**(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*.

//This is a comment!

//Programmers use this to help explain the logic in their code.

//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.

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!

**? : - The Shorthand Method of If Statements**

Occasionally you may see code with a question mark (?) and a colon (:). This is a common way of shortening *if-else* statements to just one single line and is known as the *conditional operator*.

Let’s say for example that we want to determine whether or not you’ve passed a course. Assume a passing grade is 70 or above. We’ll store the value “Pass” or “Fail” into a *String* called *courseResult*.

Using traditional *if-else* syntax, the code looks something like this:

int courseGrade = 80;

if (courseGrade >= 70)

{

String courseResult = “Pass”;

}

else

{

String courseResult = “Fail”

}

Using the *conditional operator*, it looks a little something like this:

String courseResult = (courseGrade >= 70) ? “Pass” : “Fail”;

This shorthand form of *if-else* statements is only really appropriate if you’re storing values into a variable or methods with return values.

**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 use a print statement 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 it’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. 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!

**Switch-Case Statements**

The *switch-case* statement is a unique way to select one of several potential options to be executed.

For example, suppose we have an *int* called *day* that’s can be any number from 1 to 7. We want to print out the day of the week depending on the value of *day*. We’ll assume 1 = Monday, 2 = Tuesday, 3 = Wednesday… 7 = Sunday.

One way you can do this is with *if* statements. The alternative way demonstrated here is known as a *switch-case* statement.

The syntax looks like this:

switch (expression)

{

case x:

//do something

break;

case y:

//do something

break;

default: //optional

//do something

}

*break* works the same as it does for *while* and *for* loops—it breaks the loop at some specified condition and immediately exits, regardless of whether or not the loop is in progress. For *switch-case* statements, it exits the *switch* code block when a certain case is met.

For the example above, our code would look like this:

int day = 3;

switch (day)

{

case 1:

System.out.println(“Monday”);

break;

case 2:

System.out.println(“Tuesday”);

break;

case 3:

System.out.println(“Wednesday”);

break;

case 4:

System.out.println(“Thursday”);

break;

case 5:

System.out.println(“Friday”);

break;

case 6:

System.out.println(“Saturday”);

break;

case 7:

System.out.println(“Sunday”);

break;

}

In this case, since *day* is set to *3*, we would head over to *case 3* and print out “Wednesday”.

Including *break;* is not required for *switch-case* statements, but be warned—without *break*, it will automatically jump to the next case, *even if that specific case is false*. For example, if we removed all the *break* statements in the code above, the output would be:

Wednesday

Thursday

Friday

Saturday

Sunday

Most of the time, it would be a good idea to include *break* statements in between separate cases.

**Default**

*default* is another optional statement you can include inside your *switch-case* statement. The name is pretty self-explanatory: it’s the *default* option for your statement. If none of the cases fit, it will run whatever is inside *default*. It will always be the last statement in a *switch-case* statement (thus you don’t need a *break* statement, but you should definitely add one for the *case* above).

Here’s an example:

int day = 3;

switch (day)

{

case 6:

System.out.println(“Saturday”);

break;

case 7:

System.out.println(“Sunday”);

break;

default:

System.out.println(“Waiting for the weekend!”);

}

**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!

**Introduction to Methods**

When I first started learning Java—which was around freshmen year of high school—I remember being particularly in awe of how methods worked. They were efficient, they were clean, they were awesome. I was pretty excited when I learned about it the first time.

Okay, let’s talk about methods, also sometimes called functions—they’re pretty much the same thing with some very minute differences that we won’t really gloss over (but if you’re interested in things like that, check out the discussion [here](https://stackoverflow.com/questions/155609/whats-the-difference-between-a-method-and-a-function)). For the sake of this module (and all future ones), methods and functions are synonyms.

I think the best way to illustrate what a method is, is to give you a scenario. Let’s say we want to write a program that determines whether or not a number is odd. A number is odd if it has a remainder of 1 when divided by two. Thus, we could use the modulo (%) operator to write code that looks a little something like this:

int num = ???;

if (num % 2 == 1)

System.out.println(num + “ is odd.”);

else

System.out.println(num + “ is even.”);

Pretty simple, right? So far so good. If *num* is an odd number, like 15, the output should be: 15 is odd. And if *num* is an even number, like 8, the output should be: 8 is even. Cool.

Now, let’s say that instead of one number that we need to determine is odd or even, there’s five. So, we have *num, num2, num3*, *num4*, and *num5*.

That isn’t too bad—after all, the fundamental logic doesn’t change, but we do need to pretty much copy and paste that *if* statement four more times with different variable names. Something like this:

int num = ???;

int num2 = ???;

int num3 = ???;

int num4 = ???;

int num5 = ???;

if (num % 2 == 1)

System.out.println(num + “ is odd.”);

else

System.out.println(num + “ is even.”);

if (num2 % 2 == 1)

System.out.println(num2 + “ is odd.”);

else

System.out.println(num2 + “ is even.”);

if (num3 % 2 == 1)

System.out.println(num3 + “ is odd.”);

else

System.out.println(num3 + “ is even.”);

if (num4 % 2 == 1)

System.out.println(num4 + “ is odd.”);

else

System.out.println(num4 + “ is even.”);

if (num5 % 2 == 1)

System.out.println(num5 + “ is odd.”);

else

System.out.println(num5 + “ is even.”);

This should work, but… I mean, that’s a lot of code. A lot of *repetitive* code. It’s the same *if* statement, the same logic, pretty much the same everything except for the variable that we’re using. This way of doing it may be correct, but it’s certainly inefficient, and if you know anything about programmers, it’s that they *love* efficiency. Now, if only there was a way we could reuse the *if* statement code without having to copy paste it several times…

**Enter the method.**

A method represents lines of code that are grouped together to do some specific task, and only runs when it’s called. For example, we can create a method called *isOdd* that determines whether or not some integer *num* is odd or not—a little bit like the example above. And when we pass in values, we don’t have to copy-paste the *if* statement logic over and over again like we did in the previous example—we can just print out the method name that contains all the logic.

Okay, maybe that’s a little bit confusing, so here’s an example of a method that you’re very familiar with:

System.out.println();

Yep, the print statement that everyone takes for granted is actually a method. When you call the *println()* method in *System.out.println()* (aka when you’re invoking it to print something out), the system actually executes several lines of code behind the scenes to get your text to print onto the screen.

I bet you didn’t know that, but I don’t blame you. It’s like driving a car. There’s a lot of very complicated and intricate processes that’s allowing your car to accelerate, turn, reverse, etc. But you don’t know that, of course, since it’s all done behind the scenes, so to speak. All you have to do is turn the steering wheel and step on the pedals.

But anyways, *println()* is a method provided to you by Java from the *System* class (we’ll discuss classes in a future module, so don’t worry if the terminology is foreign to you). In this module, we’re going to be showing you how to create your own custom methods.

Here’s the outline for a generic method you may write.

public static <return type> <method name> (parameters)

{

do something

}

Let’s break the code above one step at a time.

public static

The code above represents the modifiers. *Public* means that the method is accessible for all classes. *Static* means that the method belongs to the class itself.

Wait, did you say *all* classes? What’s that supposed to mean?

I know you don’t want to hear it but hold that question. Once we cover classes and objects (which is coming soon!), all of this will make sense. But for now, just think of it as something we have to include at the start of every method.

<return type>

In a method, *return* means to basically return some sort of value from the method back to where it was called. Since methods can be used for lots of different stuff, there’s a lot of things you can return—whether it be *ints*, or *doubles*, or *chars*, etc.

And by the way, *return* is not the same thing as printing. To print a returned value, you have to use the *print()* or *println()* method. Nothing appears on the console unless you print it.

So, consider the example of the *isOdd* method. If the number is odd, we want to return “The number is odd.” If it’s not odd, then we want to return “The number is not odd.” Since we’re returning text, the return type should be *String*.

Not every method has to have a return type. There are certain scenarios where you don’t want to return anything, and we’ll cover that soon. For now, just know that if you don’t want a return type, you can’t just leave it blank—you’ll have to use a special keyword called *void* as your return type.

<method name>

This is where you create a method name. Try to name it something appropriate to what you want the method to do. After all, whenever you’re calling that particular method, you’re going to be referring to its name. For the example above, I used *isOdd* as the name of the method to determine whether or not a number is odd.

You may be wondering why the first word has the first letter in lower-case and the second word has the first letter in upper-case. This is a Java naming convention known as camel-case (cause the lower-case and upper-case characters resemble the humps of a camel or something dumb like that). Normally when you write methods, they should have the first letter of the first word in lowercase and the first letter of each subsequent word in upper-case, and they should also be verbs (to illustrate what it’s doing).

(Parameters)

This represents what we’re passing into the method, known as *parameters*. Let’s use the *isOdd* example again. *isOdd* determines whether or not an integer we give it is odd or not. So, the thing we’re passing into the method is an *int* called *num* or whatever you want to call it.

That’s the skeleton of a typical method!

Still a little confusing? Let’s put it all together and create that *isOdd* method we’ve been talking so much about.

public static String isOdd(int num)

{

if (num % 2 == 1)

return num + “ is odd.”

else

return num + “ is not odd.”

}

//The order in which methods are arranged doesn’t matter

public static void main (String[] args)

{  
 //Call the method *isOdd* with the value 5

System.out.println(isOdd(5));

//Call the method *isOdd* with the value 8

System.out.println(isOdd(8));

//Call the method *isOdd* with a big number

int bigNumber = 123456789;

System.out.println(isOdd(bigNumber));

}

Output:

5 is odd.

8 is not odd.

123456789 is odd.

Here’s the big picture. Think of it like islands. We start at the main island, the *main* method, and whenever we run *isOdd*, we send a number to the island *isOdd* and it returns with some value that we print out—in this case, a *String* that tells you whether or not the number is odd. We keep going down the *main* method until we reach the end.

