**(Work in Progress, underlined portions are topics that are reserved)**

**Introduction: For those who are new to programming or new to Java:**

* **What is Java?**
* **Variables**
  + **Primitive Data Types**
  + **Introduction to Strings**
  + **Declaring and Initializing Variables**
* **Printing to the Console**
  + **print and println()**
  + **Arithmetic Expressions and Concatenation (order of operations go here)**
  + ***printf()***
* **Control Statements** 
  + **If Statements (boolean expressions go here)**
  + **If-Else Statements**
  + **While Loop**
  + **Do-While Loop**
  + **For Loop**
  + **Variable Scope**
* **Methods**
  + **Introduction to Methods**
  + **Pass-by-Value**
  + **More information on the *return* keyword**
* **Arrays**
  + **One-dimensional arrays**
  + **Multidimensional arrays**
    - **2D Arrays**
    - ***3D Arrays***

**Medium:**

**UIL Preparation: Additional content that is tested on the UIL Exam, and for those who are looking to learn more!**

* **Number Systems**
  + **Binary**
  + **Decimal**
  + **Hex**
  + **Base Conversions**
  + **Bitwise Expressions**
* **ASCII**
* **Order of Operations**
* **Regular Expression**

Welcome to Perfect Java! This is a *free* resource for you to learn Java, one of the most popular programming languages in use today. Computer scientists are in great demand today, and we wanted to provide a gateway to allow everyone to experience what it is like to code, without any ads or paywalls! This is purely a passion project and thus, we seek no profit. Learn as much as you can!

There are a variety of use cases for this app:

* A beginner’s tutorial on the ins and outs of Java: This app gives you a very high-level understanding of Java, enough for you to understand the fundamentals of how the language works. With easy and simple explanations, we turn the mundane, tricky, and downright migraine-inducing aspects of Java into something concise and compact. Familiar with a different language? Trying to delve deeper into Java concepts? Never touched a programming language in your life? This is the perfect app for you.
* Supplementing a high school/college programming course: I’m not going to lie, this app contains a lot of information, but if you decide to use this app as a substitute for your textbook, your grade in the class will not be very high. The textbook contains low-level concepts that we don’t cover in this app and goes deeper in content (though I can say confidently that it’s much more boring than our app). However, using this app to supplement the textbook is very encouraged, since most of the lessons on here are based off of the AP Computer Science curriculum. A recommended resource for high school students taking a Java programming course, or a college student taking an introductory Java course.
* Cram material for those taking the UIL Computer Science Exam: This is a niche group, but this app was originally created as a practice tool for prospective UIL Computer Science competitors before we ambitiously decided to expand the target audience to… well… the world. However, this app is sponsored by The University of Texas at Austin, who hosts the competition annually. We try to cover all the common (and uncommon) topics the written exam tests you on. This app was created by former State competitors, so you’re in good hands.

Well, that’s all we have to say, so there’s only one thing left to do… start learning! We truly hope you enjoy this app and learn some Java out of it! It’s made with love and care.

**Primitive Data Types**

Primitive data types are the most basic data types available within the Java language. Think of them as the building blocks of Java. These data types are predefined within Java—meaning that there are a lot of very cool operations you can do with them. You will learn many of those operations in due time!

There are eight primitive data types. Tap on a data type to learn more about it.

**boolean byte char short int long float double**

**Boolean**

A boolean data type has only two options: *true* or *false*. If you are familiar with other languages, you may be used to assigning numerical values to a boolean—this is not possible in Java, the only two possible values a boolean can have is *true* or *false*.

//Declare a boolean with the value *true*

boolean isSunny = true;

//Declare a boolean with the value *false*

boolean isCloudy = false;

//What do you think the output is? Hint: It’s not an error

int num = 10;

boolean isNumGreaterThan5 = num > 5;

System.out.println(isNumGreaterThan5);

**Byte**

A byte is a whole number within the range of -128 to 127. Bytes are useful to conserve memory if you are dealing with numbers within that range.

//Declares a byte with the value *12*

byte dozen = 12;

//Declares a byte with the value -1

byte num = 1;

//Declares a byte with the value 200, but the range of a byte is only from -128 to 127. That’s out of bounds! Error!

byte num = 200; //Will not compile!

**Char**

Char represents any single character, whether it be a letter, digit, symbol or space, so long as it is singular. Chars in Java are declared with single quotes (‘ ’)

//Declares a char with the value ‘A’

char firstNameInitial = ‘A’;

//Declares a char with the value ‘5’

char onesDigit = ‘5’;

//Declares a char with the value ‘\*’

char asterisk = ‘\*’;

**Short**

A short is a whole number within the range of -32,768 to 32,767. Shorts are useful to conserve memory if you are dealing with numbers within that range. Though for beginner programmers, I wouldn’t stress too much about memory usage (am I allowed to say that?)

//Declares a short with the value 5000

short num = 5000;

//Declares a short with the value -100

short negativeNum = -100;

//Declares a short with the value 800,000. However, the range of a short is only from -32,768 to 32,767. That’s out of bounds! Error!

short num = 800000; //Will not compile!

**Int**

One of the most used data types! An int is a whole number that has a minimum value of -231 and a maximum value of 231 – 1.

//Declares an int with the value 30

int num = 30;

//Declares an int with the value -8125123

int negativeNum = -8125123;

//Declares an int with the value 3.14. Wait a second. Ints can only store whole numbers, and 3.14 is a decimal. Error!

int pi = 3.14; //Will not compile!

For those who like visualizing large numbers:

Maximum value: - 1 = 2147483647

Minimum value: - = -2147483648

**Long**

The *long* data type can store some pretty large numbers! Long has a minimum value of -263 and a maximum value of 263 – 1.

//Declares a long with the value 31415926535

long bigNumber = 31415926535;

//Declares a long with the value 314159265358

long biggerNumber = 314159265358;

//Declares a long with the value of the arithmetic expression *153\*521*

long someNumber = 153\*521; //That’s 79713, for those curious

**Float**

Float comes from the term *floating-point numbers*, which is a fancy way of talking about numbers that are fractional. Some examples are π (3.1415…), the natural number e (2.71828), and 5.39, which is a decimal I made up on the spot. Floats end with *f*, to signify that it is a floating-point number (this is just Java convention!)

Thus, floats represent numbers involving decimals.

