Variables and Data Types

You've heard of variables before. In fact, you've worked with them quite a bit–in high school math and science at the very least. The variables we'll be covering today are a bit more complex, obviously, but the principle remains the same: assigning values to abstract names so they can be used elsewhere. Let's get into it.

#What is a Variable in Computer Programming?

Variables are used whenever there's a need to store a data. Variables can be used to contain any number of things, from simple strings and integers to complex objects used to store dozens (or hundreds) of values. Many computer science courses define a variable as being "a value associated with a specific memory location on a computer," but we're mostly concerned with the practical side of variables.

Let's talk about creating a variable. In some languages, making a variable is as easy as saying `let x = 5`, but Java is a little bit different. Java is what's called a **strongly-typed language**. A strongly-typed language is one that requires specific declarations for every variable that the program handles. When you intend for Java to know that `5` is an integer and not a string like `"5"`, you have to tell it that. Similarly, you have to tell Java what kinds of values are supposed to be returned from the methods you create. This may seem strange or unnecessary at first–why would `5` be a string and not a number, right?–but it's actually really helpful in a lot of ways.

Weakly-typed languages, like Javascript, can suffer from many unintended consequences that lead to confusing and frustrating bugs. Indeed, Javascript is experiencing a renaissance of sorts where developers are choosing to impose strict type rules upon it to make it work more like Java. We'll learn more about all of this when we get to Javascript in month three. For now, understand that strongly-typed languages have their benefits, even if it means a bit more boilerplate code at the beginning.

No matter what kind of programming language you're using, variables allow you to modify, reuse, and pass code from one place to another. They're one of the fundamental building blocks of code, and you're going to be making a lot of them.

#Variable Declaration and Initialization

To **declare** a variable in Java, you need only write the type of the variable and the name of that variable, followed by a semicolon:
CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
int x;
```

The above would create a variable called `x` ready to contain an integer (type `int`). It would also reserve some space for that variable in your computer's memory. Later on, when we need to assign a value to that variable, we can **initialize** that value like so:
CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
x = 42;
```

Initializing a variable means assigning it a value. Now, the space in our computer's memory that was reserved for `x` holds the value `42`.

In modern programming, reserving space in this way is not as necessary as it once was. We'll discuss when to declare a variable without initialization as we progress through the course, but we'll focus on the more common pattern here and now: you're usually going to see variable declarations that initialize a value at the same time.

To **declare** and **initialize** a value in one line, you're going to follow this basic pattern:

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
type variableName = value;
```

Let's break down the pieces:

- **Type:** This refers to the type of data that the variable contains. Examples include `String` for text, or `int` (short for integer) for a whole number.
- **The Variable's Name:** These names can be whatever you want, as long as you don't attempt to use any reserved keywords. We'll see many reserved keywords as you go, but there aren't so many that your variable names are likely to clash. If you get an error, simply change it to something that doesn't cause that error.
- **The Assignment Operator:** The equals sign is also known as the assignment operator. Use the `=` equals sign to assign the value to the variable that you're creating, or to change the value associated with the variable at a later time. More on that below.
- **The Value:** This is the value that you want the variable to contain. It must match the type you've declared, or you'll get a syntax error. Technically, defining a value is optional, as shown above. You can initialize a variable by declaring its type and giving it a name without using the assignment operator to give it a value. Then, you can assign it a value later.
- **The Semicolon:** At the end of a line declaring or reassigning a variable, you need a semicolon. Remember: Java requires this simple signal in order to know when to start and end its interpretation of a chunk of code. You'll get syntax errors if you forget it.

Once you've declared a variable, you're welcome to use it wherever you like. In the IDE below, fill in your name on line 3, then click `Run`. The results should be predictable:

Run

Main.java

