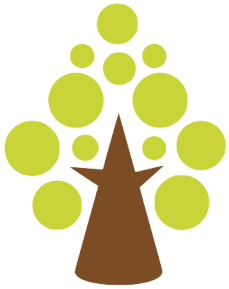
Beginner's Computer Programming Guide (Language-Agnostic)

Presented by:



Snell Scholarship Foundation

Table of Contents

Preface

Firstly, I want to say **thank you** for supporting the SSF by giving this guide a read.

For some short history, SSF was founded in 2021, and has been dedicated since then to give underrepresented people a fair chance at competing in the tech world. The Snell Scholarship Foundation's mission is to "increase retention of colored children studying STEM through scholarships and mentorship". Two of the main pain points in carrying out a mission like this, is (1) getting students interested in tech to begin with and (2) getting the interested students to stay in tech, even when it gets rough (because trust me, it *will* get rough).

As Founder of SSF, I've continually explored new avenues to increasing retention outside of our existing methods like offering scholarships and mentorship. The key piece that's been missing from this puzzle is finding a way to get more people interested in tech, which will hopefully lead to more tech majors, and subsequently more people pursuing professional roles in tech.

We've realized, this whole plan starts with education and awareness, and letting people know that if I can do it, so can you. Coding can definitely be a nightmare, but it doesn't have to be. Just like anything else in life, preparation, study, and confidence can take us a long way. The purpose of this book, document, whatever we want to call it, will be to plant those 3 very seeds. This guide will provide readers with an engagement-based crash course that steps through all of the fundamentals of programming to paint a more vivid picture of what coding is.

We won't go into too much detail about using these constructs in a practical sense, like how to build a website or an app, but we'll definitely go over the skills needed to do those things.

That said, let's lock in and get ready for this journey to become [better] programmers together!

Donnell Debnam Jr., Founder
Snell Scholarship Foundation, Inc.
Email: snellscholars@gmail.com
Official Website: snellscholars.org

I. Introduction

As computer programmers, coding is what we use to communicate with our machines to give them a set of instructions to follow. Machines themselves rely on us to program them and in some cases (like in AI), teach them how to "think". When we think of an app or even a website, all it is is a set of instructions that someone gave to a machine to say, *"Hey, show a screen that looks like this, and when a user taps on this, do this, but if this other condition is also true, do this other thing instead"*... or, at least it normally goes something like that.

Programming comes in many different forms and has a bunch of different purposes, but all forms of it use the same basis and set of fundamental practices. The idea is that the code we write as engineers is just a ton of data being consolidated and operated on, and then typically gets shown to a user in some fancy way called a UI, or user interface.

Throughout this guide, we will build a foundation of how to write simple programs using all of the low-level programming constructs to give us the tools to build anything... well, almost anything, lol.

P.S: As a small reminder, this is a language-agnostic guide, so we will not focus on any one specific programming language; the tools we'll acquire via this book will be applicable to all modern languages!

II. Fundamentals

To get started with programming in general, you'll want to be firm upon your fundamental constructs like using variables, data types, functions, etc. These are essentially the "building blocks" that you'll use to create things like apps, local programs, scripts, or even data models with Python or R.

Let's break down each of the core fundamentals:

Variables

Variables are the simplest form of programming and are used in just about every programming language known to us. **A variable is essentially just a piece of data.** This data can be a number, a word, anything. In programming though, we name these variables and assign them a value so that anytime we need the values in the future, we can reference the names that we assigned to then get the values:

```
var name = "John Doe"
var age = 21
```

In the above segment, we create 2 variables: one called `name` which holds a value of `"John Doe"` and one called `age` which holds a numeric value of `21`. Now, any time we want either of these values, we could just reference them by the name we assigned:

```
print (name) // prints “Jonn Doe”
print(age) // prints 21
```

The segment above is printing the values that are currently stored in the two variables we created. When we say “print”, we mean that we’re sending some text or other content to our machine’s screen. These results are typically sent to the same place where the code is run from, either an IDE or from the machine’s command line.

Data Types

Building off of the last section, we’ll focus now on data types. **These are specific types for the variables we create** to tell the machine how to understand the variables and their values.

Using the former example, we can update our code to explicitly state the type of the variable so that (1) the machine has an easier time understanding the code and (2) other human readers of the code understand exactly what type each variable is:

```
var name: String = “John Doe”
var age: Int = 21
```

In the above segment we have altered the 2 variables to explicitly declare a data type; `name` is of type `String` and `age` is of type `Int`, which is short for integer, which represents a number.

Here is a breakdown of the main data types that we have in programming:

Data Type	Explanation	Example(s)
Integer	A numeric value without a decimal; any whole number that is positive, negative or zero	`0`, `2`, `33`, `-199`
Float	All integer numbers that exist to infinity, plus all of their fractions and decimals	`1.26`, `-7.8`
Double	A more precise float (double = 64 bits, float = 32 bits)	`1.26735125`
Char / Character	A single alphanumeric value; any letter, number or symbol that can be typed	`‘A’`, `‘g’`, `‘3’`, `‘/’`
String	A list of characters of any length; can be null (empty), just one character, or many characters	`“Hello”`, `“ ”`
Boolean	True or false value, typically used as a flag for binary relationships (i.e. on / off, yes / no)	`true`, `false`

More on data types here.