//Declares a float with the value 5.39

long iMadeThisDecimal = 5.39f;

//Declares a float with the value 2.71828 (the natural number *e*)

float valueE = 2.71828f; //This is a constant—we’ll cover that later!

//Declares a float with the value 5. It’ll compile fine, but perhaps float isn’t the most appropriate data type to use for a whole number. Which ones would work better?

float wholeNumber = 5f;

**Double**

Double is similar to float, in that they both handle floating-point numbers (fractional numbers). Double is usually preferred to float, however, because it is more precise than floats when it comes to decimal precision.

//Declares a double with the value 123456789.987654321

double iMadeThisDecimal = 123456789.987654321;

//Declares a double with the value 2.71828 (the natural number *e*)

double valueE = 2.71828; //This is a constant—we’ll cover that later!

//Declares a double with the value 5. It’ll compile fine, but perhaps double isn’t the most appropriate data type to use for a whole number. Which ones would work better?

double wholeNumber = 5;

**Declaring and Initializing Variables**

To start programming, you are going to want to use variables. A variable is a name along with a particular data type that contains a value associated with that data type. Think of it like a container that stores some value. The cool thing about a variable is, well, that it *varies.* It can change. It’s flexible. And that’s what makes it cool.

Java is a statically typed language. And if you don’t know what that means, do not fret. Basically, Java expects a variable to be declared before we can pass in values and mess around with them. So when we create a variable, make sure it has a type and a name before we start to do anything with it. Let’s get started.

We’re going to declare an int variable, and the process is pretty simple:

int num;

In this example, we have declared an int called *num*. Now, you can name the variable pretty much whatever you want, with [some exceptions](https://mathbits.com/MathBits/Java/DataBasics/Namingrules.htm), but I heavily recommend you name it something that is easy to understand. Clarity is very important when it comes to programming.

//Let’s say you’re declaring a variable that represents your age. Which one of these is easiest to understand?

int myAge;

int qwotiqwrog;

int q;

Anyways, back to the original code:

int num;

This won’t compile—not yet! For it to compile, we need to give the variable a value. The value should correspond with the variable type. For instance, a variable of type *int* should have an integer, like 3 or -10. If you try to give an *int* a decimal (or even worse, a letter, how dreadful) Java will get angry, and give you an error when you try to compile.

So now let’s declare a variable—and give it an appropriate value. In this case, since the variable is of type *int*, any zero or nonzero whole number will do. This is known as *initializing* a variable.

int myAge;

myAge = 21;

Once you declare a variable, *you do not need to do it again*. Notice how in the second line, we do not add the *int* keyword again (doing so will actually give you an error). That is because we have already declared it in the line prior! This is cool because having to retype *int* every time we try to make a change to the variable can get very redundant.

We can declare and initialize a variable in one line:

int myAge = 21;

We can change the value of a variable as many times as we want.

//myAge is 21

int myAge = 21;

//Now myAge is 30

myAge = 30;

//Now myAge is 35

myAge = 35;

Here we add the value of two variables to a third:

int examOneScore = 90;

int examTwoScore = 95;

//combinedExamScores is the sum of examOneScore and examTwoScore

int combinedExamScores = examOneScore + examTwoScore;

//This should output *185*

System.out.println(combinedExamScores);

Declaring and initializing variables of other types work in a similar fashion. Here are some examples:

//This variable has the name *costOfApple* with a *double* data type

double costOfApple = 0.75;

//This variable has the name *isWeekday* with a *boolean* data type

boolean isWeekday = false;

//This variable has the name *letterGrade* with a *char* data type

char letterGrade = ‘A’;

**Introduction to Strings**

Hey there. Welcome to an introduction on one of the most useful data types, the *String*.

*Strings* are not too difficult to understand or use, but they can get tricky later on when we get deeper and deeper into the semantics of Java. This is a gentle introduction to what *Strings* are and how they work, perfect for the beginner coder. By the end of this lesson, you’ll have a high-level explanation on how Strings work, as well as how to declare and initialize one.

So… what exactly is a *String*?

Simply put, a *String* is a sequence of characters. Recall that a character (*char*) is a single letter, digit, symbol, or space. A *String* is just a collection of those. For example, consider the *string* “Apple, which is actually a combination of five *chars*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | p | p | l | e |

Similarly, the *String* “cat” is actually three *chars*:

|  |  |  |
| --- | --- | --- |
| c | a | t |

And the *String* “A pen” is actually five *chars*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A |  | p | e | n |

(remember, a space is also a char!)

A *String* is a data type—but it is **not** a primitive data type. Primitive data types represent the most basic data types—the building blocks of Java, so to speak. A *String* is a sequence of *chars*, so we would not consider it a basic data type, thus making it a *non-primitive data type*.

Alright, cool. So, what does this mean?

Well, for one thing, that means *Strings* work a little differently than your typical primitive data type. In Java, a String is actually an **object**, part of the *java.lang.String* class, and if you don’t know what that means—don’t worry about it. When the time comes, you’ll learn all about objects and how they work. But for now, just understand that because *Strings* are an object and are part of the *java.lang.String* class, they have a lot of cool, built-in functionality that you can mess around with. That’s beyond the scope of this lesson, but we’ll cover it in a future one.

Anyways, enough talk. Let’s declare and initialize a String:

//Java is case-sensitive! *String* and *string* are **NOT** the same!

String greetings = “Hello world!”;

System.out.println(greetings);

We’ve declared a *String* variable with the name *greetings* and initialized it with the value *“Hello world!”* Pretty easy, right? For the terminology lovers, we’ve just created what’s known as a *string literal.*

Here are some important things to remember:

* When declaring a *String* variable, remember that its first letter is capitalized. This matters because Java is case-sensitive.
* We wrapped the *String* value, *“Hello world!”* in quotes. When creating a *String* literal, make sure to wrap your text with quotation marks!
* A *String* can be printed just like any other variable. Just stick the variable in a *print* statement and you’re good to go!

**Print Statements**

The first thing we are going to learn in Java is how to print to the console. You can’t do much as a beginner programmer if you don’t know how to print! It’s not difficult. I promise.

Let’s get started:

In Java, we print using the command:

//Put whatever you want to print inside the parenthesis

System.out.println();

This is the one of the *most compiled statements in all of Java*. Printing is absolutely essential. Everyone loves printing. Everyone needs printing. And fortunately, you can do it with one simple line.