```
3
4
```

```
        String name = "";
        String message = " I love Java";
        System.out.println("Hello " + name + ","+ message);
    }
}
public class Main {
    public static void main(String[] args) {
```

CONSOLESHELL

InfoWarningTip
Remember: you're always welcome to experiment in these IDEs. If there's anything on the page you'd like to test out, feel free to use this or another IDE for experimentation purposes when time allows.

#Discussing Data Types: Primitives

Java's most basic data types are known as **primitive data types** and are already built into Java by default. Take note that primitive data types start with a **lowercase** letter, and that number sizes are determined by binary values. We'll explain why after the list:

The primitive types are as follows:

- **char:** `char` is used to represent a single letter, such as `'A'`, `'B'` or `'c'`. It cannot be used for longer strings. Note: `char`s can be upper or lowercase, but they must be defined in single quotes.
- **boolean:** A `boolean` value can only be **true** or **false**. You'll be hearing the word "boolean" a lot throughout the course, so it's a good idea to get used to it meaning `true` or `false`. Careful: some other programming languages use 0 to represent false and 1 to represent true, but Java uses only the keywords `true` and `false` in boolean variables.
- **byte:** A `byte` is an integer value between -128 and 127 inclusive.

- **short:** A `short` is an integer value between -32,768 and 32,767 inclusive.
- **int:** Short for integer, `int` is used for whole numbers between -2^{31} and $2^{31} - 1$.
- **long:** A `long` is an integer value between -2^{63} and $2^{63} - 1$ inclusive. If you need a really large number without decimals, long is probably the type you're after.
- **float:** The `float` type lets you use decimals. How big it can be is rather complicated (it's a 32-bit number, like the `int` type). Just know that this is where normal human-scale numbers with decimals generally are. It has a precision of about eight digits. That means that if you store a number like 12.3456789 (ten digits), the number will be rounded to eight digits, which is approximately 12.345678.
- **double:** The `double` allows us to use decimals and is much larger (a 64-bit number) than the `float`. For decimals, this type is usually chosen above `float` as it offers a bit more space for numbers that might get more complex. It has a precision of about fifteen digits.
- Lastly, we have `null`. This is technically a data type, but it's value can only ever be "null", so it's not much use as a variable type unless you're trying to declare something `null`. A null value is one that has no value whatsoever and is intentionally left empty.

The numeric types listed above follow a basic **binary** pattern. The reason is simple: the ones and zeroes that computers understand are binary, and the more we cater to those hardware limitations, the more easily we can ensure that our program is optimized. When Java was new, computer hardware was far less powerful than it is now, and every bit of data needed to be accurately and carefully managed. When using a smaller data type, such as a byte, you can ensure that your program only reserves a tiny amount of space for numbers that could potentially be assigned to that variable. The larger the data type, the more space needs to be reserved. In short: when you create a **primitive variable** Java will set aside enough bits in memory for that primitive type and associate that memory location with the variable name that you used in order to save memory.

In the modern world, we aren't nearly as concerned about hardware space, but it's still a good idea to make sure you're using the right data type for the job. You're going to learn this while working on projects throughout the course, so **don't feel**

compelled to memorize any of the above. If you'd like to know more about binary and other counting systems, we've linked a video in the Extra Resources section below.

InfoWarningTip
Note that the `String` type, which we've already used and discussed to some extent, is **not** a primitive data type. It's one of a set of more complex data types known as a reference type. We'll cover them more below.

#Variable Reassignment

At any time after declaring your variable, you can reassign its value by simply calling the variable's name then following it with the assignment operator `=` and a new value. Follow the comment instructions in the IDE below to try it out:

If you want to create a variable whose value can **never change**, simply put the keyword **final** before your type declaration like so:
CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
final int x = 42;
```

Feel free to copy and paste this code into the IDE above to test its effects. We recommend putting it on line 6 or so to see what happens when you try to reassign a variable declared as `final`.

#Reference Type Variables

Primitive variables are useful, but there are way more variable types available to us. Besides the categories described above, all other variables will fall under the **reference variable** umbrella. Reference variables are so called because of the way that Java stores information on your computer when it's running code. For primitive types, the code is stored directly. If you say `int x = 3;` , Java will establish a space in

memory for an integer called `x` and store the value `3` right there with the variable name. With reference variables, Java simply establishes a reference to a **different** space in memory where the data for that variable will be stored. This reference is sometimes called a **pointer,** and it gets accessed any time we interact with that variable. Therefore, when we ask Java to get us the value of a reference type variable, it will check the reference, then go find its place in memory, then return what we've asked for. This can lead to some interesting behaviour that we need to be aware of, but we'll get into the weeds with what all of this means as we learn how to code.