Functions

Functions are where programming starts to get fun and creative. A function (AKA a "method") is essentially an *organized* set of instructions for the machine to follow. Normally a function is created to do **one specific job** and should typically be as concise as possible.

We don't always necessarily *need* functions, as we could just write out each line of code within it by hand each time we need it, but functions are named just like variables, so that whenever we want to reference those specific lines of code again, we can reference the function by name, just like variables.

Check out this example. Let's say we have the same variables, `name` and `age`, and we want to print out the values. Then, we change the values assigned to those variables, and we want to print the data again:

```
// Create some variables and assign them values
var name: String = "John Doe"
var age: Int = 21

// Print the current values
print (name) // prints "Jonn Doe"
print(age) // prints 21

// Change the values
// Notice when we change the values, we don't need to use keywords "var"
// and add a data type because the variables already exist!
name = "Donald Glover"
age = 39

// Print the current values
print(name) // prints "Donald Glover"
print(age) // prints 39
```

In the above segment, peep how we changed the values and printed all of the data again. Here, we didn't need a function because we only have 2 variables of data which makes printing them individually pretty easy. Imagine we had 50 variables though and we wanted to change some values and print all of them again. Would we really want to write 50 print statements again and again?

The answer is no, lol. We can instead create a "function" which prints all of the values anytime we call the function:

```
// Create some variables and assign them values
var name: String = “John Doe”
var age: Int = 21

// Print the current values
print (name) // prints “Jonn Doe”
print(age) // prints 21

// Change/override the values
// Notice when we change the values, we don’t need to use keywords “var”
// and add a data type because the variables already exist!
name = “Donald Glover”
age = 39

// A function called “printAllData”
fun printAllData() {
   print(name) // prints the current value of “name”
   print(age) // prints the current value of “age”
}

// Print the current values
printAllData() // this is how we call the function
printAllData() // this is us calling the function again
...
printAllData() // this is us calling the function, yet again
```

Things to note here:

- We created a function named `printAllData()` which can print the value of `name` and `age`
- We called the function multiple times to illustrate how easy it is to repeat all of the instructions of the function without needing to write out everything within it

Functions become **super** important in larger scale programs because we get to define the logic once, and then use it as many times as we need throughout different apps and files.

I want to give another example to *really* show how valuable functions are. They aren’t *just* routines; they’re also constructs that help us to better conceptualize logic in our code, which becomes huge in the debugging realm. For example, let’s say we have a program that has a bunch of computations in it, like mathematical computations. Somewhere in the code we see the following:

```
if (( (y2 - y1) / (x2 - x1)) > 0   ) {  … }
```

If we're paying close enough attention, we might notice that that's actually the *slope formula*, but this probably wouldn't be that obvious to everyone reading the code, especially if there are a bunch of other hidden formulas in the code too. Functions in this case can be handy because they are named by the user, typically in a way that makes its purpose obvious to people who might read or use this code.

Instead of having that code how it is in the segment, we could rewrite it using a *function* like so:

```
// Function to find the slope of a line.
fun slope() {
   return (y2 - y1) / (x2 - x1);
}
```

So that way, anytime we want to use the slope formula, we can just call that function where needed, and it will *return* a specific value:

```
if (slope() > 0   ) {  … }
```

Now reading that in English, we would have something like, *"If the slope is greater than 0, then …"* which makes the code a lot easier to follow and understand.

Oh – and while we're here, we might as well talk about function parameters too 🤷🏾.

Parameters

Parameters are constructs within the function paradigm that help us make functions more dynamic and reusable. Let's say we have the following code:

```
// Some variables we'll use for computing
var x1 = 10
var x2 = 4

var y1 = 7
var y2 =  3

// Function to find the slope of a line.
fun slope() {
   return (y2 - y1) / (x2 - x1);
}
```

This is a valid segment of code, in which whenever we call the `slope()` function, it would use the same values for `x1`, `x2`, `y1`, and `y2`. This isn't necessarily an issue, but it confines this function to only being able to find the slope of those specific variables, unless the values are of course manually changed at some point and we call the function again.

To improve this, we can tell this function to accept *parameters*, or **inputs to the function**. When we add parameters to a function, the function itself basically gets some temporary input values to use throughout the course of the function to do computation, instead of relying on some existing values. We could rewrite the slope function like so:

```
// Function to find the slope of a line.
fun slope(x1, x2, y1, y2) {
   return (y2 - y1) / (x2 - x1);
}
```

And what this does is, (1) uses the input values to do the computation, (2) returns the result of calculating the slope, (3) then essentially forgets about those temporary input values, because they were only inputs to the function.