Now you might be wondering—whoa, he just said *main* method. Are you telling me that *public static void main(String[] args)* has been a method all along?

Absolutely. The main method is a very strange method. It’s also a very important method, because that’s where code execution begins. We always start our code by running the *main* method.

In case you’ve noticed, the *main* method also has the keyword *static* in it. That’s pretty important because static methods cannot interact with non-static methods. Since the *main* method is accessing the *isOdd* method (three times, as a matter of fact), we should include the *static* keyword on *isOdd*, and any subsequent methods referenced by *main*.

Let’s take a break for now. We’ll discuss more in-depth about methods in the next module. For now, take a deep breath, get a glass of water, and take a break. You’ve earned it. Methods are tough stuff, and no doubt will take you a while to digest it. Make sure you’re confident with this module before we move on, because the next few modules build upon the basics we’re learning here.

And hopefully, you’re beginning to understand why I was so excited about methods when I heard about them for the first time.

**Methods, Continued**

Welcome back. We learned about methods last module, but it may have been a little overwhelming. What I basically did was drop ten pounds of information on the ground and tried to explain it all as quickly as possible. Hopefully, you understood the fundamentals of what methods are and why they’re useful. This lesson is meant to reinforce the concepts I introduced and to get you comfortable with how they work.

Suppose we want to create a method that returns the square of a given integer. For example, if you pass inside the method the integer *5*, it should return *25*, because *5* squared is *25*. If you pass inside the method the integer *10*, it should return *100*, because *10* squared is *100*. Hopefully you’re following the logic.

Let’s start by creating the method header. We’ll call the method *calculateSquare*, a fitting name for what we’re going to be doing.

public static int calculateSquare(int num)

{

//do something

}

Okay, remember that we need to start methods off with *public static*, at least **for now**.

We’re going to be returning the square of an integer, so the return type is *int* (since any integer squared will always be an integer). Be careful with this, because if we return a different type than specified in the method header, we’ll get an error. So don’t be trying to return any data type that’s not an *int*.

We only need one parameter—an *int* that I named *num*. This represents the number we’re passing into the method for it to be squared.

We want to return the square of *num*. That’s pretty easy: just multiply it by itself.

public static int calculateSquare(int num)

{

return num \* num;

}

Okay, we’re finished with the method. Let’s zoom out a little bit, and we’ll reference *calculateSquare* inside the *main* method.

public static int calculateSquare(int num)

{

return num \* num;

}

public static void main(String[] args)

{

System.out.println(calculateSquare(10));

System.out.println(calculateSquare(15));  
}

Can you guess the outputs?

For the first print statement, we’re passing in the *int 10* into the *calculateSquare* method. The method will return *100*, since 10 times 10 is 100.

For the second print statement, we’re passing in the *int 15* into the *calculateSquare* method. The method will return *225*, since 15 times 15 is 225.

Not bad, right? Let’s do another one. But before we move on, a quick warning.

What’s wrong with these two lines of code? (Assume they’re in the *main* method)

System.out.println(calculateSquare(10, 15));

System.out.println(calculateSquare(“10”));

For the first line, we’re passing in two values. This won’t work, because as you can see above in the *calculateSquare* method, it’s only expecting one, given that it only has a singular parameter of type *int*. If you don’t pass in the exact number of parameters as stated in the method header, you’ll get an error.

For the second line, the number of parameters is correct: the *calculateSquare* wants one, and we pass in one. But the *calculateSquare* method requires a parameter of type *int*, and we’re passing in a *String* (the quotes around it is a giveaway that it’s a *String*). If you don’t pass in the exact data type of the parameter specified in the method, you’ll also get an error. So be careful!

Okay, another example. Let’s say we want to create a method that will return the area of a rectangle. It takes in two parameters, *length* and *width*, both of type *int*, and will return the *area* of the rectangle.

Again, let’s start by writing the method header. We’ll call this method *getRectangleArea*.

public static int getRectangleArea(int length, int width)

{

//do something

}

The return type will be an *int*, since the product of two *ints* will always be of type *int*, and for this example we need two parameters—the *length* and the *width*—to calculate the area of the rectangle.

This is pretty simple too. The formula for the area of a rectangle is *length width*. Our logic should look very similar:

public static int getRectangleArea(int length, int width)

{

return length \* width;

}

Okay, we’re finished with the method. Let’s zoom out a little bit, and we’ll reference *getRectangleArea* inside the *main* method.

public static int getRectangleArea(int length, int width)

{

return length \* width;

}

public static void main(String[] args)

{

System.out.println(getRectangleArea(5, 10));

System.out.println(getRectangleArea(12, 8));  
}

Can you guess the outputs? We’ll leave this as an exercise to the reader :)

(The answers are *50* for the first print statement, and *96* for the second)

Hopefully, you’re beginning to get the hang of it, so we’ll step it up a notch and introduce some new concepts. Let’s use that example we did last module—create a method that determines if a number is odd. We’ve done this method before, but what I want you to do is open up an IDE and try and code it by yourself. I’m just words on a screen and I can’t tell whether or not you’re actually doing it, but I strongly recommend you do. Programming, like math and many other subjects, is not a spectator sport—you have to do it as well as read it.

Anyway, I hope you came up with something similar to this:

public static String isOdd(int num)

{

if (num % 2 == 1)

return num + “ is odd.”

else

return num + “ is not odd.”

}

This example is different from the last two—in the last two examples, the method body literally had one line of code: a *return* statement with whatever computation required to return the desired value. This example has some *if* statement logic in it, which is pretty cool.

Another difference between the two is that we have two *return* commands. Depending on whether the integer you pass in is odd or not, it will run one of the two *return* commands. It won’t run both, since there is no integer that is both odd and not odd (and before you try and get smart, 0 counts as an even number).

Here’s a neat property about the *return* keyword. When the compiler encounters a *return* keyword, it will immediately return back to where it was called with the returned value. Any code under the *return* statement is untouched.

For example:

//You can create a method with NO parameter!

public static boolean uselessMethod()

{

return false;

System.out.println(“Hi”);

}

public static void main(String[] args)

{

System.out.println(uselessMethod());

}

In this extremely useless method, once it encounters the *return* statement, it will take whatever it is asked to return (in this case, the boolean value *false*), and dip back to wherever we called it. That cute *println()* under it will never be run no matter what circumstance, and is known as *unreachable code*. Java will actually give you an error whenever you write unreachable code.

Now that you know that, let’s go back to our *isOdd* method:

public static String isOdd(int num)

{

if (num % 2 == 1)

return num + “ is odd.”

else

return num + “ is not odd.”

}

Here’s a nifty little shortcut to impress your teacher and maybe your crush. In the example above, we know that there’s only two possible scenarios: either *num* is odd, or it’s not. And since *return* will immediately terminate the method and return back to where it was called, we can simplify our method to look something like this:

public static String isOdd(int num)

{

if (num % 2 == 1)

return num + “ is odd.”

return num + “ is not odd.”

}

Think about it: let’s say we pass in an **odd** number. The *if* statement is *true*, and so we return the line of code inside the *if* statement and then immediately exit the method. Thus, that second *return* statement is untouched.

Now let’s say we pass in an **even** number. The *if* statement is *false*, and so we skip that *return* statement inside the *if*,and run the second *return* statement. So really, you don’t even need that *else* statement.

**1-Dimensional Arrays**

An array is a collection of variables of the same type.

I know, I know… not a very detailed example. Arrays help store multiple values inside one single variable, and there is a myriad of use cases for arrays.

For instance, an array of *ints* is a collection of variables of the type *int*.

An array variable is declared just like you would declare a normal variable, except you add [ ] after the type, signaling that the variable we are declaring is going to be an array. When you see [ ] in a variable declaration, it is an array! If you don’t see [ ] in a variable declaration, it is *not* an array. Make sure to include those brackets if you’re planning to create an array.

Here we create a typical *int* variable.

//Declaring an int called age and initializing it to 19

int age = 19;

Now lets create an array of type *int*. Let’s say its purpose is to store the grades that five students got on the last mathematics exam.

int[] grades;

int grades[];

//Either format works, usually the first one is more common.

To initialize, or instantiate an array means to populate it with some values. Right now, our *grades* array is empty. All we did was declare it. We haven’t actually put any values inside yet. There’s two ways to fill an array with values.

**Method #1**

**int**[] grades = new **int**[5];

This creates an *int* array named *grades* with 5 *int* values inside (that’s what the 5 means). Since we didn’t clearly define them, they’ve been initialized to the default value of 0.

Here’s a little visualization:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |

**int**[] grades = new **int**[10];

This creates an int array named *grades* with 10 int variables inside.

The *new* keyword is a keyword reserved for creating *objects*, a deep topic that I’ve been hinting at but haven’t gotten around to explaining yet. You’ll see the *new* keyword appear very frequently when we move on to classes and objects. In Java, arrays are treated as objects.

In the second example, we’ve populated the array with 10 values. But since we didn’t really assign them any specific integer value, they remain at a default value of 0. Basically the rules are, if you don’t explicitly define values for your arrays, they’re gonna end up with default values.

So, what are the default values for each data type?

* Integer: 0
* Byte: 0
* Float or Double: 0.0
* Boolean: false
* String/Object: null

boolean[] values = new boolean[100];

Here we’ve created a boolean array with the name *values* that contains 100 booleans with the value of *false*.

Alright, hopefully by now you’ve mastered the art of making arrays with default values. How do we create an array with our own defined values?

**Method #2**

To put our own values in there, we’re going to use curly braces and type in explicitly what values we want in the array.

int[] grades = {95, 100, 92, 99, 43};

//I’m usually the guy at the very end of the array with the 43

String[] classes = {“Mathematics”, “Science”, “History”, “English”};

char[] letterGrades = {‘A’, ‘B’, ‘C’};

Another way to create arrays with defined values is to do a combination of both Method #1 and Method #2

int[] grades = new int[5]{95, 100, 92, 99, 43};

Here you’re creating an *int* array with 5 values, and then immediately populating it.