The reason that reference variables are stored this way lies in the fact that they're far more variable. Strings, such as `String hello = "hello world";`, are stored as reference variables because the information contained in a string can be far more complex than what an `int`, or even a `double` might contain. What if we changed the actual text for `hello` to be the entire text of Shakespeare's works? It's possible to do this, and that variability plays against the built-in strengths of Java's primitive types. When we go to copy that variable, will we have to copy the entirety of the text? Given what computers are capable of, that would seem like a terrible idea.

This is why Java stores these variables as references. Rather than re-copy the value of the variable, we can have several different variables that all **point** to this one chunk of text. Because they all point to the same position in the computer's memory, we don't have to duplicate anything at all.

We will soon talk about classes, arrays, and objects, all of which can be used to contain different kinds of information that can similarly balloon beyond the memory allocation capabilities that Java reserves for primitive data types. It's not important that you grasp everything about them just yet. For now, it's best just to understand how to reference type variables are different from primitives. Here's a handy table of differences:

Reference Type Variables	**Primitive Variables**
Type declarations begin with Uppercase letters.	Type declarations begin with lowercase letters.

Space reserved in memory is the same for all reference types	Space reserved in memory is specific to the type declared
Can store a null value	Cannot store a null value
Examples: classes, arrays, strings, interfaces, objects	Examples: int, boolean, double, long
Generally, not pre-defined by Java, except for String variables	Always pre-defined by Java.

We'll see more soon, but keep the above in mind as we delve into other variables throughout the week and into the future.

#Summary

- A **variable** is a name for a memory location where you can store a value that can be changed later.
- A **variable declaration** in Java must indicate the type and name of the variable.
- **Variable initialization** means assigning a value to a variable for the first time.
- Use the assignment operator `=` to assign a value to the variable. You must initialize a variable before using it in an expression.
- **Data types** can be categorized as either primitive type (like `int`) or reference type (like `String`).
- The three primitive data types used most throughout this course are **int** (integer numbers), **double** (decimal numbers), and **boolean** (true or false).
- Each variable has an associated store of memory that is used to hold its value.
- When a variable is declared `final`, its value cannot be changed after initialization.
- Reference-type variables are used to contain data that are more complex than what primitives can contain.
- Rather than directly storing the information with which they're associated, reference data types provide a **pointer** that tells Java where to look for the data stored by the reference variable.

#Extra Resources

Here's a helpful video about binary and other counting systems. This video even touches on how URL shorteners work by using interesting math concepts–and it's only 5 minutes long.

Open in new tab

More information about Java data types:
https://www.w3schools.com/java/java_data_types.asp.

More information about Java variables:
https://www.w3schools.com/java/java_variables.asp

Working with Variables

We've already discussed basic declarations, initializations, and the primitive data type. Now let's talk a bit about what data types do for us, and how to properly utilize some of the basic types in our code.

#Static Typing

When we declare a variable, we must declare what type of variable it is going to be. From then on, we can reassign the value of the variable whenever we want, so long as we maintain the declared type. This is called **static typing**, and it's here to prevent us from making errors that could cause our systems to throw errors or worse–to crash.

The logic of static typing is simple: if a variable starts out as a number, there's a high likelihood that you want Java to always find a number in that variable. After all, numbers are used to do numerical things like math operations, which means this variable should remain a mathematically viable number for the duration of your code. It wouldn't make much sense to tell Java to divide `5` by `"cat"`, so a variable that starts out as a number (like `int`) will always remain a number–else Java will throw an error.

These errors aren't technically syntax errors, but we group them together as what are known as **compile-time errors**. This is the name we've given to errors that occur when Java attempts to compile our code and get it ready for use. They're automatically detected, and they keep us safe from making worse mistakes down the road.