So to really drive this idea home, let's think of how we could change the way we use this function, now that it uses parameters:

```
slope(10, 4, 7, 3) // returns 0.6667
slope(2, 5, 6, 9) // returns 1.0
```

Things to note here:

- Instead of needing variables with assigned values for the function to use, we can just pass numeric values directly to the function as parameters, and those values will be used for the computation
- We aren't forced to use variables or direct values as parameters; we have the option to use either or, as long as (1) the number of inputs matches and (2) the data type of the values match
    - Since we declared our function to take in 4 inputs (`x1`, `x2`, `y1`, and `y2`), whenever we call this specific function, we must always provide 4 values as inputs
    - Each input is ordered in the sense that the first provided input will be used as `x1`, the second will be `x2`, the third will be `y1`, and the last will be `y2`

More on functions with parameters here.

Classes / Objects

On to classes. Classes are **templates, consisting of variables and functions, which help us to organize the code** we write and also group related things together.

For example, the below segment is how we could define a simple class, `Person`, which has two properties: a `name` and an `age`. It also has two functions: `getName()` and `getAge()` which we can use to get the values of those variables:

```
class Person {

      /** Member variables */

   var name: String
   var age: Int

      /** Member functions */

   // Returns the Person's name
   fun getName(): String { return name }

   // Returns the Person's age
   fun getAge(): Int { return age }
}
```

Notice how logically, we know a person has properties like a name and an age, that's why our approach to programming such a class follows the same pattern. Typically classes are very idiomatic as well in the sense that the variables and functions within a class should match the theme or purpose of the class itself.

Things to note here:

- Typically, classes have two main parts: (1) member/class variables and (2) member/class functions; we use the term "member" to indicate that these things are part of some larger construct, i.e. the class
- Classes get much trickier, but we'll go into detail about how in the Object Oriented Programming (OOP) section

More on classes here.

III. Types of Languages

There are two primary types of languages, with respect to the data we're handling. These two types are *static* and *dynamic* languages.

The differentiating factor in these two types is that static languages need to know the type of the variable when all of the code is being compiled, whereas dynamic languages typically know how to *infer* the data type, or will throw some sort of error if it can't.

Static languages often use a "compiler" to convert all of the code we write to machine language, and then let the machine carry out the instructions we gave it. This compiler is extremely sensitive, and won't be able to properly compile our code if for example, data types aren't used correctly.

Let's talk more about these types...

Static

"Static" languages are languages that require the programmer to specify the data type of the variables and functions they create. Some examples of these languages are **C++ and Java**. The compiler will not understand the code you've written in those languages if you do not specify a type for variables and function return types.

Dynamic

On the other hand, we have dynamic languages. The most popular of this family is **Python.** These languages allow us to keep our code far more concise and generic, and often use an "interpreter" to understand the code.

We can define a variable, `myVariable` in **Python 3** like so:

```
myVariable = 3
```

Which would be interpreted by Python as an integer because it is a numeric value within the int range. However, we never explicitly stated that this is an integer. This is because Python interprets the code, rather than compiling exactly what you told it, like we see in languages like Java or C++. So, this means we could later change that value to something *completely* different:

```
// Changing the variable's value
myVariable = "New value" // this value is now a String, not an Int
```

This is completely legal in Python; any variable can be any value at any time. However in Java or any static-typed language, once we state the data type of said variable, it can never hold a value of another type because it would not be understood by the compiler.

More on static and dynamic languages here.

IV. Object-Oriented Programming (OOP)

OOP is a very popular style of programming, which leverages "objects". Objects are simply an idiomatic construct of programming in which the classes we create are operated on in what we call "instances".

Creating an instance

I know the beginning of this section was probably a bit confusing, so let's look at an example. Imagine we have the same `Person` class from the "Classes / Objects" section. In order to use that class, we would plug that logic into some other class (eg. `Test`) like so:

P.S: We'll be using Java for this section just to get an understanding of how these things work, but this concept applies to all object-oriented languages and should only vary in syntax.

`Test.java`:

```
class Test {

   // Create an "instance" of the [Person] class
   Person me = new Person() // this creates the object instance

   ...
}
```

Here, we created a new object called `me`, which has a type of `Person`; formerly all of the types we used for variables were primitive, but the classes we create can be types as well! The same logic would apply if our original class was called `Vehicle` and we wanted to create instances of that; we'd probably create an object named `truck`, `car`, `motorcycle`, etc., because they would be logically considered instances of a vehicle.

Manipulating an instance

In OOP, we operate on instances by (1) overriding the member variables directly and/or (2) calling the member functions. For example the `Person` class has two variables, right: `name` and `age`. It also has two functions: `getName()` and `getAge()`. Follow me through this quick example.

Let's say we want to change this instance's `name` and `age` to be `"John Doe"` and `21`; we could do so like this:

`Test.java`:

```
class Test {

   // Create an "instance" of the [Person] class
   Person me = new Person() // this creates the object instance

   // Update the name and age
   me.name = "John Doe" // this changes the name
   me.age = 21 // this changes the age
}
```

Note that the work we just did only uses the variables of the class, not any of the functions. If we want to call the functions, let's say to print these values that we just set, we could do so like so:

`Test.java`:

```
class Test {

   // Create an "instance" of the [Person] class
   Person me = new Person()

   // Update the name and age
   me.name = "John Doe"
   me.age = 21

   // Get and print the values using member functions
   var myName = me.getName() // returns "John Doe"
   var myAge = me.getAge() // returns 21

   print("My name is " + myName)
   print("My age is " + myAge)
}
```

Things to note here:

- Each variable and function we used was a *member* of the `me` instance, which is an instance of the `Person` class, so it has all of the variables and functions of that class

V. Data Structures

Data structures are **what we use to store data**. Aside from the variables we've discussed which are used to store *single* pieces of data, a structure is able to store multiple variables as a "collection". Different data structures may come in handy for different jobs, but typically you'll want to have a firm understanding of these key structures below:

Array

Arrays are considered the most basic and fundamental of all data structures because they are a **linear collection of values that are [usually] of the same type**.