Keep in mind that arrays are immutable—that means they can’t grow or shrink. Be careful when selecting the size of your array, because once you do, you can’t change it!

**Length**

You can access the length of an array through a special *length* property that’s part of the array.

int[] grades = {95, 25, 93, 99, 100};

**int length = grades.length;**

System.out.println(length); //Output: 5

//Or you can just do…

System.out.println(**grades.length**); //Output: 5

**Accessing a Specific Element of an Array**

Arrays are *zero-indexed*, meaning we start counting from 0 instead of 1. Zero-index is a property that we will continuously see not only in Java but in many programming languages. If you’re curious why that is, read a high-level explanation [here](https://medium.com/@albertkoz/why-does-array-start-with-index-0-65ffc07cbce8).

int[] grades = {95, 25, 93, 99, 100};

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 95 | 25 | 93 | 99 | 100 |

0 1 2 3 4

When you’re looking for index 1 of *grades*, you’re actually looking for the *second* value, not the first, so index 1 of the *grades* array is actually *25*. Zero-index is a tricky thing for nascent programmers to learn, but you’ll get the hang of it *very* quickly.

What is index 3 in the *grades* array? (Answer: 99)

What about index 2? (Answer: 93)

What about index 5? (Answer: Out of bounds! Trick question—this array only goes up to index *4*. Any further and we go out-of-bounds).

Array values can be referenced by index using the format *arrayname[index]*.

int[] grades = {95, 25, 93, 99, 100};

System.out.println(grades[0]); //Output: 95

System.out.println(grades[3]); //Output: 99

System.out.println(grades[5]); //Error: index is out of bounds!

**Print out an Array**

You can loop through all the elements of an array using a for loop. Here is an example of looping an array with a for loop:

String names = {“Kevin”, “John”, “Sam”, “Daniel”, “Jacob”};

for (int i = 0; i < names.length; i = i + 1)

{

System.out.println(names[i]);  
}

Notice that this *for* loop starts at 0 and ends right before *i* reaches *names.length*. We print out the array at index *i* until we reach *names.length – 1*.

In the example above, *names.length* is 5, since *names* has five elements—Kevin, John, Sam, Daniel and Jacob. So in the *for* loop, *i* starts at 0 and ends at 4. Thus, when we print out the array, we print out all the values from index 0 to index 4.

Alternatively, you can use the *Arrays.toString()* method that’s predefined for you by Java in the *util* package. Just another example of the power of methods. Although I’m curious to see how it’s done behind the scenes. Perhaps the same method as the *for* loop above?

String names = {“Kevin”, “John”, “Sam”, “Daniel”, “Jacob”};

System.out.println(Arrays.toString(names));

//Output: Kevin

John

Sam

Daniel

Jacob

**2-Dimensional Arrays**

Before we start on 2D arrays, make sure you’re very familiar and comfortable with 1D arrays! Take a moment to review them.

Okay, are you guys ready?

2D arrays are—as you may have figured out—arrays that are represented in two dimensions. 2D arrays are represented with rows and columns.

2D arrays follow the same declaration and initialization rules as 1D arrays, but with a twist: instead of one set of brackets, you have to use two!

Let’s do a quick demo.

int**[][]** ary = new int**[2][3]**;

// row col

Notice how there’s two sets of brackets. We should always use two sets when we intend to create a 2D array. In the example above, we’ve created a 2x3 array (2 rows and 3 columns) with default values of 0. It looks like this:

|  |  |
| --- | --- |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |

Think of it like a table, because honestly, that’s really what it is. If you’ve ever messed with Excel or some other spreadsheet software, this should be second nature to you.

int**[][]** ary = new int**[3][4]**;

System.out.println(ary.length); //Prints row length: 3

System.out.println(ary[0].length); //Prints col length: 4

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

Or we could create an array with defined values. To do that, wrap your rows around an additional curly bracket and separate them with commas. Here’s an example:

int[][] ary = { {1, 2, 3},

{4, 5, 6},

{7, 8, 9} };

//There’s 3 rows, so printing ary.length gives us 3

System.out.println(ary.length);

//Output: 3

/\* This checks for columns, at ary[0], which is {1, 2, 3}, the length inside that is 3. That represents the columns. \*/

System.out.println(ary[0].length);

//Output: 3

Notice how each row is wrapped in { }, and each comma outside the { } represents a different row.

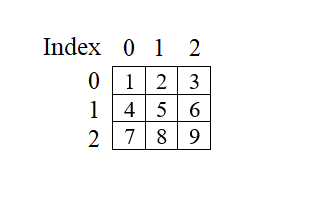
|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

Here’s another example, but without the fancy formatting so that the entire array initialization is just in one line.

int[][] ary = { {3, 1, 4, 2, 7, 2}, {1, 2, 5, 8, 9, 11} };

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 1 | 4 | 2 | 7 | 2 |
| 1 | 2 | 5 | 8 | 9 | 11 |

**Accessing 2D Arrays**

When we create a 2D array, the first index is the row. The second index is the column. Let’s use the 3X3 2D array above as an example:

int[][] ary = { {1, 2, 3},

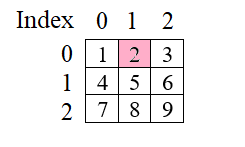
{4, 5, 6},

{7, 8, 9} };

//[Row index][Column index]

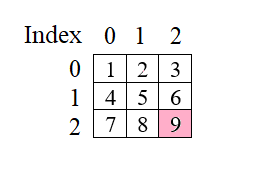
System.out.println(ary[0][1]);

//Output: 2



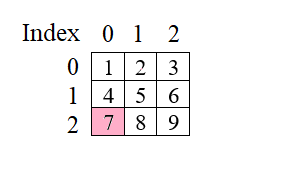
System.out.println(ary[2][2]);

//Output: 9



System.out.println(ary[2][0]);

//Output: 7



**Printing 2D Arrays**

You can print out 2D arrays with a nested *for* loop. It’s quite creative, actually. First we iterate through all the elements in the row, and then we run a second *for* loop inside our first one to loop through all the elements of the current column.

for (int i = 0; i < ary.length; i++)

{

for (int j = 0; j < ary[i].length; j++)

System.out.print(ary[i][j] + “ ”);

System.out.println();

}

**Base Conversions and Arithmetic**

Base conversions are an incredibly important topic that will come up very often on the written exam. The first question of the exam is *always* going to be a base conversion question. Knowing how to do them efficiently nets you some free points and saves valuable time!

To start, let’s talk about bases. If you’ve never heard this word used in a mathematical context, it may surprise you a little to learn that you’ve been using bases your entire life, even if you may not know it. The number system you use every day, such as when you’re adding or subtracting numbers, is part of a base known as *base-10* or the *decimal system*.

In *base-10*, each digit of a number can be *10* possible digits—0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. Each place value is *10* times larger than the place value to its right. Consider the number 325, which can be expressed as 300 + 20 + 5.

* 5 is in the *ones* place and can be expressed as Keep in mind that anything to the power of 0 is 1.
* 2 is in the *tens* place, and can be expressed as
* 3 is in the *hundreds* place, and can be expressed as

325 can be rewritten as ( or more formally, . The subscript represents the base we’re using, but since everyone pretty much uses *base-10* throughout their daily lives, we don’t include the subscript for the sake of redundancy.

So, in a base 10 system, we use 10 as the base and 0 as the exponent. Every time we move one place value to the left, we increment that exponent. Let’s use a visual example. Here’s an illustration of 82791 in *base-10*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **8** | **2** | **7** | **9** | **1** |
|  |  |  |  |  |
|  |  |  |  |  |

Hopefully, you kind of understand how bases work by now. I find that *base-10* is a great introductory example because everyone is familiar with it. Now we can move on to different bases.

**Binary**

Binary, or *base-2*, follows a similar principle. In binary, each place value is *2* times larger than the place to its right and can only have *2* possible numbers—0 or 1. A *bit* is defined as a single digit of the entire binary number. has *four* bits—1, 0, 0, and 1.

So, what exactly is the *decimal*, or *base-10* equivalent of ? To figure that out, we’ll use the same method as we did to calculate *base-10* numbers, with just one twist—instead of using *10* as the base, we use *2*, since we’re working in *base-2*.

|  |  |  |  |
| --- | --- | --- | --- |
| **1** | **0** | **0** | **1** |
|  |  |  |  |
|  |  |  |  |

Thus, the decimal equivalent of is ( Since anything multiplied by a value of 0 becomes 0, we can simplify this equation down to (



Binary pops up quite frequently on the written exam. It would be beneficial for you to memorize powers of 2 from to to save time. Remember, every second counts!

Alternatively, you may see binary written with the prefix *0b*, indicating that it is a binary number. For example, *0b1111* is equivalent to , or

There’s many other bases out there—*base-3*, *base-5, base-8* (also known as *octal*)—that operate on the same logic. Simply replace the base with the base you’re working with and apply the same methods as above!

**Hexadecimal**

We have one last base to cover, and it’s rather unique. It’s known as *hexadecimal*, or *hex* for short. It’s a *base-16* numbering system.

There are *16* possible values in *hexadecimal*—the ten decimal numbers (0 to 9), and six additional numbers (10 to 15). However, there are no numbers that can represent values greater than 9. To fix that, *hex* uses English letters for two-digit numbers. 10 is represented by *A*, 11 is represented by *B*, 12 is represented by *C*, all the way up to 15, which is represented by *F*.

Oftentimes, hexadecimal numbers may start with the notation *0x* to indicate that it’s a hexadecimal number, and not a decimal number. For example, *0x51* represents , not .

Let’s try a hexadecimal number, like *0x3FA5*, and convert it to decimal.

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | F | A | 5 |
|  |  |  |  |
|  |  |  |  |

Wow, that looks like a big number…

**Binary to Decimal**

Let’s convert to decimal.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Remember that we can just omit any zeros, so we’re left with

This is an easy way to go from binary to decimal, and make sure you memorize your powers of 2 to make it even easier.