Let’s try and print something out. We’ll print out a classic “Hello, World!” For those of you who don’t know, most programmers, when learning syntax for a brand-new language, like to print a simple “Hello, World!” to make sure they got the very basics down. Let’s make our first line of code:

System.out.println(“Hello, World!”);

Once you compile, you should see the text appear on the console. Pretty easy!

If you are observant, you may have realized that inside the print statement, we put quotes around the text. This is necessary if you’re trying to print text out to the console!

You can use multiple print statements, of course. With *System.out.println()*, the console automatically goes on to the next line after the text is printed. Here, let me show you:

System.out.println(“Hello”);

System.out.println(“World!”);

The console first prints “Hello” and then automatically jumps to the next line. Then it prints “World!” and automatically jumps to the next line (but since we’re not printing anything else, that doesn’t really matter). Thus, the output will look like:

Hello

World!

You may be wondering, “Okay, that’s cool and all, but is it possible to have it *not* automatically jump to the next line every time it prints out text?

The cheeky answer is “yes, just put it all in one print statement.”

The educational answer is “yes, use *System.out.print()*”

*System.out.print()* is the little brother of *System.out.println()*. Don’t be scared, because you would do the same thing as you would with *System.out.println()*—simply just stick your desired text inside the parenthesis and wrap it with quotes.

//Output: Hello, World!

System.out.print(“Hello, World!”);

The difference between the two is that *System.out.print()* does not automatically jump to the next line. If you use multiple *System.out.print()* statements, it will keep printing in the same line. Let’s revisit the example above:

System.out.print(“Hello”);

System.out.print(“World!”);

Notice in this example, we’re using *System.out.print()*. Since *System.out.print()* doesn’t automatically jump to the next line, the output would look like:

HelloWorld!

Feel free to mess around with a mixture of *print()* and *println()* statements to get a general feel of how printing works in Java!

**Arithmetic Expressions and Type Casting**

Java lets you use arithmetic expressions—the ones we know and love, like addition, subtraction, multiplication, and division. We use *+* to signify addition, *-* to signify subtraction, \* to signify multiplication, and / to signify division.

//Declare an int named *i* with a value of 10 + 5 (which is 15)

int i = 10 + 5;

//Print out the product of 15 \* 15 (which is 225)

System.out.println(15 \* 15);

Java follows your traditional order of operations (PEMDAS, for those of you who remember the mnemonic):

//The output should be 8

System.out.println(3 + 5 \* 3 – 10);

Division is when things get a little bit strange. What do you will be outputted when I print this arithmetic statement?

System.out.println(5 / 2);

To those of you who are exceptionally skilled in division, 2.5 seems like the obvious (and only) answer. However, what if I told you the output was not 2.5, but rather… 2?

No, I did not make a computational error. And yes, Java understands how division works. The reason lies within the numbers themselves. This is a peculiar thing in Java, known as *integer division*. When dividing two integers, we throw away the remainder, or fraction or decimal or however you want to think about it, leaving the answer a whole number.

5/2 = 2.5 = 2

We all know that 5 / 2 = 2.5. But since the two numbers are integers, we drop the .5, leaving us with an answer of 2. Note how we don’t round. We simply truncate, or drop the values proceeding the decimal

Here’s some more examples:

System.out.println(15 / 4);

//Will output 3 (3.75 -> 3)

System.out.println(-5 / 2);

//Will output -2 (-2.5 -> -2)

System.out.println(1 / 5);

//Will output 0 (0.2 -> 0)

Integer division can be very useful, and you’ll be able to see it in action when we get deeper with Java. For now, though, it’s nice to know that it exists.

At this point, you’re probably wondering: “so is it possible to do *real* division? Because 5 divided by 2 is actually 2.5, regardless of what Java thinks.”

Yes! You absolutely can. Java will do *real division* when it sees a *double* as one of the operands. You just need to turn one of the numbers into a *double*, or in other words, a decimal number. So, for the number 5, simply change it to 5.0. Java will treat this as a *double*.

//Will output 2.5

System.out.println(5.0 / 2);

//Will also output 2.5

System.out.println(5 / 2.0);

//Will also also output 2.5

System.out.println(5.0 / 2.0);

As long as one *double* value is present, you will get a *double* output.

System.out.println(1 + 5 / 2 – 1.5); //Output is 1.5

/\* Order:

\* 5 / 2 = 2 (int divided by int results in an int quotient)

\* 1 + 2 = 3 (Add 1 to the quotient of 5 / 2 which yields 3)

\* 3 – 1.5 = 1.5 (Subtract the sum by 1.5 to get 1.5)

\*/

Another way to force an *int* to be a *double* is to type-cast it as a *double*.

System.out.println((double) 5 / 2);

The *(double)* represents *casting*—we are forcing the *int* value 5 to become a *double*—5.0. Thus, the output will be 2.5.

Before we go too deep into that, let’s throw in some data types into the picture.

int num = 10;

int doubleNum = num;

//What are the outputs?

System.out.println(num);

System.out.println(doubleNum);

In this example, we have an *int* named *num* with a value of 10. Okay, seems normal so far.

Then, we have a *double* named *doubleNum* with a value of *num*. Well, we know *num* is 10, so *doubleNum* should be 10 as well. Right?

Close! Remember that a *double* variable has to be a decimal number. Since 10 is a whole number, and we are expecting a decimal number, Java automatically turns 10 into 10.0. Thus, the print statements will yield:

10

10.0

This is known as *upcasting*. We pass a smaller type (*int*) to a bigger type (*double)*, and Java is kind enough to automatically convert it for us. Thanks, Java!

This might seem trivial since it’s the same number, but just think about the statement:

System.out.println(num / 3); //10 / 3

System.out.println(doubleNum / 3); //10.0 / 3

The first one is integer division, since *num* is an *int*. Do the math and you get 3, dropping the .33. Thus, the output is 3

The second one is normal division, since *doubleNum* is a *double*. This will yield an output of 3.33

Here’s another example to really drive the point across.

double num = 5;

//5 is upcasted to 5.0; the output is 5.0

System.out.println(num);

This is **not** valid:

double num = 3.2;

int num2 = num;

Hold it right there! This is invalid casting. Since *double* is a bigger type than *int*, trying to force a decimal number into an *int* data type (which only accepts whole numbers) will give you an error: incompatible types.