Let's say a professor is grading final exams and wants to run some analyses on the test scores they gathered (i.e. finding the mean or median score). They would probably want to first get all of these values together in some list like so:

```
// An array of test scores
var scores = [99, 68, 61, 88, 92, 80, 77] // fill the array with values
```

Things to note here:

- All of the values in the array are of the same type, which in this case we can probably assume to be integers based on their values

Length / Size

Arrays also have special properties that can tell us more about the array once it's been created. For example, every array has a `length` property that when invoked, returns the length of that specific array.

Using our earlier example with the test scores, we could do something like the following to find out the size of this specific array:

```
var arraySize = scores.length // returns 7
```

Indexing

Arrays also have indices, which are essentially just **positions in the array**. At each position of an array, there will either be a value (if we have specified a value) or a default value like `0` or `null` (depending on the programming language and data type that the array is storing).

Indices become super important when iterating through an array, because it allows us to directly access elements of the array, like so:

```
var scores = [99, 68, 61, 88, 92, 80, 77]

scores[0] // returns 99
scores[1] // returns 68
...
scores[6] // returns 77
```

The bracket `[]` syntax is appended to the end of the array's name which indicates we want to get the value of that array at a specific index, **starting with 0**.

When we draw these things out, normally we make **one, contiguous block**, divided into sections representing each index. This is to emphasize that an array is one space in the machine's memory, but within itself, it holds multiple values:

scores							
Value	99	68	61	88	92	80	77
Index	0	1	2	3	4	5	6

More about arrays here.

List / ArrayList

Now moving on to Lists or ArrayLists. These are extremely similar to arrays, except in languages like Python we get the option of storing values of all different types, together. Without getting too much into language-specific detail, another advantage of the ArrayList compared to the array is that **arrays are often of a fixed length**, whereas an **ArrayList is dynamic and can be of any size** (as long as the machine has enough memory to store the values).

BTW, the fact that a List can store different types is exclusive to *some* languages; this is **not the case in languages like Java or C++**. In the strictly-typed languages, we must tell the compiler what type all of our elements will be, and if we do not honor that, the compiler won't be able to understand the code.

To connect the dots from some of the previous concepts we've covered, an ArrayList is just a class that has some variables and functions that programmers use to hold their data in one place.

In a nutshell, the class for an ArrayList looks something like:

```
class ArrayList {

      /** Member variables */

   var size: Int
   ...

      /** Member functions */

   // Adds the input element to the ArrayList
   fun add(...) { ... }

   // Removes the input element from the ArrayList
   fun remove(..) { ... }

   // Returns the size of the ArrayList
   fun size(): Int { return size }

   ...
}
```

In which there are member variables and functions that can tell us the size, what elements are in the structure, if the structure is empty, and much more. See the full class implementation.

Let's try creating a `List/ArrayList` in **Java 8**:

`Test.java`:

```
class Test {

   // Create a List / ArrayList of integers
   List<Integer> list = new ArrayList<Integer>();

   // Add our values to the list
   list.add(99);
   list.add(80);
   ...
   list.add(77);
}
```

Things to note here:

- The `List` declares that all of its elements will be of a specific type (`Integer`)
- A Java ArrayList must add its elements one by one (unless you do something fancy to add multiple elements at once which is beyond the scope of this lesson!)

Set / HashSet

A Set is much much like a List or array, except **it does not store duplicates**. Using the example of the professor and the test scores, we know this would be a bad time to use a `Set`, because we could definitely expect duplicate test scores for students in a class. The chances of 2 students getting the same final exam score is pretty high, so we would be much safer using a List in this case, or at least something that allows duplicates.

However, there are definitely cases where a `Set` would be more appropriate, such as dealing with **unique** objects or values.

Defining a set in **Python 3**:

```
mySet = set([1, 2, 2])
print(mySet) // prints {1, 2} because the duplicate 2 is disregarded
```

More about sets here.

Dictionary / HashMap

Dictionaries are my personal favorite. They are structures made specifically for pairing. The way it pairs is using a key-value system, where **the keys are unique** and all have corresponding values (that don't have to be unique). This would be a perfect structure to use for students' test scores, perhaps if we use the student's name as a key and their final exam score as a value.