**Base Conversion Shortcuts**

Now that we know what bases are, we can start talking shortcuts! Here are some interesting methods to convert from one base to the other.

**The Divide by Two Method (Decimal to Binary**)

Suppose you need to find the binary equivalent of 125. How would you do this efficiently?

This is a simple method that involves simply dividing the number by 2 over and over again until we reach 0. If there’s a remainder when divided by 2, we write down *1* and **truncate the number down**, similar to how Java does integer division; if there’s no remainder we write *0*. Instead of going from right to left, however, we start at the least significant bit and move leftwards.

Let’s try it on 125.

125 / 2 = 62 (Remainder: 1, Total: 1)

62 / 2 = 31 (Remainder: 1, Total: 01)

31 / 2 = 15 (Remainder: 1, Total: 101)

15 / 2 = 7 (Remainder: 1, Total: 1101)

7 / 2 = 3 (Remainder: 1, Total: 11101)

3 / 2 = 1 (Remainder: 1, Total: 111101)

1 / 2 = 0 (Remainder: 1, Total: 1111101)

**125 in binary is**

One more for good measure.

Convert 98 to binary.

98 / 2 = 49 (Remainder: 0, Total: 0)

49 / 2 = 24 (Remainder: 1, Total: 10)

24 / 2 = 12 (Remainder: 0, Total: 010)

12 / 2 = 6 (Remainder: 0, Total: 0010)

6 / 2 = 3 (Remainder: 0, Total: 00010)

3 / 2 = 1 (Remainder: 1, Total: 100010)

1 / 2 = 0 (Remainder: 1, Total: 1100010)

**98 in binary is**

**Separate and Combine (Binary to Hexadecimal)**

Suppose we want to convert to hexadecimal (*base-16*).

First, divide the binary number into groups of 4 starting from the rightmost bit. That results in and . If we started from the left, we would have gotten and , which would totally give us the wrong answer. So, make sure you start from the right side!

Now, convert the two groups into decimal. is , and is .

Finally, simply combine the two *base-10* numbers—5 and 9 becomes 59—and that’s our hexadecimal value: .

Let’s try another one: convert to hexadecimal (*base-16*).

Starting from the right, divide the binary number into groups of 4. We get two groups: and .

Now, convert the two groups into decimal. We get and respectively.

Finally, combine the two numbers—15 and 6 become *F* and 6 (remember in hex, 15 is represented by the character *F*). The hexadecimal value is

**Note:** 15 and 6 does *not* become To obtain a three-digit hex number, we would need three groups of four.

**Binary Arithmetic**

We all know how to add with base-10 numbers—well, at least I sure hope you do. In a decimal system, In binary, however, things are a little bit different. Addition follows a few different rules than the ones you’re familiar with. This is not explicitly tested on the written exam, but it’s still a very useful concept to understand.

In binary, there are four rules for addition:

* 1 + 1 = 10 ( in binary, we carry over the 1 bit to the left)
* 1 + 1 + 1 = 11 ( in binary, we carry over the 1 bit to the left)

Let’s do an example question:

Starting from right to left:

, carry the 1 bit to the left)

, carry the 1 bit to the left)

Let’s do another example:

?

Again, starting from right to left:

, carry the 1 bit to the left)

, carry the 1 bit to the left)

**Binary Subtraction**

Binary subtraction is a lot more intuitive than binary addition. There’s still 4 rules, but you’ll find that they bear a pretty striking resemblance to the subtraction rules we’re familiar with.

(borrow 1 from the next left-most digit with a 1)

Here’s an example question:

?

From right to left:

(borrow 1 bit)

(originally 0)

(originally 1 but we borrow a bit)

For more examples of binary subtraction, take a look [here](http://sandbox.mc.edu/~bennet/cs110/pm/sub.html).

**Signed and Unsigned Binary Values**

Binary numbers can be represented in two ways: signed, or unsigned, though that’s probably not a very helpful statement if you don’t know what it means for a number to be signed.

Do you ever wonder how we can represent *negative* binary numbers? It’s a little bit more complex than slapping a negative (-) sign before the binary number. The written exam likes to test you on whether or not you know how to calculate them. Chances are you’ll encounter a type of question like this on every written exam you’ll take.

**Unsigned Numbers**

Unsigned numbers can only be positive. This is the binary that we’re familiar with. When you see a binary number—say, , you assume it’s positive by default. So, you’d say something like, “the unsigned representation of this binary number in decimal is .”

However, if we’re talking about the signed representation of the binary number , we’ll get an entirely different number.

If you’re still a little confused about converting from decimal to binary and vice versa, give the *Base Conversion and Arithmetic* module a re-review.

**Signed Numbers**

For signed numbers, the most significant bit (aka the bit with the greatest number; the left-most one), determines if the binary number will be positive or negative. That bit is known as the *sign bit*, because it determines the sign (positive or negative) of the binary number. The range of numbers for signed numbers is smaller compared to unsigned binary numbers, since one bit is used for the sign bit.

If the most significant bit is a 0, the number can be assumed *positive*

If the most significant bit is a 1, the number can be assumed *negative*.

Let’s illustrate this: assume is a signed binary value and we want to convert it to decimal. The sign bit is *1*, meaning that this is a negative number. All we need to do is find the binary value of , which is . Accounting for the sign bit, our answer is

Or how about The sign bit here is 0, meaning this is a positive number. Thus, our answer is .

Here’s the thing, though. Having the ambiguity of signed numbers and unsigned numbers make arithmetic such as addition quite difficult.

For example, let’s say we’re adding 3 and -2 together. In signed binary format, they would be represented as and respectively.

Okay, let’s try and add the two numbers together.

0011

+ 1010

= 1101

We got a sum of , whereas the sum we expected to get was Using the signed number method I introduced above, basic arithmetic computation like addition doesn’t yield the right answer. That’s confusing.

Another problem is that there’s two ways to represent 0:

0000 – Zero

1000 – “Negative” zero, which is still zero

This makes any computation involving 0 a particular pain. Hopefully, I’ve illustrated some compelling arguments as to why that version of representing signed numbers is not particularly helpful. Fortunately, because the world is populated with geniuses, a new way of representing signed value was created, one that does not allow arithmetic ambiguity and double-zero problems. This is the format you’re most likely to encounter on the written exam.

**Two’s Complement**

Most computers represent integers using this format. It’s easy to learn, efficient, and eliminates a lot of issues that we mentioned above.

The steps to convert a negative number to two’s complement is three-fold. Let’s use -30 as an example number, and we want to convert it into two’s complement notation. We’ll assume its 8-bits (and so will the exam cause it’s *way* easier to do it with 8 bits).

First, let’s write out 30 in binary. Yes, *positive* 30. I know we’re dealing with a negative number, but we want the positive number in binary. You’ll kind of see why in a bit.

I’ll save you the work and just write out 30 in binary for you:

Now, invert the bits. Every bit with a 0 becomes 1, and vice versa.

Now, simply add 1 to the current binary number, and we’ll be done.

1110 0001

+ 0000 0001

= 1110 0010

Thus, the two’s complement notation for is

One more example for good measure:

Convert -45 to signed 8-bit two’s complement notation:

First, write out 45 in binary.

Now, invert the bits.

Add 1.

1101 0010

+ 0000 0001

= 1101 0011

The two’s complement notation for is .

**Two’s Complement to Decimal**

We know how to convert a decimal value to signed 8-bit binary two’s complement, but how about the other way around?

It follows a pretty similar procedure to converting decimals to two’s complement. Let’s take as a two’s complement binary number and try to convert it to its decimal counterpart of .

First, we want to invert the bits. All 0’s become 1’s, all 1’s become 0’s:

Now, because we’re going from two’s complement to decimal, *subtract* 1.

Convert that to decimal, and you get Since the original number *was* negative, slap a negative sign on there to get as your final answer.

For more examples on converting two’s complement to decimal, check out [this](http://sandbox.mc.edu/~bennet/cs110/tc/tctod.html) link.

**Bit Shifts**

Bit shifts, as the name implies, shifts the positioning of the bits on binary number either to the left or to the right.

**Left Shift**

Left shifts, expressed as “<<”, moves each binary digit to the left.

In the example above, we’re given the binary representation for 1. We then shift all the bits left by 2. This results in new binary number or 4 in decimal.

The most significant bit (the bit with the greatest value, aka the one on the very left) is lost.

In thisexample, we’re given the binary representation for 22. We then shift all the bits left by 3. This results in the new binary number , or 176 in decimal.

Since we’re shifting left by a power of 2 each time (because we’re in base-2), we can just multiply the current number by *n* being the numerical amount we’re shifting.

For example, look at the first problem:

And the second:

**Right Shift**

Right shifts, expressed as “>>”, moves each binary digit to the right.

You may have guessed that right shifting follows a similar principle to left shifting—only instead of multiplying by , we divide by instead.

When dividing a number that leaves a remainder, we drop the remainder, just like integer division:

We drop the remainder because we lose the least-significant bit when shifting right, as you can see with the example above. The “1” bit at the very-left is lost.

The written exam will usually give you shift questions like this:

System.out.println(12 << 2);

//Output: 48

Usually, they don’t give you a binary number to shift. They give you a decimal number and want you to waste time converting it to binary, shifting it, and then converting the shifted number back to decimal. Use the shortcut to bypass the process completely.

**Bitwise Operations**

Those who are familiar with finite state machines (something you’ll learn in your discrete math class) or bit fields should know how bitwise operators work. If you’ve never worked with bitwise operators before, it’s pretty simple—they’re operators used for binary, or to be more specific, each individual bit of a binary number.

**Bitwise AND (&)**

The bitwise AND is represented by the character ‘&’. Bitwise AND compares two binary numbers and performs a boolean AND comparison on each individual bit.