Here is a more acceptable way to do something like the example above, this time without Java getting upset:

double num = 3.2;

int num2 = (int) num;

Here we are manually casting the *double* 3.2 into an *int*, making it 3. This is acceptable, since we turn 3.2 into a value that is acceptable for an *int* to handle. This is known as *downcasting* and must be manually done—don’t make the mistake of thinking Java will automatically convert it for you like it would for upcasting. This is something you have to do yourself. Don’t worry, though, you’ll get an error letting you know if you forget.

**The Modulo Operator**

Let’s wrap up this lesson with a new operator that some people may not have heard of: the modulo (or mod) operator. It’s represented by the % symbol.

Modulo represents the remainder when two whole numbers are divided.

For instance, let’s look at this example:

int a = 5;

int b = 2;

System.out.println(a % b);

What’s the remainder when 5 is divided by 2? If you’re a mathematics prodigy, you would answer that it’s 1. Thus, the output of the print statement is 1.

//What’s the remainder when 15 is divided by 10? It’s 5

System.out.println(15 % 10); //Output: 5

//What about the remainder when 100 is divided by 9? It’s 1

System.out.println(100 / 9); //Output: 1

//And how about the remainder when 8 is divided by 2? It’s 0

System.out.println(8 % 3); //Output: 0

Some fun facts about modulo (very important for those who are competing in UIL):

* The sign of the *first* operand decides the sign of the result.

//The sign of the first operand is negative, output is -2

System.out.println(-5 % 3);

//The sign of the first operand is positive, output is 2

System.out.println(5 % -3);

* If the first operand is smaller than the second operand, the result is the value of the *first* operand.

//1 < 3. The result is 1

System.out.println(1 % 3);

//23 < 81. The result is 23

System.out.println(23 % 81);

//Tricky! Testing both fun facts! The result is -3

System.out.println(-3 % 8);

Be careful not to mix up division and modulo—they’re similar!

**Concatenation**

The + operator isn’t just for adding numbers in Java—it lets you join variables onto print statements. You know what, it’s probably much easier to just show you:

String name = “David”;

System.out.println(“My name is ” + name);

//Output: My name is David

We’re “adding” the variable *name* to the end of the print statement. This is known as *concatenation*, which is just a fancy word for joining two different strings together. In the example above, we concatenated *“My name is ”* and the value of the variable *name* (“David”).

String firstName = “David”;

String lastName = “Jones”;

String fullName = firstName + “ ” + lastName;

System.out.println(fullName);

Here’s a longer example: *String fullName* concatenates the variables *firstName* and *lastName* as well as a space. When we print out *fullName*, we get:

David Jones

Before we move on, I want to reiterate that you *need* a + operator if you’re concatenating. The following example will **not** work:

String name = “David”;

System.out.println(“My name is ” name “, nice to meet you!”);

There’s a noticeable lack of the + operator that we need to concatenate. Here’s how it’s done:

String name = “David”;

System.out.println(“My name is ” **+** name **+** “, nice to meet you!”);

//Output: My name is David, nice to meet you!

We can concatenate more than *Strings*; we can also concatenate variables using the same principle.

int age = 25;

System.out.println(“I am ” + age + “ years old.”);

Output: I am 25 years old.

And of course, if we’re working with numeric values or numeric data types, like *int* or *double*, you can do arithmetic.

System.out.println(“12 \* 12 = ” + 12\*12);

//Output: 12 \* 12 = 144

int num = 12\*12;

System.out.println(“12 \* 12 = ” + num);

//Output (same as above): 12 \* 12 = 144

Okay, let’s move on to a slightly trickier example. This will illustrate some of the semantics of Java.

int age = 25;

System.out.println(“In 10 years I will be ” + age + 10);

Interesting. The first + is obviously concatenating *“In 10 years I will be ”* and *age + 10.* The second + is an arithmetic expression, adding 10 to the value of age. The intended output should be:

In 10 years I will be 35

However, the actual output is

In 10 years I will be 2510

What gives? If my math is accurate, if you’re 25 years old, you’ll be 35 in 10 years, not 2510. Where did that number come from?

If you’re observant, you may realize that 2510 is in fact not a random number—it’s the concatenation of the value of the variable *age* and 10. Java treated *age + 10* literally—instead of adding 25 + 10, it just stuck 10 to the end of 25, leaving you with 2510. It concatenated the numbers, as it would for *String*.

This is a golden rule. The Java compiler evaluates expressions from left to right. When it encounters a *String*, it also considers the rest of the entire expression as a *String* as well, which is why 25 and 10 were concatenated instead of added. Anything before the *String* is still evaluated normally.

To circumvent this, put parenthesis around the arithmetic expression:

System.out.println(“In 10 years I will be ” + (age + 10));

Now Java knows we want to treat it as an arithmetic expression.

In 10 years I will be 35

Thus, when you’re using + as an arithmetic operation rather than concatenating, you should enclose the arithmetic expression with a set of parenthesis ( ). It looks a lot nicer. Also, you may encounter some strange outputs if you don’t.

I know this can be a little confusing, so here’s some additional examples.

System.out.println(1 + 2 + 3 + 4 + “Five”);

/\* Java reads from left to right: the integers precede the String so they are treated like ints

Output: 10Five

\*/

System.out.println(“Five” + 1 + 2 + 3 + 4);

/\* Java reads from left to right: the ints follow the String literal “Five”, so the ints are treated as Strings and concatenated

Output: Five1234

\*/

System.out.println(1 + 2 + 3 + 4 + “Five” + 4 + 3 + 2 + 1);

/\* The ints preceding the String are treated like ints; the ints following the String are treated as Strings and concatenated

Output: 10Five4321

\*/

System.out.println(“Five” + (1 + 2 + 3 + 4));

/\* Parenthesis indicates to Java that it should treat it as an arithmetic expression

Output: Five10

**If Statements**

Let’s say that you have an *int* called *courseGrade*. It represents the grade you got in a certain course. The range is 0 – 100 (we’ll assume no extra credit).

Now, let’s say I ask you to print out whether or not you’re passing the class. Let’s say a score of 70 or above is passing. Anything below that is a failing grade. So, for example, if *courseGrade* is 85, you pass. If *courseGrade* is 50, you fail, etc.

This is certainly tricky. For starters, what’s being outputted depends on the value of *courseGrade*, which can be whatever the user wants. What should we do when there is a relationship between variable and output?