In theory, a mapping of test scores would look something like this, where each `String` (student name) has its own associated value (`Integer` test score):

```
{
   "John" : 99,
   "Hether" : 80,
   "Matthew" : 61,
   ...
   "Chris" : 77,
}
```

In **Python 3** we could literally just copy the theoretical code above and assign it to a variable:

```
scoreMap = {
   “John” : 99,
   “Hether” : 80,
   “Matthew” : 61,
   …
   “Chris” : 77,
}
```

In **Java 8**:

```
HashMap<String, Integer> scoreMap = new HashMap<String, Integer>();

scoreMap.put(“John”, 99);
scoreMap.put(“Hether”, 80);
…
scoreMap.put(“Chris”, 77);
```

Things to note here:

- Just like `ArrayList` in Java, we must add the elements one by one 😠 (again, unless we use some fancy logic which is beyond the scope of this lesson)

When it comes time to *retrieve* elements from a dictionary or HashMap, **we always need the key**. Much like how we tell an array which index to give us a value for, we use the same approach here, instead we pass the key.

Here’s how we can retrieve elements from the structure using the key in **Python**:

```
scoreMap[“John”] // returns 99
scoreMap[“Hether”] // returns 80
…
scoreMap[“Chris”] // returns 77
scoreMap[“Unknown”] // throws a KeyError
```

And in **Java**:

```
scoreMap.get("John"); // returns 99
scoreMap.get("Hether"); // returns 80
...
scoreMap.get("Chris"); // returns 77
scoreMap.get("Unknown"); // throws an Exception
```

Things to note here:

- When an invalid key (i.e. a key that was never added to the dictionary) is passed as a parameter, the interpreter or compiler will throw an error and the code will not run

More about Dictionaries / HashMaps here.

VI. Algorithms

Here we'll touch on some of the most common types of algorithms. An algorithm isn't necessarily something super complex that requires a bunch of math or headaches. An algorithm can be simple, as it's really just a procedure used for solving a problem or performing a computation [1].

Conditionals

Conditional statements are our way to tell the machine to do one thing if a certain condition is true, or to perhaps do something else if it is false.

This is generally done using the `if` keyword followed by a condition (which is normally wrapped in parentheses). An example of a valid if-statement would be:

```
if (300 > 200) { doSomething() }
```

In which we know 300 is greater than 200, so we would expect the function `doSomething()` to be called. If this ever evaluates to `false` though like we'll see in later examples, the logic within the parentheses would never execute because it is conditional and only evalues when the specified condition is `true`.

Let's say the professor from our earlier example is determining passing or failure statuses for students based on the median test score. If the median is below 70, the professor will pass all students. If the median is above 70, everyone passes:

```
// An array or list of scores
var scores = [99, 68, 61, 88, 92, 80, 77]

// Let's call some function that gets the median of all of the scores we
// have.
var median = getMedian(scores)

// Let's check if the median we got is greater than or equal to 70.
if (median >= 70) {
   // everyone passes!
} else {
   // everyone fails!
}
```

Things to note here:

- We have an `if-else` clause; the `if`-statement is where we add our condition, and in case that is ever `false`, we have an `else` clause that will be executed as a default case
- Only one part of the conditional statement will actually execute; either the median is greater than or equal to 70, or it is not
    - Based on what that condition evaluates to, the code will only hit one of the paths

Looping

Another common type of algorithm you need to know is **looping**. This is essentially how we iterate over some collection of data which allows us to do hundreds of other things, like run analyses on it. There are a few different types of loops, so we'll go through each type below.

1. For-loop

The for-loop is the most fundamental loop we have; it exists in pretty much every programming language. Loops are used for iterating over some collection of data (i.e. a data structure) to view or manipulate its elements.

Taking the same professor example with the test scores, let's say we wanted to iterate over the array of scores and check if there is an `88` in the list. In order to do so, we would need to start from the beginning of the list and check each element (in order) to see if it is an `88`:

```
// This is how we create a for-loop
for (int i = 0; i < scores.length; i++) {

   // Check the element in the array at index i; if
   // it equals 88, do something below
   if (scores[i] == 88) {

      // if we found an 88, print that we found an 88.
      print("we found an 88!")

   } // end if-statement

} // end for-loop
```

The for-loop has three (3) main parts in its header: the index, a condition for when to stop the loop, and how we want to handle each iteration. Let me explain...

1a. The index:

```
for (int i = 0, ..., ...) { ... }
```

As discussed earlier, the index of the array is important because it is a numeric value that tells us where we are in traversing this array. For example, if you read the list of `[99, 68, 61, 88, 92, 80, 77]` from left to right, `99` would be the first element, which makes it index `0`. `68` is the second element, so its index is `1`. This continues until the end of the list. So, `i` is what we use in our loop to keep track of whether we're currently at index `0`, index `1`, ..., index `6`.

1b. A condition for when to stop:

```
for (..., i < scores.length, ...) { ... }
```

Here, we have `i < scores.length`. With for-loops, the loop will iterate and continue until this condition becomes `false`. So, in our case, the condition is basically saying, *"Let's continue iterating over this list **only while** the value of `i` is less than that of scores.length"*.

1c. How to handle the end of each iteration:

```
for (..., ..., i++) { ... }
```

Here, the `i++` just means, after every iteration, *"do `i++`"*. `i++` in particular means, *"add 1 to `i`"*. So if `i` is currently 0, after the iteration it would bump to 1, then 2, so on and so forth **until** we reach the number of iterations we set for the loop in part 1b.

