Why Comment?

There have been plenty of classes that we have already used, like `Scanner` or `Random`. The great thing about programming is that you don't have to reinvent the wheel. Plenty of programmers will design helpful classes, and we can just use their code! When we use a class that someone else has already created, we are the **client** of their class and they are the **implementer**. Comments are the way that an implementer can get a client to fully understand all the details they need to know to use their class properly. A client should be able to understand how to use a method fully just by reading the comment and the method header!

Format

- Your comments should all be formatted consistently. You can use pre/post, JavaDoc, or some other format that includes all the required information.
- Use either multiline comments (/* */) or single line comments (//).
- No line should be longer than 100 characters wide. Having long lines decreases the readability of your code
- Each comment should have the same level of indentation as the code it is commenting

Comment Tags

There are a couple of ways to designate a comment in Java

```
// You can write comments with two slashes for every line of comments
// You need to put slashes on every line, which can be tedious.

/*
 * You can also write blocks of comments using comment open/close tags.
 * The commented section starts with /* and ends when a */ appears. Do not
 * use these for internal method comments.
 */
```

Implementation Details

What we are most concerned with when documenting code for a client is getting them to understand *what* our code does and now *how* it does this. This is a key concept known as abstraction. Imagine that you got a manual for a radio. The manual tells you what each button will do and how they will interact with each other, but if the manual started going into how turning the volume knob increases the voltage across the resistor in the circuitry, you'd quickly get confused! We shouldn't confuse clients of our code with unnecessary documentation of the internals of our methods. A client should only know the information they need to know to use a comment properly and leave any other "implementation details" out.

The way that we should think about **what a client "sees" is that they see all the comments and method headers for anything that is public**. They will not see the insides of your methods, nor will they see anything private like private fields or private helper methods. Talking about those things is considered implementation details. Note that the one exception to this is when you are commenting these private helper methods, private fields, and internal code. Leaving "internal comments" is helpful for you and other implementers (people who are working on that same class as you) to understand what you are doing in your code, and talking about implementation details there is fine because those comments are reserved for other implementers rather than clients of your code.

This means that you should never mention anything that a client wouldn't see in the behavior of using the object. Let's take a look at the following example

```
// Tyler Mi
// TA: Jake Peralta
// The ArrayIntList class represents a list of integers. It
// keeps track of the elements inside with an int[] elementData
// and an int size.
public class ArrayIntList {
     private int[] elementData;
     private int size;

     // Counts up all the integers in elementData with a for-loop
     // up to size, and returns the sum
     public int sum() {
          int sum = 0;
          for (int i = 0; i < size; i++) {
               sum += elementData[i];
          }
          return sum;
     }
}
```

Note that in the above code example, while they are very informative, we are talking about our private fields elementData and size in multiple places. These are fields that the implementer use (like the circuitry of the radio) and documenting them is not useful at all in understanding how to use the class or its methods.

Let's look at how it could be improved.

```
// Tyler Mi
// TA: Jake Peralta
// The ArrayIntList class represents a list of integers that
// can be variable in size.
public class ArrayIntList {
     private int[] elementData;
     private int size;

     // Returns the sum of all the elements in the current ArrayIntList
     public int sum() {
          int sum = 0;
          for (int i = 0; i < size; i++) {
               sum += elementData[i];
          }
          return sum;
     }
}
```

Class Comment

You should always include an overall comment for any classes that you write. This should include your name, your TA, and a brief description of the class you are writing. What can objects of this class do? What is the purpose of using this class?

Method Comments

Every method, **whether it is public or private**, should be documented with a comment. In each method, you should talk about Exceptions, Returns, Parameters, and Behavior

Exceptions:

Oftentimes, we will add exceptions into our code in order to prevent clients from passing using our objects in incorrect ways that might break the object. It is imperative that we document exactly what exceptions will be thrown and under what conditions they are thrown. This is so a client understands exactly what they might have done wrong when using your object and what they can do to avoid having exceptions be thrown.

Parameters:

We should document all parameters to our methods and how it affects the behavior of the method/program. This also includes documenting any requirements of the parameters, such any required format of the object or the state of the objects inside. This may differ from exceptions in that an exception is not thrown, but the object needs to be in a particular format for the method to behave in an acceptable manner.

Returns:

If the method returns something, we should document what the value of the return signifies. If any special values are returned in certain cases (eg. `null`), we should document the conditions in which such a return would occur.