With that question in mind, I introduce you to… the *if statement*:

if (statement)

{

do something;

}

Yeah, okay, so that was a bit melodramatic. But let’s dissect the statement line by line before we go and plug anything into it:

if (statement)

First things first: you may have noticed something interesting.

*This line doesn’t end with a semicolon*.

The initial declaration of a control statements and loops usually don’t include a semicolon. Consider this a rare exception!

The statement embedded within the *if statement* is known as the *condition*. The condition is usually a *boolean expression* (I’ll be using *boolean expression* and *condition* interchangeably when discussing if statements and loops), meaning it must either be *true* or *false*. Here’s some examples of boolean expressions below:

int num1 = 5;

int num2 = 10;

System.out.println(num1 > num2);

//Here’s an example of a boolean expression evaluating whether or not *num1* is greater than *num2*. It is, so the output is *true*.

System.out.println(num1 + 5 >= num2);

//Is *num1 + 5* greater than or equal to (expressed as “>=”) *num2?* Well, *num1 + 5* is *10* and *num2* is *10*, so they are equal. Output is *true*.

System.out.println(num1 == num2);

//Equivalence is expressed with “==”. Remember that “=” represents initialization. In this case, we’re not setting *num1* equal to *num2*, we’re checking to see if they are equal. To do that, we use “==”

boolean b = num1 > num2;

//What do you think is the current value of *b*? Answer: false

Since boolean expressions evaluate to either *true* or *false*, they can’t be standalone. It wouldn’t make much sense.

int num1 = 5;

//The line below will give an error!

num1 > 1;

The difference between = and == may get you at first, but with some practice you’ll be able to differentiate the two, easy as pie.

int num1 = 5;

//= is to initialize or assign values!

System.out.println(num1 == 5)

//== is to compare values!

Anyways, back to the if statement:

if (boolean expression)

{

do something if expression is true

}

If the boolean expression inside the if statement evaluates to *true*, we run whatever is inside of the if statement (whatever is enclosed inside of the curly braces). If the boolean expression evaluates to *false*, we do not run whatever is inside of the if statement and move on.

Let’s go back to our question above. We want to print out whether *courseGrade* is passing or not. If your grade is 70 or above, you pass.

int courseGrade = 95;

if (courseGrade >= 70)

{

System.out.println(“You pass”);

}

In the example above, *courseGrade* is greater than 70, which means that the if statement is *true*. Thus, we run whatever is inside of the if statement. The only thing inside of the statement is the print statement, so you’re going to get an output: You pass. If *courseGrade* was not greater than or equal to 70—let’s say it was 65—then we would *not* run whatever is inside of the ifstatement, and thus we would get no output.

But hold on—we want to notify the user if the grade is failing. As of right now, there’s no output if the *courseGrade* is a failing grade. How can we change that?

One way is to make another if statement, like so:

int courseGrade = 95;

if (courseGrade >= 70)

{

System.out.println(“You pass”);

}

if (courseGrade < 70)

{

System.out.println(“You fail”);

}

This technically works. However, it can be simplified. In this scenario, there’s only two options: either your score is a passing score, or your score is a failing score. If *courseGrade >= 70* is *false*, we know that the score is *not* a passing score, and thus a failing score. There are only two options, after all.

For situations like those, you can attach an *else*. If the boolean expression in the *if* statement is *false*, then it will run whatever is inside the *else* statement.

int courseGrade = 50;

if (courseGrade >= 70)

{

System.out.println(“You pass”);

}

else

{

System.out.println(“You fail”);

}

//Fun fact: You can omit the curly braces { } if you only have one

line inside your if/else statements

In this case, the boolean expression in the if statement *is* *false* (50 is not greater or equal to 70), so we jump to the *else* statement. Thus, our output is: You fail.

You can have an if statement without an else statement, but not the other way around!

**If/Else If**

Let’s go back to the same problem discussed in the **If Statements** section, but with an added twist. Now, instead of printing whether or not we pass or fail, we want to print out the letter grade corresponding to *courseGrade*.

We’ll use the following grade conversion:

A: 90 – 100

B: 80 – 89

C: 70 – 79

F: < 70

If *courseGrade* is 75, the output should be “C”. If *courseGrade is* 90, it should be “A”. If *courseGrade* is 50, it should be “F”. So on and so forth.

Let’s start off with A. If your grade is 90 or above, you get an A.

int courseGrade = 95;

if (courseGrade >= 90)

{

System.out.println(“Your grade is A”);

}

In the example above, *courseGrade* is greater than 90, which means that the if statement is *true*. Thus, we run whatever is inside of the if statement. The only thing inside of the statement is the print statement, so you’re going to get an output: Your grade is A. If *courseGrade* was not greater than 90—let’s say it was 85—then we would *not* run whatever is inside of the ifstatement, and thus we would get no output. Remember?

So, can we apply the same logic for a grade of B, a grade of C, and a grade of F? Hmm… let’s see.

if (courseGrade >= 90)

{

System.out.println(“Your grade is A”);

}

if (courseGrade >= 80)

{

System.out.println(“Your grade is B”);

}

if (courseGrade >= 70)

{

System.out.println(“Your grade is C”);

}

if (courseGrade < 70)

{

System.out.println(“Your grade is F”);

}

Looks good at first glance, right? Take a look again. What do you think is wrong with this code? Take some time to think about it.

Done thinking? Alright.

Let’s assume *courseGrade* is 95. The output should be: Your grade is A. Let’s run through the *if* statements and see.

if (courseGrade >= 90)

{

System.out.println(“Your grade is A”);

}

Okay, looks good.

if (courseGrade >= 80)

{

System.out.println(“Your grade is B”);

}

Whoa. Problem. *courseGrade* is 95. And 95 is greater than 80. So, because the boolean expression is *technically true*, this line of code will run. Same for the following if statements. So not only will you get the output: Your grade is A, you will also get the output: Your grade is B. And also: Your grade is C (since *courseGrade* >= 70)

Basically, in this case, we want to move through the if statements until we find one that’s *true*. And once we discover the true statement, we want out. So, we kind of need to chain these if statements together. We can do that by adding an *else* before subsequent *if statements*.