When looking at binary values in a boolean perspective, 0 is considered to be the boolean value *false*, and 1 is *true*. Here’s the truth table for bitwise AND for binary bits in case you don’t remember.

|  |  |  |
| --- | --- | --- |
| **Bit 1** | **Bit 2** | **Bit 1 & Bit 2** |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

The written exam will attempt to test your knowledge of bitwise AND (&) in the following manner:

//Example exam question: What’s the output?

System.out.println(11&9);

Notice how despite the bitwise AND, the numbers given to you are decimal (base-10) numbers. First, we’ll have to convert both those numbers to binary in order to perform a bitwise operation.

11 in binary = 1011

9 in binary = 1001

Now that we have the binary representation for the two numbers, we can perform our bitwise AND on each individual bit:

The result, , will be converted back to decimal, which is 9. Thus, the output of that print statement is 9.

System.out.println(11&9); //Output: 9

**Bitwise OR (|)**

The bitwise OR is represented by the character ‘|’. Bitwise OR compares two binary numbers and performs a boolean OR comparison on each individual bit.

|  |  |  |
| --- | --- | --- |
| **Bit 1** | **Bit 2** | **Bit 1 | Bit 2** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Let’s try two different numbers:

//What’s the output of the following print statement?

System.out.println(10|12);

We pretty much follow the same rules as with bitwise AND, except, obviously, we’re performing a bitwise OR instead.

10 in binary: 1010

12 in binary: 1100

Now that we have the binary representation for the two numbers, we can perform our bitwise OR on each individual bit:

The result, , will be converted back to decimal, which is 14. Thus, the output of that print statement is 14.

System.out.println(10|12);

//Output: 14

**Bitwise XOR (^)**

After the last two examples, you can probably guess how this goes. The bitwise XOR is represented by the character ‘^’. Bitwise XOR compares two binary numbers and performs a boolean XOR comparison on each individual bit.

|  |  |  |
| --- | --- | --- |
| **Bit 1** | **Bit 2** | **Bit 1 ^ Bit 2** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Let’s try this practice question:

//What’s the output of the following print statement?

System.out.println(7^15);

You know the drill… let’s convert those numbers to binary.

7 in binary: 0111

15 in binary: 1111

Now that we have the binary representation for the two numbers, we can perform our bitwise XOR on each individual bit:

The result, , will be converted back to decimal, which is 8. Thus, the output of that print statement is 8.

System.out.println(7^15);

//Output: 8

Bitwise operations by themselves aren’t tricky, but things get slightly more challenging when they’re thrown around with other operators like this:

System.out.println(3 << 2 & 8 \* 3 - 10);

Did you know that the order precedence, from highest to lowest, is multiplication/division, addition/subtraction, shift, and then bitwise operations? Oftentimes the exam will test you on your order of operations by giving you a whole jumbled mess of different operators. Make sure you know your operator precedence cold before exam day.

Oh, and the output for that print statement would be 12.

**Basic Arithmetic**

The second question of every written test will be a question on basic arithmetic. Usually, it involves asking you the output of a print statement, like the following example:

System.out.println(1 + 5.0 / 2 \* 3 – 5);

Some important things to keep in mind are that Java arithmetic follows the same order of operation rules as regular arithmetic, meaning that in the example above, multiplication and division are evaluated first, and then addition and subtraction.

With that in mind, we evaluate first. Since one of the operands is a *double* value (5.0), then we perform decimal division, giving us 2.5. Remember, if both operands are *int* values then we would perform *integer* division, not decimal division.

Next up is . Now we’re left with , which will give us 3.5. Thus, the output for the line of code above would be 3.5.

Let’s try another one:

System.out.println(3 + 8 / 2 % 3 + 5);

In terms of operator precedence, modulo is the same tier as multiplication and division. So, , and Then, you’re left with 3 + 1 + 5 = 9. The output for the line of code above would be 8.5.

Last but not least, around the region/state level, UIL tries to get a little bit tricky with arithmetic (since honestly, it’s pretty hard to make basic arithmetic difficult) by giving tricky modulo scenarios. We covered this in the *modulo* module (say that fast three times) but I’ll copy it here for convenience.

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 facts! The result is -3

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

**String Class Methods**

The fourth question of the written exam tests your knowledge on methods from Java’s *String* class. I’ve provided a list of methods for you to prioritize memorizing. They aren’t exhaustive but should serve as a decent outline for what you will encounter on the exam. Basic *String* methods like *.equals(), isEmpty(), toUpperCase(),* and *.length()* are omitted since you should (hopefully!) already know what they do.

**Common** (These methods appear frequently; you should definitely memorize these)

*substring()* – Returns the characters from the String based on an **inclusive** starting position and (optional) **exclusive** ending position.

String str = “Examination”;

System.out.println(str.substring(3)); //Index 3 = ‘m’

//Output: mination

System.out.println(str.substring(2, 5)); //Index 2 = ‘a’ Index 5 = ‘n’

//Output: ami



If you forget that the ending position is exclusive, the difference between the ending and starting values represent the character length! For the second print statement, begin at the starting position and count the first 5 – 2 = 3 characters you encounter starting from index 2.

*concat()* – Appends the String to the end of an existing String.

String str = “UIL”;

str = str.concat(“ Written Exam”);

System.out.println(str);

//Output: UIL Written Exam

String str1 = “Hi”;

str1 = str1.concat(“ I wanna”).concat(“ take the”).concat(“ exam”);

System.out.println(str1);

//Output: Hi I wanna take the exam

*charAt()* – Returns the character at some given index

String str = “Katherine”;

System.out.println(str.charAt(5));

//Output: r

System.out.println(str.charAt(str.length() – 3));

//Output: i

*equalsIgnoreCase()* – Returns whether or not two strings are the same ignoring case (boolean).

String str = “This world”;

String str1 = “tHIS wORLD”;

System.out.println(str.equalsIgnoreCase(str1));

//Output: true

*indexOf* – Returns the index of the *first* occurrence of the given character *(optional) starting at the specified index*

String str = “Mysterious”;

System.out.println(str.indexOf(‘s’));

//Output: 2

String str = “Mysterious”

System.out.println(str.indexOf(‘s’, 3));

//Output: 9

*lastIndexOf()* – Returns the index of the *last* occurrence of the given character *(optional) searching backwards starting at the specified index*.

String str = “Mysterious”;

System.out.println(str.indexOf(‘s’));

//Output: 9

String str = “Mysterious”;

System.out.println(str.indexOf(‘s’, 5));

//Output: 2

*toCharArray()* – Turns the *String* into an array of *chars*

String str = “mouse”;

char[] ary = str.toCharArray();

System.out.println(ary[0] + “ ” + ary[ary.length – 1]);

//Output: m e

*split()* – Splits the String into an array of Strings depending on given specifier.

String str = “What so not”;

String[] ary = str.split(“o”); //Split by ‘o’

System.out.println(Arrays.asList(ary));

//Output: [What s, n, t]

System.out.println(ary.length);

//Output: 3

The UIL exam may try and trick you by putting two of the same character consecutively. What if we try to split by that?

String str = “Possible”;

String[] ary = str.split(“s”);

System.out.println(Arrays.asList(ary));

//Output: [Po, , ible] //Yep, you get an empty space!

System.out.println(ary.length);

//Output: 3

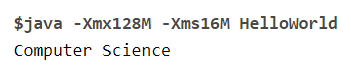
**Uncommon** (These methods appear occasionally, but it’s a good idea to memorize them)

*trim()* – Removes all whitespace (both leading and trailing from a String)

String str = “ Computer Science ”;

System.out.println(str.trim());

//Output:



*replace()* – Replaces *all* specified *chars* in a String, and returns a new String where the specified chars are replaced.

String str = “Good luck”;

System.out.println(str.replace(‘o’, ‘a’));

//Output: Gaad luck

*replaceAll()* – Pretty much the same as *replace()*, but instead of replacing *chars*, it replaces *Strings*.

String str = “Good luck at school”;

System.out.println(str.replaceAll(“oo”, “abc”));

//Output: Gabcd luck at schabcl

Yeah, not sure what that means either…

*replaceFirst()* – Similar to *replaceAll()*, except that it only replaces the *first* occurrence rather than all of them (you could probably infer from its very appropriate name)

String str = “Good luck at school”;

System.out.println(str.replaceFirst(“oo”, “abc”));

//Output: Gabcd luck at school

*regionMatches()* – Checks to see if two Strings are equal given some specified region.

This is a tough one because this method accepts *four* arguments: *toffset*, the substring where we begin matching the String, *other*, the name of the String we are matching it to, *ooffset*, the starting substring of the *other* String, and *len*, representing the number of characters we want to compare.

String str = “watermelon”;

String str2 = “mellifluous”;

System.out.println(str.regionMatches(5, str2, 0, 3));

//Output: true

In the example above, we check if the next three characters of *str*, starting at substring *5* (which will be the character ‘m’), matches the next three characters of *str2* starting at substring 0 (which will also be the character ‘m’). The next three characters of *str* **and** *str2* will be ‘mel’, so the output will be *true*.

Case sensitivity matters! For the example above, if “mellifluous” was capitalized, the output would *not* be true since we’re matching upper case characters with lower case characters. There may be an optional *fifth* argument that appears at the front of the argument list: a boolean *ignoreCase* that determines whether or not we care about case sensitivity. If it is *true*, then we can disregard casing.

*matches()* – Returns *true* if the given String matches the specified *regular expression*. We’ll talk about this in more detail in the *Regular Expression and the Pattern Class* module.

**Boolean Logic**

* + - **Simple Boolean Algebra**
    - **Advanced Boolean Algebra**
    - **Digital Electronic Diagrams**

**Simple Boolean Logic**

The fifth question on the written exam will ask you questions over boolean logic. Essentially, do you know your ANDs, ORs, XORs, and NOTs? And more specifically, do you know the order of precedence for each of those?

Here’s the good thing about boolean algebra—the answer will be in boolean format, meaning that the answer will either be *true* or *false*. So, for most questions involving boolean algebra, you’ll have 50/50 odds!