The type of a variable, therefore, defines an entire domain for that type. This includes how the value should be written **and** the kinds of operations that can be applied to it. In some cases, it also defines the ways that operators (like `+`) affect them. For instance, you may have noticed the use of the `+` sign in some of our code so far. What do you think this line would output?
CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
System.out.print("Hey, " + "how " + "are " + "you?")
```

Think about it, then reveal the answer if you like:
Reveal Content

But why? We'll learn more about **operators** soon, but for now, it's important to note that the plus sign operator works according to the domain defined for the type of data it has been asked to operate upon. In the case of strings, it **concatenates** them, tacking them onto one another to create a single stream of text. With numbers, it will add them, but other behaviours are possible, too. Feel free to play around in one of your IDEs to see what else can happen.

InfoWarningTip
Note: there are ways to change a variable's type after declaration and initialization, but Java is a strongly-typed language for a reason. It's best to stick to the type you assigned your variable whenever possible.

#String: a Reference Variable

A `String`, as previously mentioned, is not a primitive type. While `char`, which can be used to store a single letter, is a primitive, `String`s are what's known as **reference** or **object variables**. With primitive types, specific amounts of space are set aside to hold specific values in specific positions in the system's memory. `String`s, however, only store a reference to that information, which is stored as a more complex form of data called an **object**. The best way to think about object variables is through abstraction. Rather than holding the data itself, they establish a reference to a specific place in memory, which will then be used to hold that data. One analogy would be a tracking number for a package you've ordered. The tracking number helps you to know where the package is, but it doesn't necessarily define the contents of the package itself.

Strings are values wrapped inside of a set of quotes `""`. They can contain text and/or numbers and symbols, all of which will be treated as a single unit by your Java program. What you type inside of a string will not affect your code, but will instead be interpreted literally. `"Hey + you"` will not print `"Heyyou"` if you feed it to a `System.out.print method`, but will rather print exactly what it says: `"Hey + you"`.

Another detail to be aware of is what happens when you run code like this:

In the above code, why does line 9 print the text "helloStatement" instead of "Hello World"? Think it through, then reveal the answer to check your understanding:
Reveal Content

Beyond this strangeness, object variables can behave differently from primitives, depending on how we attempt to use them. For now, that's beyond the scope of our learning, so let's stick with something more pressing: how to name your variables properly.

#Naming Variables

While you can name your variable almost anything, some rules should always be observed. For starters, a variable name should start with an alphabetic character (like a, b, c, etc.) and can include letters, numbers, and underscores `_`. It must be all one

word with no spaces, as spaces will tell Java that you're starting a new word/bit of code.

Second, you can't use any of the keywords or reserved words as variable names in Java (`for`, `if`, `class`, `static`, `int`, `double`, etc). For a complete list of keywords and reserved words see here, or experiment and see what happens. Java is good at telling you when you're using reserved words, and your IDE itself may even call you out if you're doing something you're not allowed to do.

Third, the name of the variable should describe the data it holds. A name like `score` helps make your code easier to read, assuming the information it contains is actually a score. Names like `x` are usually not a good idea because they give no clues as to what kind of data they hold. For rather opposite reasons, you should also avoid naming variables crazy things like `thisIsAReallyLongName`. Your goal is to make your code easy to understand, not harder.

Finally, the convention in Java and many other programming languages is to always start a variable name with a lower case letter and then uppercase the first letter of each additional word. This is called camelCase. We do this because variable names can not include spaces, and uppercasing the first letter of each additional word makes it easier to read the name. Another option is to use an underscore `_` to separate words. This is called snake_case, and is generally used for specific instances in Java. See the Extra Resources for a link with more information on naming conventions.

InfoWarningTip
One word of warning: variable names are case-sensitive and spelling sensitive! Each use of the variable in the code must match the variable name in the declaration exactly. Keep yourself consistent, or you'll have quite a few more errors than you anticipated.

In summary:

- Use meaningful variable names!

- Start variable names with a lowercase letter, and then use camelCase for all additional words.
- Never put variables inside quotes `""`, unless you actually want to print the name of the variable rather than its value.

#Summary

- A **variable** is a name for a memory location where you can store a value that can change or vary.
- A **variable declaration** indicates the type and name of the variable.
- Use the assignment operator `=` to assign a value to the variable. You must **initialize** a variable before using it as an expression.