It should look something like this:

int courseGrade = 95;

if (courseGrade >= 90)

{

System.out.println(“Your grade is A”);

}

else if (courseGrade >= 80)

{

System.out.println(“Your grade is B”);

}

else if (courseGrade >= 70)

{

System.out.println(“Your grade is C”);

}

else //If your grade isn’t 70 or higher, then you get an F

{

System.out.println(“Your grade is F”);

}

Now these if statements are all connected. Instead of running through every if statement no matter what, now it only runs the following *if statement* if the previous one yielded *false*.

**Fun Fact**

The *if* statement doesn't need braces if there is only one line of code in a specific part. Here both the true and false parts have only one line of code, so we can omit the braces:

if (condition)

do something

else

do something

Notice there’s an absence of curly braces. This works **only** if there’s one single line of code. Always use curly braces if you plan to have more than one line of code inside the statement.

if (condition)

do something

do something 2

In this case, only *do something* is part of the if statement. Though *do something 2* also looks like it’s part of the group, it’s really not. That code above is the same thing as this:

if (condition)

{

do something

}

do something 2

It’s a nifty little shortcut but be careful!

**While Loops**

Let’s say I ask you to print “Hello world” 100 times. I don’t know why you would ever do that, but just humor me for the sake of this example.

Printing out the same thing 100 times isn’t difficult, but… it’s kind of repetitive, no?

System.out.println(“Hello world”);

System.out.println(“Hello world”);

System.out.println(“Hello world”);

//97 more print statements to go!

Yeah, that doesn’t look particularly fun to do. There’s got to be a more efficient way to do it, right?

The answer is yes, of course. There’s a much, *much* easier way to do this, and you only need to print once!

We can simplify this process by using loops. In this section, we’ll cover the *while* loop:

while (boolean expression)

{

do something if expression is true

}

Hmm… this is very similar to the structure of an *if statement*. There’s a boolean expression, and if that’s true, we run whatever is inside. There’s just one difference, but it is an incredibly significant difference. Are you ready? Here it is:

As long as the boolean expression is true, we will continually run whatever is inside the while loop, hence the name *while loop*.

Here’s how a while loop works:

1. Check if the boolean expression is true. If it is false, exit loop, move on. If true, go to 2
2. Run whatever is inside of the while loop
3. Go back to 1

Let me illustrate an example:

//*i, j, and k* are very popular variable names for loops

int i = 1;

while (i == 1)

{

System.out.println(“Hello world”);

}

Let’s apply the 3 steps above to see what will happen.

1. **Check if the boolean expression is true. If it is false, exit loop, move on. If true, go to 2.**

The boolean expression is checking if variable *i* is equivalent to 1 (remember, to check for equivalency we use ==, not =). We know *i* is equivalent to 1 because we initialized it to 1 in the previous line. Thus, the boolean expression is true. Move to 2.

1. **Run whatever is inside of the while loop.**

There’s only one line inside the while loop: a print statement that prints “Hello world”

1. **Go back to 1**

If you’re observant, you’ll notice that there’s something terribly wrong with this while loop.

int i = 1;

while (i == 1)

{

System.out.println(“Hello world”);

}

In the example above, the variable *i* will *always* be equal to 1. We don’t ever modify the value of *i*, so every time we loop and check the boolean expression (*i == 1*), it’s always going to be true. That means that the loop will run infinitely, and most of the time, that isn’t what you want.

How do we modify this loop so that it *doesn’t* run infinitely? Well, let’s look at the boolean expression: *i == 1*. We just have to modify the while loop so that at some point, *i == 1* is false.

int i = 1;

while (i == 1)

{

System.out.println(“Hello world”);

i = i + 1;

}

We added a new line inside the while loop: *i = i + 1*;

Now, after we print out “Hello world”, we increment the value of *i* by 1, making the value 2. When we run the while loop again to test the condition, we know that *i == 1* is *false*, as *i* is now 2; therefore, we exit the while loop.

Logically, this makes sense. Pragmatically though, it’s not really useful. So let’s go back to our previous example about printing out “Hello world” 100 times.

The common way to do this is to start *i* at 0, increment it by 1 and stop before it reaches 100. In Java (and a lot of other programming languages), we tend to start counting at 0 instead of 1. A little strange, for sure, but once we start learning arrays, you’ll understand the reasoning behind it.

We want to print “Hello world” 100 times, which means that we want the loop to run 100 times. By incrementing *i* by one 100 times, we cause the loop to repeat 100 times. We want to stop after we loop 100 times. After 100 loops, the value of *i* will be 100. Thus, the condition *i < 100* ensures we do not go beyond the desired loops.

int i = 0;

while (i < 100)

{

System.out.println(“Hello world”);

i = i + 1; //Can also be expressed as *i++;*

}

Let’s break this down line by line:

int i = 0;

We’re initializing an *int* named *i* with value 0. We’re going to be using this as the “counter” for the while loop: we slowly increment this value, stopping before it reaches 100.

while (i < 100)

Remember that for a while loop to run, the condition must be *true*. *i* is equal to 0 right now, so of course *i* is less than 100.

System.out.println(“Hello world”);

We print our desired output.

i = i + 1;

This beautiful line of code ensures that our while loop does not run infinitely. Incrementing *i* by 1 guarantee that at some point (in this case, 100 loops later), the condition eventually will yield *false* and we break out of the while loop.

While loops are a little confusing at first. Don’t worry if you feel a little confused or overwhelmed. Try some practice problems and mess around with it on your compiler.

Before we wrap this up, instead of printing out “Hello world” 100 times, let’s print out 1 2 3 4 5… all the way up to 100. How can we do that? Think about it yourself, then look at the solution.

**Solution**:

Here’s two ways we can do this:

Method 1:

int i = 0;

while (i < 100)

{

System.out.println(i + 1);

//I’ll be using i++ instead of i = i + 1, they mean the same

i++;

}

\**i++* is a special shortcut for *i = i + 1*, but its structure is very unique, you can’t use *i+2* for *i = i + 2;*

Method 2:

int i = 1;

while (i <= 100) //or i < 101

{

System.out.println(i);

i++;

}

And as an afterthought, what would be the output of these two?

int i = 100;

while (i > 0)

{

System.out.println(i);

i--;

}

int i = 1;

while (i <= 1024)

{

System.out.println(i);

i = i \* 2;

}

**For Loops**

There’s one more important loop that we have to discuss, and that’s for loops. For loops are very similar to while loops. It’s best if I just show you, so let’s jump into it.