2. For-each loop

Don't worry, loops get easier from here. A for-each loop typically does the same thing as a for-loop, except it is simplified in the sense that you don't have to tell it what to do with all of the things we discussed in the header. All a for-each loop needs to know is (1) what collection of data it should be iterating through, (2) the type of data that collection consists of, and (3), what to call each element, since it won't be indexed using `i` like a for-loop. Let's examine that...

Using the same example with the scores, we could use a for-each to iterate over the same scores, but in a friendlier way:

```
var scores = [99, 68, 61, 88, 92, 80, 77]

// This is how we create a for-each loop
for (int score : scores) {

   // Check if the current element is an 88.
   if (score == 88) {

      // if we found an 88, print that we found an 88.
      print("We found an 88!")

   } // end if-statement

} // end for-each loop
```

The header for a for-each loop normally reads just like plain English; the loop we implemented would read something like, *"For each integer called score in scores"*, meaning `scores` is the collection of values and each element within the collection will be called `score` when we get to its specific position.

3. While-loop

If you made it this far, you can catch a breather with while loops. A while loop is a type of loop that continues while the specified condition is `true`. You have to be careful with these though because if your condition never ends up being `false`, it will never stop and typically crashes the code.

Let's use a while loop to find out if there's an `88` in the list:

```
var count: Int = 0
var found: Boolean = false

while (found != true) { // “while found is not true”

   // Check if the current element is an 88.
   if (scores[count] == 88) {

      // if we found an 88, print that we found an 88.
      print(“We found an 88!”)
      found = true // update the value since we found the 88.

   } // end if-statement

   // Otherwise, we haven’t found the 88, so let’s keep looking
   else {
      count += 1 // increment the counter
   } // end else

} // end while-loop
```

Things to note here:

- We ended up having to do more work than we wanted to when using the while-loop because we needed to (1) keep track of the index to check the value at each index and (2) have some way of breaking out of the loop; using a while loop is not efficient for this job
- In the event that there was no `88` in the array, this while-loop would continue on forever, and then crash the machine because the condition we set never became `true`

Efficiency

“Algorithm Efficiency” is a term you’ll hear a bunch about once data structures start being introduced and used. The idea is that every data structure was thoughtfully created with a unique purpose and functionality that gives it an edge over its competitors. Based on the data we’re storing and planning to access, we must choose a structure that suits our specific needs, making it the most efficient for the job.

For example, technically we *could* use a fork to eat cereal, there’s nothing stopping us, but we know it wouldn’t be the most efficient tool for this specific job. In computing, we measure how efficient an algorithm is using something called “Big-O Notation”.

This notation is a mathematical notation that helps us to understand one important concept: *for a specific input and algorithm, how fast can a machine (on average) compute an output?*

The easiest way to think about "efficiency" is probably by thinking about how much work the machine has to do, and how long it is expected to take. If a machine has to do little to no work to solve our problem, we generally consider this a very efficient algorithm. On the contrary, if our machine needs, let's say 30 minutes to an hour to finally solve our problem, this is likely a very inefficient algorithm that needs some work.

The caveat here is we don't actually describe an algorithm's efficiency using time since time is relative and depends on a bunch of other things, so we use Big-O as a general way of describing the algorithm's workload, which tells us how many actions the machine will need to perform (at the worst case) to reach a result.

Here's a breakdown of how we classify algorithms in terms of efficiency when our input (N) is *N = 10* (10 in this case signifies our input is of length 10):