- **Data types** can be categorized as either primitive type (like `int`) or reference type (like `String`).
- The three primitive most-used data types used in this course are **int** (integer numbers), **double** (decimal numbers), and **boolean** (true or false).
- Each variable has associated memory that is used to hold its value.
- The memory associated with a variable of a primitive type holds an actual primitive value.
- When a variable is declared `final`, its value cannot be changed once it is initialized.

#Extra Resources

For more information about naming conventions, check out this blog entry. It may have some language you don't quite understand yet, but it's good to get used to reading difficult things and figuring out what information you need right now and what can wait until later.

#Daily Scrum: Successes and Struggles

Each day, at 4PM EST, you're going to meet as a team and discuss where you're at. These are your Daily Scrum meetings where you'll be discussing where you're doing well, and where you need support. These meetings are only 15 minutes long–so do your best to come prepared. Think ahead about how you're doing, and be ready to report it to your Lead.

If you are the Team Lead, be prepared to gather your team and start the conversation. Responding to the Daily Scrum Form is your responsibility. Keep the conversation flowing and try to gather all the specifics you can. The feedback received by the Team Lead must be recorded in the Daily Scrum page for the day. **We'll be expecting your reports by 4:30 PM. Please note that the form automatically timestamps your response**.

Once we have your feedback, your instructors will use this information to inform the topics for our Morning Sessions, breakout rooms, and other discussions that happen throughout the week. The more we know–both on the good and the bad ends of the spectrum–the more we can maneuver to ensure that everyone achieves their goals.

Let's get into some specifics to try and make the task easier for everyone:

#Successes

It's easy to focus on what's giving you trouble, but being a dev is also about figuring out where you're doing well and keeping things in perspective. Imposter syndrome is real, so make sure you don't dwell on what's going wrong for too long. Start your Scrum meetings with a discussion of your **successes**.

The following questions are a good jumping off place for the conversation, but the meetings will go much more smoothly if everyone comes prepared with a little something to talk about. Feel free to ask yourself these questions before showing up:

1. What are you enjoying in the learning process this week so far, or just from today?
2. What's sinking in well? Are there any topics that you feel like you're growing to understand?
3. What's better today than it was yesterday?
4. What are you excited about from the lessons? Are there concepts/tools/technologies that you're hyped for?

InfoWarningTip

Remember that these meetings are only **15 minutes long**, so Leads should feel free not to ask every question here. All participants should come prepared to offer something to the group, then move on. This isn't your only opportunity to communicate–you have Slack and can talk to each other whenever you like. Try to keep this meeting dedicated to its specified goals.

#Struggles

Once you've hashed out a few positives for the day, talk about what's got you down. Here are some constructive ways to contextualize what's going on:

1. Some topics are going to make more sense than others. What do you kinda understand that isn't fully clicking with you? What could help?
2. Some stuff can be downright confusing. What are you totally lost on?
3. When we move onto larger projects, it's easy to get lost. What parts of your projects/assignments do you need help with ASAP to keep the ball rolling?

4. Where can your team help you out, and where do you need help from your instructors?

InfoWarningTip

If the Team Lead has trouble facilitating this conversation while taking notes, make sure to ask if anyone can type everything up for you. Being on a team means asking for help when you need it, so don't think that Team Leader means Team Superhuman: work with your people!

#Team Lead Responsibilities

During the Scrum meetings, it's important the Team Lead keep the group on task and moving. Ask questions to guide the session, and make sure to dig into issues if you feel like your team is being too vague. Get help where you need it, and plan on providing a short paragraph/list of bullet points to help guide the Academy forward. **Our biggest request here is that you offer us specifics**. There's a difference between saying that your team is "struggling with variables" and that they're struggling with "variable reassignment". Ask follow up questions, pursue the root causes, and give us what we need to make sessions that will bring everyone up. Feel free to tell us what's helping, too. Teams should be aiding each other where possible, and you can even tell us about outside resources that you've found that have put you on the path to success.

#Example

Use the following example as a starting point. If you have more to report than what you see below, if you want to get into more details, or want to call out specific lessons/instructors for helping you on your journey, go ahead. We all grow better when we leverage each others' strengths to iterate and continue improving.

Successes:

- Everyone feels good about variable declarations.
- Variable reassignment is making more sense after morning sessions with Sweytha.
- Hyped to finish the Challenge activity from yesterday now that we understand X better.

Struggles:

- Variable types are still a little fuzzy. Don't understand what "static" means.