Let’s say we want to write a program to print out all numbers from 1 to 100. Here’s how we do it via a *while loop*:

int i = 1;

while (i <= 100)

{

System.out.print(i + “ ”);

i++;

}

//Output is: 1 2 3 4 5… all the way to 100

If this is still tricky, I recommend going back to the lesson on while loops.

You can do the same thing with a for loop, and it’s actually easier. It can be done in one line. I’m actually going to skip the long-winded explanation and just show you how it’s done:

for(int i = 1; i <= 100; i++)

{

System.out.print(i + “ ”);

}

Okay, let me give you some time to digest this *for loop*. Take a look at it for a minute, and mentally note some observations you see. I color-coded some lines of code for you to demonstrate the similarities between the loops.

Here are some of my observations:

1. The *for loop* has the same structure as the *while loop*, but instead of there being just a boolean expression inside there’s two other statements, separated by semicolons.
2. The first statement declares/initializes a value, the second statement is a boolean expression/condition, and the third statement is the update to make sure the loop does not run infinitely. Since the update is built in the *for loop*, you are much less likely to have your loop run infinitely.
3. It’s a lot shorter than a while loop.

You may be wondering: the *for loop* looks a lot nicer and easier to code than *while loops*! Why not choose for loops over while loops? While it’s true that every *for* loop can be rewritten as a *while* loop and vice versa, the *while* loop is actually used for more complex use cases, such as when we don’t know how many times we want to loop. Check out the **Loop Use Case**section to learn more about the differences between the two.

Basically, when you know how many times you want to loop through some chunk of code, use a *for* loop.

**Example 1:** Print out all the integers from 1 – 1000 that are divisible by 17.

In this example, we want to start at 1, and stop when we reach 1000. Here’s two ways to do it:

**Method 1:** Classic brute force

The term *brute force* in computer science basically means you try every single possible scenario to see if it fits what you’re looking for. Basically, we’re brute forcing the answer by trying literally any possible scenario. This works because computers are very fast, and they can usually do it in good time. However, it’s often not the most optimal scenario—there’s usually a much prettier and more concise solution.

With that being said, let’s brute force this. We’ll literally try every integer from 1 to 1000 and check if it is divisible by 17. If it is, print it out and move to the next number. If not, don’t print it out and move on to the next number.

How do we check if a number is divisible by 17? Well, to answer that we have to first define what it means for a number to be divisible by another number. For a number *n* to be divisible by *k*, *n* must be cleanly divisible by *x*—meaning it must have a remainder of 0 when we divide it with *x*.

For example, 15 is divisible by 5, because when we divide 15 by 5, we get 3 with a remainder of 0.

21 is divisible by 3, because when we divide 21 by 3, we get 7 with a remainder of 0.

9 is *not* divisible by 5, because when we divide 9 by 5, we get 1 with a remainder of 4.

This is where the modulo (%) operator comes in handy! Remember that modulo helps us find the remainder of a number. Thus, if some number % 17 is 0, then we know it is a multiple of 17.

for (int i = 1; i <= 1000; i++)

{

//If there is no remainder when we divide by 17, print it

if (i % 17 == 0)

{

System.out.println(i);

}

//If it’s not a multiple of 17, move on… no need to do anything

}

**Example 2**: The cheeky way

Well… we’re looking for multiples of 17 right? 17, 34, 51, etc. Why not just increment *i* by 17

for (int i = 17; i <= 1000; i = i + 17)

{

System.out.println(i);

}

While this *does* work… it’s a little bit like hard coding. Example 1 is, in my opinion, the way to do it. Yes, example 2 is easier, and I suppose it would be perfectly acceptable to do if the question was as simple as finding multiples of 17, but you’ll see as we progress through more intricate examples that hard coding is generally frowned upon.

**FizzBuzz**

It’s time for a classic technical interview question.

Yes, you heard right. I said technical interview. Don’t be scared, though. You have all the knowledge you need to solve it.

This is a commonly asked question in technical interviews. It’s kind of like a weed-out question to separate those who can code with those who just flat-out can’t. It’s called “FizzBuzz.” You ready?

FizzBuzz is actually a game people play. The first player starts by saying out loud the number 1. The next player says the next number, and so on. Pretty simple game, no? Here’s the catch:

- If the number is a multiple of 3, you must say “Fizz” instead.

- If the number is a multiple of 5, you must say “Buzz” instead.

- If the number is a multiple of both 3 and 5, you must say “Fizzbuzz” instead.

If you say the wrong number or word, you’re out. Last player standing wins.

You’ve probably figured out what this question is about: given a range of numbers from 1 to… let’s say 100, print out all the numbers with the FizzBuzz rules above.

The tricky thing here isn’t defining the *for* loop. In fact, it’s pretty easy. The question explicitly says *all* numbers from 1 to 100. We have everything we need to make our *for* loop. The difficulty here is the logic. Let’s see…

So, if the number is a multiple of 3, print “Fizz”. That’s not too bad… we can test multiples using the modulo (%) operator.

If the number is a multiple of 5, print “Buzz”. That’s also not bad since it’s pretty much the same logic as the first rule. Use modulo to test if it’s a remainder of 5.

And if the number is a multiple of both 3 and 5 (if you’re astute, you may have realized this is the same thing as saying if it’s a multiple of 15), print “FizzBuzz”. This is the same logic! If it’s a multiple of 3 *and* a multiple of 5, we’re good!

If the number doesn’t fit any of the above rules, we print it out as is. Hey, this isn’t too bad! Let’s use *if* and *else if*, since we don’t want to run anymore statements if the condition is met (remember that *if-else if* statements act like a connected chain instead of separate statements).

Give it a shot and come back when you think you’ve got the solution.

**Solution..?**

Did you come up with something like this?

for (int i = 1; i <= 100; i++)

{

if (i % 3 == 0)

System.out.print(“Fizz” + “ ”);

else if (i % 5 == 0)

System.out.print(“Buzz” + “ ”);

else if (i % 3 == 0 && i % 5 == 0)

System.out.println(“FizzBuzz” + “ ”);

else

System.out.println(i + “ ”);

}

Hmm, looks good… or is it? Let’s test out some numbers:

Suppose *i* is 3. We run the first *if* statement, and *i % 3* does in fact equal 0, as *3 / 3* yields a remainder of 0. We print out “Fizz”. Working as intended…

Now let’s suppose *i* is 5. The first *if* statement is obviously false, but the second one is true. We print out “Buzz”. So far so good.