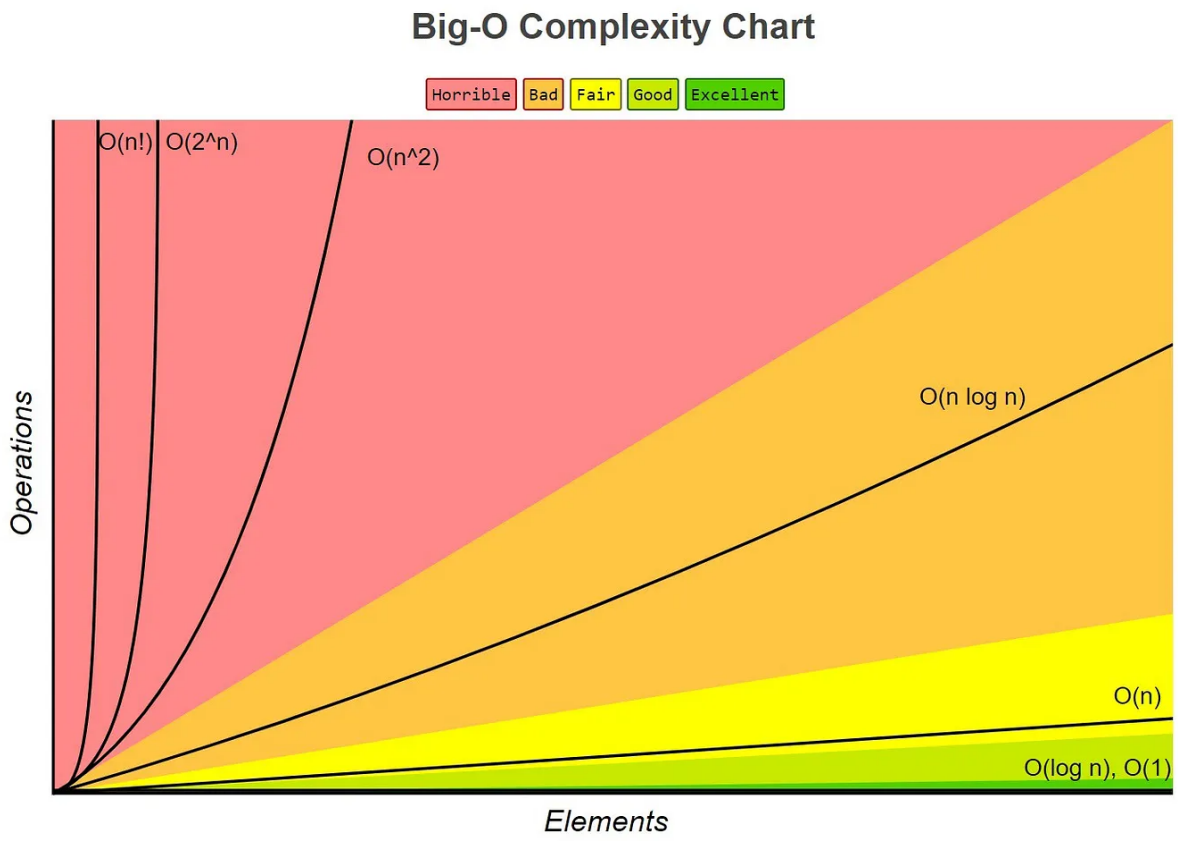
	Name	**Notation**	**Rating**	**Evaluation (# of steps)**
1.	Constant	O(1)	Excellent	O(1) = **1**
2.	Logarithmic	O(logN)	Excellent	O(logN) = log(10) = **1**
3.	Linear	O(N)	Fair	O(N) = O(10) = **10**
4.	Loglinear	O(NlogN)	Bad	O(NlogN) = 10log(10) = **10**
5.	Quadratic	$O(N^2)$	Horrible	$O(N^2) = 10^2$ = **100**
6.	Exponential	$O(2^n)$, c > 1	Pretty Horrible	$O(2^n) = 2^{10}$ = **1024**
7.	Factorial	O(n!)	Super Horrible	O(n!) = 10! = **3628800**

Things to note here:

- If our input is of 10 elements (hence *N = 10*), the best algorithms will allow us to find a result with 10 steps or better
- The values we get in the 'Evaluation' column all vary on the size of the input; there's a **direct relationship** between the input size and number of steps needed to complete the job
- Though constant, logarithmic and linear solutions are considered the best, they aren't always available options; sometimes a job *has* to be done a certain way, which warrants the need for a slower, less ideal algorithm

The below illustration [2] shows a graphical representation of what we just covered, but shows how the number of operations is **a function of** the number of elements.

Without getting too mathy, just try to focus on the colors and what they mean, and the shape of each trend line:



We can see that for algorithms like constant (O(1)) and logarithmic (O(logN)), no matter how large the 'Elements' axis grows, the 'Operations' axis always stays extremely low; this means no matter how large our input size is, the algorithm will always be super efficient.

On the other hand, let's look toward the red area. We can see, for example, $O(N^2)$ has a steep, upwards curve. This means that as the number of elements grows, the number of operations is **heavily** impacted and grows too. We can see this in the chart from earlier, that $O(N^2)$ when *N = 10* puts us at 100 operations. This isn't extremely bad for a smaller input like *N = 10*, but try to imagine if our input was *N = 100*. We'd then have $O(100^2)$, which puts us now at 10,000 operations (at the worst case).

Best vs Worst case

If I were you I'd probably be wondering why we keep saying "at the worst case". We say this to emphasize the fact that an algorithm's efficiency can technically be pretty good like O(1), but it'll also have cases where it slips severely and becomes something like O(N).

Let's look at an example with some code to make this make sense. I know you're probably tired of the scores example, but it's about to make everything make sense right here (hopefully). Let's say we have the same list of scores, right, and we have some code that uses a for-each loop to find out if there's a 99 in the list:

```
var scores = [99, 68, 61, 88, 92, 80, 77]

// A for-each loop that looks for any occurrence of 99.
for (int score : scores) {
   if (score == 99) {
      print("we found 99!")
   }
}
```

Our input size for this algorithm is the length of our array, which is currently 7 (i.e. there are 7 elements in the collection). Let's analyze that here:

	Best Case	Worst Case
Explanation	In the **best case**, 99 is the *first* element in the collection, meaning we find it right away and don't have to visit any other positions in the array to find it (1 step total)	In the **worst case**, 99 is the *last* element in the collection (or doesn't appear at all), meaning we just looked at every single element in the collection (7 steps total)
Big-O	Constant, O(1) = **1**	Linear, O(N) = **7**

Things to note here:

- The same algorithm (known as a linear search) in the worst case is O(N) where *N* is the number of input elements, but in the best case is always O(1)

More about algorithm efficiency here.