- Why do we always have to write static void main…?
- Could use an example for X.

InfoWarningTip

Take brief notes during the meeting, then write them up into a short and specific bullet point or series of sentences. We'll be watching out for your feedback, so make sure you get it in **on time: 4:30 or earlier.**

#Quiz Time

For each of the questions on this page, follow the instructions and provide the best answer. We won't often be doing big quizzes like this, but it felt right for the first day–to give you a chance to see what you know and to inspire you to ask your instructors some questions. Give them a shot, and keep track of how you're feeling about the topics we've discussed so far.

#Variable Types

#Coding Challenge:

In the following IDE, add a print statement to concatenate the string literal "Favorite color is " with the value stored in the `color` variable.

InfoWarningTip

Although much of the world spells "favorite" and "color" as "favourite" and "colour", there will be times throughout the bootcamp where you'll need to use the American spellings of these words. Get used to seeing them, even in cases like these where they aren't strictly necessary.

#Coding Challenge:

This is an example of something called *assignment dyslexia*, when the coder has put the value on the left and the declaration on the right side of the equals sign. Try to fix the following code to compile and run.

#Debugging Challenge:

Debug the following code. Can you find all the bugs and get the code to run?

InfoWarningTip

Hint: if clicking the Run button doesn't successfully run your code, it definitely won't pass the automated tests we've set up. Follow the error messages and see if you can get it all workin

#Challenge Activities

Outside of the Self-Led and Instructor-Led folders each day, there will usually be a Challenge activity. These activities come with some special rules and regulations, so let's go over the details:

1. Challenge activities are optional: if you don't have the time or energy to do them, it's 100% OK to skip them. If you do have time, please give them a shot. They're good practice.

2. Challenge activities are designed to put you through your paces with something you've recently learned. Sometimes, they'll put together a few lessons you've learned into one larger task. Other times, they'll explore a single concept more deeply.
3. No matter what, Challenge activities are something you should be able to do based on the knowledge you have at that moment. You're welcome to Google parts of the question, or find new ways to do achieve the goals we've set for you, but they should be doable based on what you've learned directly from us in your Self-Led and Instructor-Led lessons.
4. Challenge activities will be worth either 5 or 10 points. We will tell you which at the top of each activity, so you can decide whether or not you want to do the work. Note that these points will be given as a kind of "extra credit" beyond the day's normally available points.
5. Challenge activities will usually be graded by midnight of the day **after** they were assigned. Challenges submitted after this cutoff will not be given credit, as the "challenging" aspect of the activity will generally be made less challenging with each new day of learning that you do. Therefore, a Week 1 challenge done in Week 10 will essentially be a walk in the park, and unworthy of the credit you'd receive. If the activity became available on a Tuesday, you have until midnight Wednesday to get the task done.
6. Partial credit will generally not be given for Challenge activities. If you did your best and would like partial credit, we ask that you do a couple things first:
    1. Talk to your team and see if you can get over the part of the task that's tripping you up. The more you work with others, the more perspectives you'll get on solving a problem, so seek out your teammates first. It's part of being a dev, so get used to it.
    2. After you've spoken with your team, if you're still stuck, you can ask your instructors if they can review your work. At this point, they may refer you to a Morning Session, or offer another solution that fits their schedule. It's up to them (and what they have going on), if they can dig in and review your work in a timely manner, so please be patient with them.
    3. At their discretion, the reviewer will then give you points commensurate with the effort and time they believe you've put into the work, as well as how much of the brief you've completed. Their decision will be final, though you can ask for feedback. If they have time, they will walk you through their thoughts. Be aware that poor formatting, unintelligible code, improper usage of coding/naming conventions, and unaddressed edge cases are all common reasons for losing points. If you want to maximize your potential, stick to the guides we've given you in your lessons.
7. Lastly, be aware that Challenge activities are governed by Academy's standard procedures regarding **plagiarism**. Copying/pasting code without understanding how it works will not be tolerated, nor will the usage of tools that complete tasks for you. We advise that you not risk your position at Academy by being dishonest, but we also want to remind you that putting in the work, learning something, and getting better as a developer are your main goals here. There will always be code out there that you can copy and paste to get something menial done–you're here to learn how to put the pieces together, think for yourself, and grow. If you're not willing to do that, you will come to regret your decision sooner or later, especially when the problems you're expected to solve grow in complexity beyond what your Google skills can cover.