Behavior:

We should describe the method in a broad sense. We should document any behavior that a client might see, including any edge cases that might occur. How will the object's *public* state change? For example, when we add into an ArrayList at a specific index, it's important to note that all of the elements at that index and beyond shift over 1 and the size increases by 1, rather than just replacing what is at that spot of the ArrayList. If we are documenting a toString() method, we can document the format of it. Remember however, that you never should reveal implementation details in this. We should discuss ***what*** about the object will change in ways that the client sees, but never ***how*** we do it.

Example:

Let's say we're commenting the following method in the following way

```
// Adds numbers using a for-loop
public static int summation(int max) {
	if (max < 0) {
		throw new IllegalArgumentException();
	}
	int sum = 0;
	for (int i = 1; i <= max; i++) {
		sum += i;
	}
	return sum;
}
```

In ways, this comment doesn't give us enough detail and in other ways, gives us too much. It doesn't tell us anything about what is being returned, and yet it gives us implementation details about how exactly we are implementing it internally by revealing we are using a for-loop. It also doesn't document the throwing of the exception, which a client might run into! Let's see how we can fix it, step by step:

Fixing Exception Documentation:
Let's say we tried to document the exceptions like this:

```
// Adds numbers using a for-loop. Throws an exception if number isn't positive.
public static int summation(int max) { ... }
```

This improves on before by saying an exception will be thrown but it's not quite there. We should mention the exact type of exception that is thrown so it is clear that the exception they saw was the expected one. This is particularly important if a method can throw multiple different types of exceptions, as a client can identify what they did wrong depending on the specific exception they saw. Another key issue is that the method only throws an exception if the parameter is less than 0. The above comment however documents it as if an exception would be thrown if the parameter was 0, as 0 isn't a positive number. Thus this documentation is close but inaccurate. Let's fix these issues!

```
// Adds numbers using a for-loop. Throws an IllegalArgumentException if given
// max is less than 0.
public static int summation(int max) { ... }
```

Fixing Parameter Documentation:
Currently, we don't describe how the parameter affects the behavior or result of the method at all. Let's fix that:

```
// Adds integers from 1 up to and including the given max using a for-loop.
// Throws an IllegalArgumentException if given max is less than 0.
public static int summation(int max) { ... }
```

Note that we were specific in stating that the sum is from 1 and *up to and including* the given max.

Fixing Return Documentation:
While it might seem intuitive from existing comment and knowing from the method header that it will return an int, that the returned int is the sum from 1 to the given max, it never hurts to be explicit!

```
// Adds integers from 1 up to and including the given max using a for-loop,
// then returns it. Throws an IllegalArgumentException if given max is
// less than 0.
public static int summation(int max) { ... }
```

Fixing Behavior Documentation
Here we should look over what we have, make sure that there's no edge cases we need to document and get rid of any implementation details. Something to consider when reading this method is the case when 1 is returned. The comment currently says "from 1 up to and including 0". This might be small, but what if 0 is passed in? It is confusing for someone to count from 1 *up to* 0. We can be explicit on what happens when 0 is passed in and get rid of the implementation details.

```
// Returns the sum of the integers from 1 up to and including the given max. If
// max is 0, 0 is returned. Throws an IllegalArgumentException if given max is
// less than 0.
public static int summation(int max) { ... }
```

Fixing Overall Formatting

Looking pretty good! The contents are all there. If you want to have code that is super easy to understand however, we recommend taking inspiration from JavaDoc, a format used by professional programmers to document Java code. You can leave an overall comment, then for each of the parameters, returns, and exceptions, you use a would use a tag to make it clear what you are documenting. See below for an example!

```
/*
 * Calculates the sum of the integers from 1 to the given int
 * @throws - IllegalArgumentException if given max is less than 0
 * @param max - the maximum integer to add up to
 * @return - the sum of the first "max" positive integers. If max is 0,
 *           then 0 is returned
 */
 public static int summation(int max) { ... }
```

This comment might seem like a lot to create, but there are no questions as to how the parameter affects the return or behavior of the method, no matter the input.

Some of you math wiz's might know that there's actually a shorter formula to calculate this sum that doesn't require any adding of values in a loop. We could change the internal implementation of the method to this formula, (namely the sum of values from 1 to n is $\frac{n * (n+1)}{2}$.

```
/*
 * Calculates the sum of the integers from 1 to the given int
 * @throws - IllegalArgumentException if given max is less than 0
 * @param max - the maximum integer to add up to
 * @return - the sum of the first "max" positive integers. If max is 0,
 *           then 0 is returned
 */
 public static int summation(int max) {
     if (max < 0) {
          throw new IllegalArgumentException();
     }
     return (max * (max + 1)) / 2;
}
```

Note that even after changing the internal implementation of the method, our comment still holds true! This is a great indication that we didn't reveal any implementation details and that we kept only the necessary details for a client to fully understand our code!