But let’s not try and rely on chance to get these questions right. Let’s talk a little about boolean logic. I’m going to skip the hard parts and just give you the relevant details you’ll need to know for the written exam (but for a fabulous, in-depth explanation of boolean logic and its history and use cases I heavily recommend [this](https://www.i-programmer.info/babbages-bag/235-logic-logic-everything-is-logic.html) website).

First, let’s think of booleans as *propositions*—a statement that is either *true* or *false*. For example, “water is wet” and “1 + 1 = 3” are *propositions* because they are unequivocally either *true* or *false*, but a statement like “apples taste better than oranges” is not, thus it is not a *proposition*.

Suppose we have two booleans, *p* and *q*. *p* will be *true*, and *q* will be *false*. We’re going to combine these two propositions and see if the combined proposition is *true* or *false*.

Let’s think of a *true* statement for *p*. How about “you are a human being?” If you’re reading this, chances are you’re probably a human being.

*p* = “You are a human being”

The value of the boolean *q* is *false*, so let’s think of a false statement. How about, “You have five arms?” If you’re reading this, chances are you probably don’t have five arms.

*q* = “You have five arms”

Okay, now I’m going to give you a proposition based on the two booleans above, and you’ll try to discern if the statement is *true* or *false*. Remember, it *has* to be one or the other.

You are a human being **AND** you have five arms

Hmm, well, you *are* a human being, that much is true. But five arms? Nah. This proposition is totally *false*. When you use **AND**, you’re basically saying the statement *preceding* it and the statement *proceeding* it both have to be true in order for the proposition to be true.

You are a human being **OR** you have five arms

With **OR**, at least one of the statements have to be *true* in order for the proposition to be *true*. You may not have five arms, but you are a human being. This proposition is *true*.

Simple enough, right? Let’s illustrate it out in a truth table with all the possible combinations of *p* and *q*:

|  |  |  |
| --- | --- | --- |
| **p** | **q** | **p && q** |
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | F |

|  |  |  |
| --- | --- | --- |
| **p** | **q** | **p || q** |
| T | T | T |
| T | F | T |
| F | T | T |
| F | F | F |

|  |  |  |
| --- | --- | --- |
| **p** | **q** | **p ^ q** |
| T | T | F |
| T | F | T |
| F | T | T |
| F | F | F |

If you’re still a little confused, keep substituting propositions for booleans. It’s a lot easier to understand when you can contextualize it. It takes a while, but you’ll remember it quick. I promise. Okay, let’s convert all this into code:

boolean p = true;

boolean q = false;

System.out.println(p&&q);

//Output: false

System.out.println(p||q);

//Output: true

System.out.println(p^q);

//Output: true

**The NOT (!) Operand**

The NOT operand, represented in Java as “!”, serves pretty much the same purpose that you’re familiar with: it negates the boolean, flipping the value.

boolean p = true;

boolean q = false;

System.out.println(!p||q); //NOT(true) && false

//Output: false

**Operator Precedence**

Here’s a trickier question: what will be the output here?

boolean p = true;

boolean q = false;

System.out.println(p || q && !p);

Alright, this question may give us some pause. What’s the order of operations for boolean operators?

1. NOT
2. AND
3. XOR
4. OR

(The full Order of Operations list can be found in the *Stuff You MUST Memorize* Module)

With that being said, let’s revisit the question above:

boolean p = true;

boolean q = false;

System.out.println(p || q && !p);

//Original: true || false && !true

//After NOT: true || false && false

//After AND: true || false

//After OR: true

//Output: true

Congratulations! Whatever UIL tries to throw at you for question five, you should be good.

**The Math Class**

Question 6 of the written exam tests you on specific functions in Java’s built-in Math class.

The use case of most Math functions, like *sqrt(), cbrt(),* and *pow()* can pretty much be inferred from its name alone. Still, it’s a good idea to make sure you know the slightly more obscure ones like *ceil()* and *floor()* as well as brush up on your trig (*sin(), cos(), tan(), asin(), acos(), atan()*).

Here’s a list of Math functions you should memorize:

*abs(int num)* – Returns the absolute value of *num*

System.out.println(Math.abs(-8));

//Output: 8

*pow(double base, double exponent)* – Returns the value of the base raised to the value of the exponent

System.out.println(Math.abs(5, 2));

//Output: 25

*sqrt(double num)* – Returns the square root of *num*

System.out.println(Math.sqrt(64));

//Output: 8.0

*cbrt(double num)* – Returns the cube root of *num*

System.out.println(Math.cbrt(64));

//Output: 4.0

*ceil(double num)* – Returns the smallest integer that is greater than *num* (it will return the *integer* as a *double* eg: 5 -> 5.0)

System.out.println(Math.ceil(3.14));

//Output: 4.0

*Note:* Be careful with negative numbers: *Math.ceil(-3.5)* will return *-3.0*, not *-4.0*. Remember the smaller a negative number is, the greater it is. This is a common trick!

*floor(double num)* – Returns the smallest integer that is less than *num* (it will return the *integer* as a *double* eg: 8 -> 8.0)

System.out.println(Math.floor(2.11));

//Output: 2.0

*Note:* Be careful with negative numbers: *Math.floor(-3.5)* will return *-4.0*, not *-3.0*. Remember the greater a negative number is, the smaller it is. This is a common trick!

*min(int a, int b)* (they can also be *double* values) – Returns the smaller of the two values

System.out.println(Math.min(3, 2));

//Output: 2

*max(int a, int b)* (they can also be *double* values) – Returns the greater of the two values

System.out.println(Math.max(15, 20));

//Output: 20

*round(double num)* – Rounds the double *num*

System.out.println(Math.round(8.5));

//Output: 9

*random() –* Returns a double greater than or equal to 0.0 and *less* than 1.0

System.out.println(Math.random());

//Output: 0.8224294876928601

**The Random Value Problem**

A very popular question involving the *random()* method looks something like this:

Which of the following values can *not* be generated in the following line of code?

System.out.println((int)(Math.random() \* 10));

1. 0 b) 3 c) 5 d) 8 e) 10

First of all, remember that *Math.random()* generates a double greater than or equal to 0.0 and less than 1.0 (. That means the value of *Math.random()* will never be 1.0. It could be 0.998 or something very close to 1.0, but never 1.0.

Essentially, that line above is multiplying the value generated from *Math.random()* and multiplying by 10, and then type-casting it to an *int*. So for example, if your random number is 0.31415926, multiplying that by 10 yields 3.1415926, and after type-casting to an *int*, the output will be *3*.

Let’s look at the answer choices and see if it’s possible to get that number:

1. 0

Sure, that’s possible. In the scenario that your randomly generated number is , multiplying that value by 10 and then truncating to an *int* will get you 0.

1. 3

Possible. In the scenario that your randomly generated number is , multiplying that value by 10 and then truncating to an *int* will get you 3.

1. 5

Possible. In the scenario that your randomly generated number is , multiplying that value by 10 and then truncating to an *int* will get you 5.

1. 8

Possible. In the scenario that your randomly generated number is , multiplying that value by 10 and then truncating to an *int* will get you 8.

1. 10

Not possible. To get a value of 10, you need a minimum value of 1.0, which can not be achieved with *Math.random() \* 10*.

This question is fairly common on the written exam. Enjoy your free points!

**Constants**

In terms of constants, you only need to know two: Math.PI is approximately 3.14, or , and Math.E is approximately 2.718. If you’re in trig or pre-calculus, you should be more than comfortable with any math questions they throw at you.

**The Supplemental Reference**

One last thing—let’s say you get a question that asks for this:

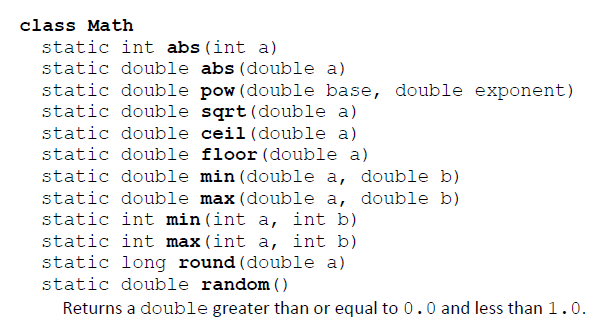
System.out.println(Math.pow(3, 2));

This question asks what the output of 3 raised to a second power is. Okay, well, 3 squared is 9. Simple enough. But let’s say the answer choices look a little something like this:

1. 3 b) 3.0 c) 9 d) 9.0 e) 18

With a quick glance, we can eliminate *a*, *b,* and *e*. We’re left with *c* and *d*. Hmm… does *Math.pow()* return an *int* or a *double*?

In case you forget, the second page of the UIL Exam, appropriately titled “Standard Classes and Interfaces – Supplemental Reference” has a list of… well, standard classes and interfaces as well as a few method signatures. If you forget the return type of a certain function, take a glance through the reference sheet, and see if you can find it.



According to this reference sheet, *pow* returns a *double*. Thus, *9.0*, aka answer choice *D* would be correct.

And if you knew this already, good for you! I think the proctor explains this before the beginning of every written test, but I usually tune them out. I can’t help it… it’s like one of those things that naturally just goes in one ear and out the other.

**ASCII**

ASCII (pronounced “ask-e”) is an acronym that stands for American Standard Code of Information Interchange (no, you don’t need to memorize that for the exam). It’s basically an encoding scheme that uses numbers to represent certain characters. Characters are assigned some number between 0 and 127. For example, the number “0” is assigned to value 48.

There is a difference between upper- and lower-case letters. Upper-case “A” is represented as 65 in ASCII, whereas lower-case “a” is represented as 97.

ASCII has been around for a long, long time now—even before the Internet! The original ASCII encoding scheme has a range of only 128 characters (7-bits, which might seem a little strange, but that’s a byproduct of the way computers were built back then), which really isn’t that much—it can pretty much only fit the English character set. If you know your encoding schemes, you’re probably more familiar with Unicode and its encoding types like UTF-8, which can display a lot more characters than that of English. Fortunately, UIL only requires you to memorize ASCII values of letters and numbers, nothing more. Still, that was a fun little history lesson to introduce you to what this module is all about.

