." or "uses a ... to ..." and use programming keywords such as `for` loop, class constant, `println`, etc. When writing class/method comments, you should imagine that you are writing to an audience that has very little/no programming experience – so avoid using coding jargon as much as possible. Notice how the below comment mentions the use of a `for` loop.

```
// uses a for loop to print out a triangle made out of stars
public static void printTriangle() {
   for (int line = 1; line <= 4; line++) {
      for (int stars = 1; stars <= line + 1; stars++) {
         System.out.print("*");
      }
      System.out.println();
   }
}
```

Instead, this comment should be phrased in a way that doesn't include implementation details.

```java
// prints out a triangle made out of stars
public static void printTriangle() {
   for (int line = 1; line <= 4; line++) {
      for (int stars = 1; stars <= line + 1; stars++) {
         System.out.print("*");
      }
      System.out.println();
   }
}
```

The bottom line is that you should be writing high-level descriptions of your class/methods that focus on *what* your code does, not *how* it does it.

3.3.2 words taken verbatim from homework write-ups

This one should be fairly obvious. We want you to describe your code in your own words, not in ours, so copying and pasting from the spec or just changing around a few words is not allowed.