VII. Getting Started with Projects

One of the keys to preparing for a career in tech is to have as much technical experience as possible. You don't necessarily need to learn a bunch of languages and work in multiple areas, but you'll want to get proficient in at least one area and use that as a foundation.

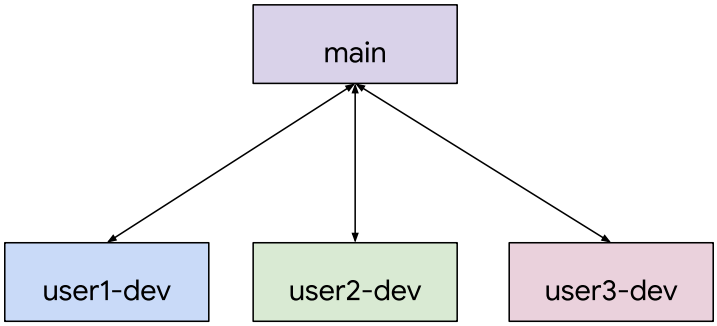
GitHub

As a beginner, you'll need to create a GitHub account, which is a free, online resource that stores all of your code. The code can be app code, website code, random code, even files like images and PDFs. The purpose though is to build a portfolio of projects that can be found and viewed in one place (recruiters typically want to see a GitHub link on your résumé).

Collaboration

GitHub is also known for its collaborative features that allow engineers to contribute to projects individually, but as part of one project. Typically each engineer will create a "branch", stemming from the source of the code (main / master branch) which serves as like a personal workspace.

Let's observe the following diagram:



Notice that the `main` branch is the central branch that all of the sub branches communicate with. The three sub branches, `user1-dev`, `user2-dev` and `user3-dev` represent three engineers who are all developing and contributing to the same project, which is stored in the main branch.

Once one engineer wants to add some new code, they'll add it first to their branch, then "push" it to the main branch. Once it gets pushed, all of the other users in their branches can "sync" or "pull" in those new changes.

More on GitHub here.

VIII. Guided Practice

Here we'll go over some quick steps to get started with actual hands-on programming. We won't be going over any specific coding exercises to do, but these steps will help us create an environment where we can work on some beginner projects and

Download an IDE [JGrasp]

An Integrated Development Environment (IDE) is what most developers write their code in, and do things like run the code and debug it. We'll download JGrasp because its interface is very simplistic and it has all the tools we need, by default, as it was originally developed for academia.

Commit to a language [Java]

Remember, in programming, we have both static and dynamic languages, which are very different and have unique advantages and disadvantages. You'll likely want to choose between **Java** and **Python** as a "first language" and start from there.

Personally, **I recommend starting with Java**.

Create a new program

1. Creating a new file

Open JGrasp and go to **File > New > Other,** and **choose your language**.

2. Saving the file

After we get a fresh, clean environment to work in, we'll want to save this file so that JGrasp knows exactly where to save all of the code on our machine. Go to **File > Save As**, and find a suitable directory on your machine to save the file in (eg. a folder titled `SSF Coding Practice`).

If coding in Java you must name your file with the `.java` extension (eg. `Main.java`)
If coding in Python, you must name your file with the `.py` extension (eg. `Main.py`)

Start Coding!

For a first program, we'll just focus on reinforcing some of the fundamental concepts we learned throughout the guide. Here's a great video we can follow along with to get a Java program going: Intro to Java using JGrasp

IX. Links & Resources

Here we have a compilation of some beginner's resources to help us get started with programming in different aspects.

General

- CS Dojo on Youtube: teaches the basics in a very beginner-friendly way
- W3Schools: web tutorials, online bootcamp, free general programming resources
- Khan Academy: free online course for into-level programming

Command Line / Terminal

- Compiling Java: tutorial on how to manipulate files and run a Java program from the terminal
- Running Python: tutorial on how to run a Python program from the terminal

App Development

- Mobile Android App: tutorial on how to create a simple Android app with Android Studio
- Mobile iOS App: tutorial on how to create a simple iOS app with XCode
- Online Game: 12-hour livestream of engineer coding an online game
- Mini Python Projects: tutorial on how to create different Python projects for beginners

Artificial Intelligence (AI)

- Image Recognition: recognizing images of numbers on Youtube with MNist dataset
- Creating a Neural Network from Scratch: tutorial on how to create a neural network in Python

Hardware Engineering / Robotics

- Python Programming for Raspberry Pi Robot: engineer writing code to program a robot using a Raspberry Pi

Cybersecurity

- Track Phone Number's Location: tutorial on how to create a Python program that tracks the location of a phone number (not precise locations, lol)

Interview-based Practice

- CodingBat.com – beginner's coding exercises; you can switch between Java and Python and there are different levels
- Leetcode.com – more advanced exercises than codingbat, can use any language you prefer

X. Works Cited

[1.] TechTarget (n.d.). *What is an Algorithm?* WhatIs.com. Retrieved May 28, 2023, from https://www.techtarget.com/whatis/definition/algorithm#:~:text=An%20algorithm%20is%20a%20procedure,throughout%20all%20areas%20of%20IT.

[2.] Huang, S. (2020, January 16). *What is Big O Notation Explained: Space and Time Complexity?* FreeCodeCamp. Retrieved May 28, 2023, from https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/