**Display the ASCII Value of a Character**

First, let’s talk about how we can get Java to display the ASCII value of a character. The answer is pretty simple: just type-cast it to an *int*.

char letterGrade = ‘A’;

We have our character. Now, let’s create an *int*.

int letterASCII = (int)letterGrade;

The ASCII value of ‘A’ is 65. Thus, the value of *int letterASCII* is 65.

System.out.println(letterASCII);

//Output: 65

//Quicker way if you don’t plan on using the *int* to do anything

System.out.println((int)letterGrade));

//Output: 65

In Java, *chars* and numbers (both *ints* and *doubles*)are compatible with each other—that means you can add and subtract numbersand *chars* together (you could multiply and divide them too, though I don’t know why you would). When you add a *char* and a numbertogether, you’ll be actually adding the ASCII value of the *char* with the number.

System.out.println(‘A’ + 5);

//’A’ in ASCII has a value of 65 -> 65 + 5 -> 70

//Output: 70

System.out.println(‘a’ + 5);

//97 (ASCII value of ‘a’) + 5 = 102

//Output: 102

System.out.println(‘K’ + 2 – 3.5);

//75 (ASCII value of ‘C’) + 2 – 3.5 = 73.5

//Output: 73.5

To turn a number into the corresponding character associated with its ASCII value, simply type-cast numerical value as a *char* (if your number is a decimal, it will be truncated down; *e.g: 73.5 -> 73)*. Keep in mind the entire expression needs to be type-casted to a *char*.

System.out.println((char)(‘A’ + 5));

//ASCII value of 70 = F

//Output: F

System.out.println((char)(‘a’ + 5));

//ASCII value of 102 = f

//Output: f

System.out.println((char)(‘K’ + 2 – 3.5));

//ASCII value of 73 (truncated down from 73.5): I

//Output: I

The UIL Exam doesn’t provide you with an ASCII character map for reference. You’re expected to memorize the ASCII values for numbers and letters (both upper and lower case). You don’t need to memorize any characters beyond that.

That might seem daunting, but before you begin cramming, there’s one little trick that will turn every single ASCII question into a free six points. If you’re astute, you may have already guessed it.

**The 48-65-97 Rule**

What if I told you that you only needed to memorize *three* ASCII values? Here they are!

**48** – The ASCII value for the number 0

**65** – The ASCII value for ‘A’

**97** – The ASCII value for ‘a’

You may have already picked up on it, but the reason you only need to memorize these three values is because numbers start at ASCII value 48, so the ASCII value 49 represents the number 1, the ASCII value of 50 represents the number 2, and so on, all the way up to ASCII value 57 for 9. The same logic for upper- and lower-case letters.

*What is the ASCII value for ‘h’?*

Since the ASCII value of 97 represents *‘a’*, we count starting from 97 until we reach **104**, which is the ASCII value of *‘h’*.

*What is the ASCII* value for *‘k’?*

Since the ASCII value of 65 represents *‘A’*, we count starting from 65 until we reach**75**, which is the ASCII value of *‘k’*.

I have never seen a single written exam question ask for any ASCII value beyond the scope of the 48-65-97 rule. You truly only need to memorize these three values to conquer any ASCII question.

**Sorting Algorithms**

Oh boy, sorting algorithms, everyone’s favorite topic. It’s a little tough to do sarcasm via just text on a page, but that was meant to be sarcastic.

So, what’s the big deal with sorting algorithms anyway? All we’re doing is sorting some data in a certain order (usually numbers from least to greatest or *Strings* in lexicographic order). The algorithms themselves, and the implementation even, isn’t really that difficult to visualize, either. Most of it is really just memorization.

**Bubble Sort**

This is one of the simplest (and most inefficient) sorting algorithms. All we do here is repeatedly swap the adjacent elements if they’re out of order.

Suppose we have an array (named *ary*)of five numbers and we’re looking to sort it via bubble sort:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 2 | 3 | 1 | 4 |

Starting from *ary[0]*, we check if its smaller than its adjacent element (*ary[1]*)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **5** | **2** | 3 | 1 | 4 |

Last time I checked, *5* is not smaller than *2*. That means these two values are out of order. Thus, we swap them:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *2* | *5* | 3 | 1 | 4 |

Now we move on the new *ary[1]*, and compare it with its adjacent element, *ary[2]*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | **5** | **3** | 1 | 4 |

Because 5 is not smaller than 3, we swap again:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | *3* | *5* | 1 | 4 |

Now we compare *ary[2]* with *ary[3]*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | **5** | **1** | 4 |

Because 5 is not smaller than 1, we swap again:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | *3* | *1* | *5* | 4 |

Finally, we compare *ary[3]* with *ary[4]*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 1 | **5** | **4** |

Because 5 is not smaller than 4, we swap again:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 1 | 4 | 5 |

We’ve successfully brought the largest number to the last index of the array, where it belongs, but notice that the entire array is not sorted yet. Thus, we must go back again, to *ary[0]* and repeat this whole process again until the entire array is sorted.

With that being said, we begin the second iteration:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2** | **3** | 1 | 4 | 5 |

2 is smaller than 3, so we don’t swap anything here. This is fine as is, for now.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | **3** | **1** | 4 | 5 |

3 is not smaller than 1, though, so we should swap the two.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | *1* | *3* | 4 | 5 |

Let’s keep going:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 1 | **3** | **4** | 5 |

3 is smaller than 4, so everything is good, no need to swap. We’ve already established 5 as the largest number, so there’s no need to compare those. Thus, we are now at the end of the array, but its isn’t completely sorted yet. Time to start back at *ary[0]* for iteration number 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2** | **1** | 3 | 4 | 5 |

2 is not smaller than 1, so we swap them:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *1* | *2* | 3 | 4 | 5 |

Let’s move on:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | **2** | **3** | 4 | 5 |

2 is smaller than 3, so this is fine. We’ve reached the end of the array, and now the array is sorted. We’re finished.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |

That was a whole lot of fun, wasn’t it. It took 3 iterations to sort it all out, but the worst-case scenario is actually *n – 1* iterations, with *n* being the number of elements in the array (aka the array length). We got lucky and did it in 3, but there are scenarios where it may take 2, or 4, or 1 iteration if you’re lucky.

**Bubble Sort Code**

There are several code variations of bubble sort, but you will never be asked to ever program a bubble sort, only to identify one. If you remember the core principles of bubble sort (swap till the end), it won’t be a difficult task.

//Bubble Sort Algorithm Example

public static void bubbleSort(int[] ary)

{

for (int i = 0; i < ary.length - 1; i++)

for (int j = 0; j < ary.length – i – 1; j++)

if (ary[j] > ary[j+1])

{

int temp = ary[j];

ary[j] = ary[j + 1];

ary[j + 1] = temp;

}

}

Here’s an example of a prototypical bubble sort algorithm. We have a nested loop that compares adjacent elements. If the current element is greater than the one immediately proceeding it, then we perform a traditional swap algorithm. We will do a total of *ary.length – 1* iterations to guarantee that the array is completely sorted.

Bubble sort is not an efficient sorting algorithm. However, it’s one of the first sorting algorithms you learn, and it gives a good illustration on what to expect going forward

**Selection Sort**

Selection sort is an interesting sorting algorithm that relies on scanning the array for the smallest element. Once it finds that, it puts it at index 0, and continues the same process until the entire array is sorted.

Let’s use a visual example to illustrate this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 2 | 3 | 1 | 4 |

Let’s sort that above array. In selection sort, we first assume the number at index 0 is the smallest number. We then go through the array to see if any smaller number exists, replacing the current smallest number if needed as we go along. By the time we traverse the entire array, we will have the definitive smallest number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **5** | 2 | 3 | 1 | 4 |

We first assume 5 to be the smallest number (which I will represent with an underline). Any number smaller than 5 will, of course, be the new smallest number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | **2** | 3 | 1 | 4 |

2 is smaller than 5, so it becomes the newest temporary smallest number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 2 | **3** | 1 | 4 |

3 is not smaller than 2. The smallest number is still 2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 2 | 3 | **1** | 4 |

1 is smaller than 2. The new smallest number is 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 2 | 3 | 1 | **4** |

4 is not smaller than 1. The smallest number is still 1.

We’ve iterated through the array and guaranteed that the smallest number is 1. Since we know for sure that 1 is the smallest number, we can put it in index 0. We’ll do this by swapping its current position with whatever number is at index 0 (in this case, 5).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 5 | 4 |

You’ll notice, of course, that the entire array is not sorted. Thus, we iterate again. This time, we start at index 1, since we’ve already established that the value in index 0 is the smallest number. Now we find the second smallest number to put in index 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | **2** | 3 | 5 | 4 |

We start off with 2, the current smallest number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | **3** | 5 | 4 |

3 is not smaller than 2, so 2 remains the current smallest number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | **5** | 4 |

5 is not smaller than 2, so 2 remains the current smallest number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 5 | **4** |

4 is not smaller than 2, so 2 remains the current smallest number. And just like that, we’ve finished the second iteration through the array, and have identified the smallest unsorted number: 2. Luckily, it’s already in the desired location, so we don’t have to swap it with any number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 5 | 4 |

Now we have two values that are sorted, and three to go:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | **3** | 5 | 4 |

Our current smallest value is 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | **5** | 4 |

5 is not smaller than 3, so 3 remains the current smallest number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 5 | **4** |

4 is not smaller than 3, so 3 remains the current smallest number. These iterations are getting faster and faster. And just like last time, we don’t need to swap 3, it’s right where it should be.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 5 | 4 |

We’re almost done. We only have 2 numbers left:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | **5** | 4 |

5 is the temporarily smallest number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 5 | **4** |

Well, its reign of smallest number was very short-lived. 4 is smaller than 5, and our iteration is finished. We know that 4 is the 4th smallest number, and thus we swap its position with 5.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |

We only have one number left, so we can’t compare it with anything. By process of elimination, that’s our largest number, so we put it at index *ary.length – 1*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |

Our array is finally sorted.