How about 15? 15 is a multiple of 3 *and* 5, so theoretically it should print out “FizzBuzz”. But notice what happens at the first *if* statement. 15 *is* divisible by 3, actually. Thus, the *if* statement **will** run, and we will output “Fizz” instead of “FizzBuzz” as was intended. That’s a problem.

Notice the logic error here? The if statements are out of order. First, we should check if they’re both multiples of 3 and 5 first before checking the individual cases.

**The Actual Solution**

for (int i = 1; i <= 100; i++)

{

if (i % 3 == 0 && i % 5 == 0)

System.out.print(“FizzBuzz” + “ ”);

else if (i % 3 == 0)

System.out.print(“Fizz” + “ ”);

else if (i % 5 == 0)

System.out.println(“Buzz” + “ ”);

else

System.out.println(i + “ ”);

}

If you managed not to fall for that trap, good for you! You’ve got a knack for critical thinking and being thorough in your problem solving, skills extremely valuable for programming. And if you fell for that dirty trick, don’t fret! Now you’ll sure to be more careful when evaluating problems, and that’s a fantastic habit to build early on in your programming journey.

Congrats! You’ve learned about *for* loops and how useful they are, tackled an introductory interview question, and hopefully learned a thing or two about logic. A job well done!

**Variable Scope**

Okay, pop quiz: what’s the output of the code below?

int num = 15;

if (num % 10 == 5)

{

int num2 = 100;

num = 0;

}

System.out.println(num2);

Is the code above useful? No, not really. But it serves a pretty valuable purpose. Anyways, back to the question. What do you think the output of the code is going to be? Think about it.

Well, the answer seems pretty obvious. We declare *num2* as *100* inside the *if* statement, and we’re printing *num2*, so it’s going to be *100*, no?

Actually… it’s not going to be *100*. In fact, the code is not even going to compile. Why do you think that is? What do you think the error is going to be?

If you typed that code inside your IDE and ran it, you’re probably going to get an error that reads similar to this:

error: cannot find symbol

System.out.println(num2);

^

Now you’re probably wondering—Java, have you lost your mind? What do you mean you can’t find *num2*? It’s literally there, inside the *if* statement.

Well, that’s the problem. See, *num2* is in the *if* statement, enclosed in curly braces. And because it’s enclosed inside those curly braces, *num2* is **only accessible by the code between those curly braces**.

Let me illustrate that. I’ll shade green where *num2* is accessible:

int num = 15;

if (num % 10 == 5)

{

int num2 = 100;

num = 0;

}

System.out.println(num2);

Since the variable was declared inside that code block, we can only use it within that specific code block. This is known as the variable **scope**—in other words, where it exists.

Maybe you’re asking the question—well, what about the variable *num* in line 1? We declared that outside of the *if* statement, but it doesn’t seem like the compiler throws us an error if we modify the value of *num* inside the *if* statement.

To answer that question, I’m going to zoom out a bit:

public class VariableScope

{

public static void main(String[] args)

{

int num = 15;

if (num % 10 == 5)

{

int num2 = 100;

num = 0;

}

System.out.println(num2);

}

}

This is the full snapshot of the code. There’s three code blocks here and I’m going to list them from biggest to smallest: *VariableScope* (the class name), the *main* method, and the *if* statement. The *if* statement is contained within the *main* method, which in turn is contained within the class *VariableScope*. In short, since we declared *num* in the *main* method, we can use it inside the *if* statement—since the *if* statement is still enclosed by the *main* method.

**Examples of Code Blocks**

Here are some examples of code blocks:

if (statement)

{

}

while(statement)

{

}

for(statement1; statement2; statement3)

{

}

If you declare a variable inside of the code blocks (inside of the curly brackets), then be careful! They exist only within those curly brackets.

Lastly, you can technically create a code block without using an *if* statement or a loop.

int num = 10;

{

int num2 = 5;

}

System.out.println(num2); //ERROR!

In this example, the curly brackets act as the code block, meaning *num2* is only accessible within said curly braces. Now, I personally haven’t really found a reason to use this form of code block before and they look pretty strange, but that’s an example of a code block that is legitimate. And yes… that code is not very useful either, but it’s served its purpose.

**Do-While Loops**

Remember the *while* loop? You know, the one that looks like this:

while (condition)

{

do something

}

We’re going to introduce a twist on the *while* loop, called the *do-while* loop. If you’ve forgotten what a *while* loop is or if you’re a little murky on how they work, re-read the **While Loops** module and come back—or you’ll find yourself a little bit lost!

Here’s what the *do-while* loop looks like:

do {

do something

}

while (condition); //Yes, the semicolon is deliberate

Notice that, unlike your traditional *while* loop, the *while (condition)* section is at the bottom of the loop structure, not at the top.

That’s actually what makes the *do-while* loop so unique. The loop actually executes the code block *first* and *then* checks the condition, as opposed to the traditional *while* loop that first checks the condition and then runs the code block.

Well, what does that mean? That means that the loop will always run at least *once*, **even if the condition is false**, because we execute the code block before checking the condition.

int i = 1;

do {

System.out.println(i);

}

while (i > 100);

Take a look at the code above. We’ve declared and initialized an *int* named *i* with a value of 1. Now, since this is a *do-while* loop, we run the *do* statement first without looking at the condition. It says to print *i*, so we print out the current value of *i*, which is 1.

Now we can check the condition. *i > 100*… hmm… now I’m not math genius, but I’m pretty sure 1 is not greater than 100. So, we exit the loop.

Compare this to your traditional *while* loop:

int i = 1;

while (i > 100)

{

System.out.println(i);

}

Nothing will print out here; the OG while loop first checks the condition, and since *i > 100* is *false* (1 is not greater than 100), we skip the *while* loop and thus, nothing is outputted.

Remember that a *do-while* loop can also run infinitely if you’re not careful—so always be vigilant!

**Pop Quiz**

What’s the output of the code below?

int i = 0;

do {

System.out.print(i + “ ”);

i = i + 1;

}

while (i < 5);

Answer: 0 1 2 3 4

*Do-while* loops are sort of a niche case and should be used when you absolutely need a code block to run at least once. I don’t use them nearly as often as I use the traditional *while* loop, but that doesn’t mean it’s useless. Keep it in mind!