**Selection Sort Code**

This is a pretty typical selection sort code. Usually when you have an index that you assign to a smaller value, it’s a pretty clear sign that you’re working with selection sort.

//Selection Sort Algorithm Example

public static void selectionSort(int[] ary)

{

for (int i = 0; i < ary.length - 1; i++) {

int index = i;

for (int j = i + 1; j < ary.lengtj; j++){

if (ary[index] > ary[j]) //Giveaway!!

index = j;

}

//Typical swap algorithm

int smallestNum = ary[index];

ary[index] = ary[i];

ary[i] = smallestNum;

}

}

**Insertion Sort**

Insertion sort is an efficient sorting algorithm for small arrays. In this sorting algorithm, we start by searching the array sequentially and comparing adjacent indices, swapping when needed.

Not a very great explanation, I know. Let me use a visual example instead. Say on the table there’s *n* cards. We’re going to pick up those *n* cards and sort them.

First, you’ll obviously take one card from the table and “insert” it in your hand. Then we’ll take another card from the table and put it to the right of the first card you picked up. Now you compare the two. If the card you just picked up is smaller than the card on its left, you’d swap the positions. Okay, sorted. Now we take another card from the table, put it on the right-most position and then compare with the card on its left. If the card you just picked up is smaller, you’d swap the positions. Then we still have to compare it with the left-most card (remember, you have three cards in your hand). If your new card is smaller, you’d swap positions. You’d continue to do this process until you’ve sorted your *n* cards.

Hopefully that explains things a little better. If you’re still confused, no worries. Here’s a visual example, we’ll use the same array as we did for bubble and selection sort:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 2 | 3 | 1 | 4 |

We start at index 1 and compare it with its left neighbor.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **5** | **2** | 3 | 1 | 4 |

2 is smaller than 5, so we swap the positions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 5 | 3 | 1 | 4 |

Okay, now we can move on to index 2, which contains the integer 3. We compare it with its left neighbor:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | **2** | **3** | 1 | 4 |

3 is greater than 2, so this is fine. We’ll leave it as is.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 2 | 3 | 1 | 4 |

Moving on to index 3, we’re comparing 1 its left neighbor: 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 2 | **3** | **1** | 4 |

1 is smaller than 3, so we swap.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 2 | 1 | 3 | 4 |

We’re not done with 1 yet—now we have to check it with its *new* left neighbor: 2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | **2** | **1** | 3 | 4 |

And since 1 is less than 2, we swap again.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 1 | 2 | 3 | 4 |

We’ve got one last left neighbor to check: 5.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **5** | **1** | 2 | 3 | 4 |

1 is smaller than 5, so we do one final swap.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 5 | 2 | 3 | 4 |

Phew, we’re done with that. Now, where were we again? We were at index 3 before we got into all these swaps, so let’s just move on to index 4. We compare the integer in index 4 with its left neighbor.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 5 | 2 | **3** | **4** |

4 is greater than 3, so no change is needed. So ends our first iteration.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 5 | 2 | 3 | 4 |

We’ve guaranteed the smallest value in index 0, so there’s no need to compare it with index 1. Thus, in the next iteration, we can start by comparing index 2 with index 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | **5** | **2** | 3 | 4 |

2 is smaller than 5, so we swap:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 5 | 3 | 4 |

Moving on:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | **5** | **3** | 4 |

3 is smaller than 5, so we swap:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | **2** | **3** | 5 | 4 |

3 is greater than 2, so we don’t swap.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | **5** | **4** |

4 is smaller than 5, so we swap:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | **3** | **4** | 5 |

3 is not smaller than 4, so we don’t swap.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |

Luckily for us, it only took us two iterations to get it done. Insertion sort works best with small arrays… it would be quite the pain to have to do this with 1,000 or even 100 numbers.

**Insertion Sort Code**

Not much to explain here, but if you happen to come across a *while* loop or a nested *for* loop that seems to embody the logic of “visit index – 1 and compare to see if your value is smaller,” chances are the mystery sorting algorithm is insertion sort.

//Insertion Sort Algorithm Example

public static void insertionSort(int[] ary)

{

//We start by comparing index 1

for (int i = 1; i < ary.length; i++)

{

int valueToMove = ary[i];

int j = i – 1;

//Here’s where we visit our left neighbors

while (j >= 0 && ary[j] > valueToMove)

{

ary[j + 1] = ary[j]

j--;

}

ary[j+1] = valueToMove;

}

}

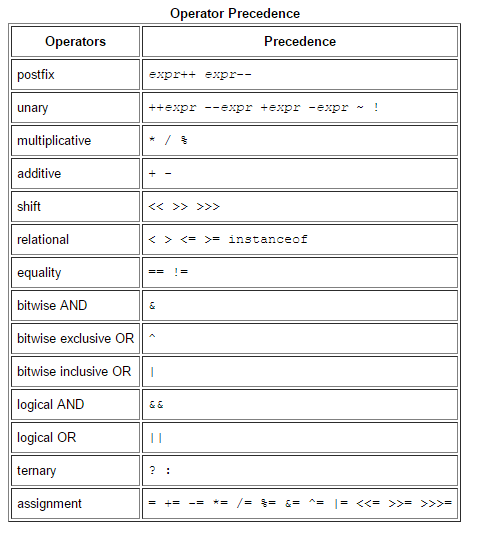
**Stuff You MUST Memorize**

I don’t wanna say it, but there are some things that you’re going to have to straight-up memorize if you’re going for a top tier score. Unfortunately, there’s no shortcut around it; you pretty much either know it or you don’t.

This is the list of things that you MUST memorize. I *guarantee* every single written exam you take will test at least one of the concepts mentioned below.

**Order of Operations**

You may know your PEMDAS, but do you know which comes first between << (left shift) and & (bitwise AND)? Knowing your basic arithmetic order of operations isn’t enough to nab some of the trickier questions on the exam. Here’s the full list of operator precedence that you need to know for those pedantically pesky problems!



**Primitive Data Types**

Hopefully by now you have the eight primitive data types memorized (if not, we have a module on *Primitive Data Types* to jog your memory). Unfortunately (and you’re going to notice that this is gonna be a common trend), that won’t be enough to nab you points on the more specific questions. You’re going to have to be fearlessly fastidious to fare favorably on these finicky fact-checking questions!

You’ll have to know both the size (usually in bits), as well as the min and max values they can hold.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size** | **Value Range** |
| byte | 8 bits | -128 to 127 |
| short | 16 bits | -32,768 to 32,767 |
| int | 32 bits | -2,147,483,648 to 2,147,483,647 |
| long | 64 bits | **Beyond the scope of the written exam** |
| float | 32 bits | **Beyond the scope of the written exam** |
| double | 64 bits | **Beyond the scope of the written exam** |
| boolean | 1 bit | *true* or *false* |
| char | 16 bits | Character, letter, symbol, or ASCII value |

(If you’re curious on the freakishly large numbers that a *double,* *float,* or *long* can hold, tap [here](https://cs.fit.edu/~ryan/java/language/java-data.html))

A possible way they might ask you for the size of a primitive type (in bits) is through the SIZE field found in the wrapper classes of primitive types.

System.out.println(Integer.SIZE);

//Output: 32

System.out.println(Double.SIZE);

//Output: 64

Ok, that’s a lot of information, I know. Before you try and start memorizing anything, here’s a few tested tips that I can give you that will hopefully save you some time.

* Don’t be scared if the question asks you the size of a primitive data type in *bytes* instead of *bits*. 8 bits = 1 byte, so if you know that an *int* has a size of 32 bits, just divide 32 by 8 to get 4 bytes!
  + There’s two ways they can test your knowledge on that: either they straight up ask you (What is the size of an *int* in bytes?), or they might use the BYTES field found in the wrapper classes of primitive types.

System.out.println(Integer.BYTES);

//Output: 4

System.out.println(Byte.BYTES);

//Output: 1

/\* Boolean.BYTES doesn’t exist since a boolean is 1 bit. I highly doubt that’ll be tested, just a little fun fact \*/

* For *bytes*, *shorts*, and *ints*, the **max** value is one smaller than the absolute value of the **min** value. If you know that, you only need to memorize either the min value or max value.
  + Again, there’s two ways they can test your knowledge on that: either they straight up ask you the min or max value of some primitive type, or they might use the MIN\_VALUE or MAX\_VALUE field found in the wrapper classes of primitive types.

System.out.println(Integer.MAX\_SIZE);

//Output: 2147483647

System.out.println(Short.MIN\_SIZE);

//Output: -32768

* + I know that the *int* min/max value is long. You don’t necessarily have to memorize every single digit of the max value of an integer, you just need to know that it’s 2 billion something and it ends with a 7. Usually when they ask you questions regarding the max value of an integer, they’ll have 2147483647 and 2147483648 as answer choices and test if you remember the difference between max and min.

**Tildes (~)**

Tildes (expressed as ~) are Java’s version of performing bitwise complements. If you want to look up the details of how to do that, by all means, go for it! I’ll just give you the shortcut to calculate the bitwise complement, but if you want to find out how it’s done behind the scenes, I highly recommend researching it! I didn’t fully appreciate binary until my computer architecture class in college, but understanding just how powerful it is gives you a profound appreciation for computer science.

Given an integer *x*, the formula to find the bitwise complement is

System.out.println(~10);

// -x – 1 -> -10 – 1 = -11

// Output: -11

System.out.println(~-8);

// -x – 1 -> 8 – 1 = 7

// Output: 7

Binary is interesting! We have a module on Two’s Complement that you should definitely check out if you haven’t already. It demonstrates another interesting application of how computers use binary.