**(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**

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

**First 15 Questions**

* **Number base concepts**
  + **Arithmetic**
    - **Addition**
    - **Subtraction**
  + **Conversions**
    - **Bin to Hex**
    - **Hex to Dec**
    - **Dec to Bin**
    - **Bin to Dec**
* **Simple math expressions**
* **Simple output**
  + **Print, println, print f(%d, %f, %s, \”, \\, \n)**
* **String class methods**
* **Simple boolean logic (And, Or, Xor, Not)**
* **Math class methods**
* **Simple variable expression**
* **Conditionals**
  + **if/else (Not ternary)**
  + **switch**
* **Simple output loop**
* **Array**
* **ArrayList**
* **Input concepts**
  + **Scanner and File classes**
* **Accumulation loop**
  + **Factorial**
  + **Summation**
* **Order of operation**
  + **Beyond basic with mod included**
* **Java specific data types concepts**
  + **Memory size, max, min, wrap around, complements**

**Advance Concepts**

* **Enumerated Data types**
* **Parsing**
  + **String.split**
  + **Integer.parseInt**
* **Regex**
  + **Pattern class**
* **Recursion, Dynamic programing**
* **Data Structures**
  + **Stacks**
  + **Queues**
  + **Binary Trees**
  + **LinkedList**
  + **Heaps**
  + **Hash Tables**
  + **Priority Queues**
  + **Graphs**
* **Sorting Algorithms**
* **Big O notation**
* **Two's complement**
* **Polish notation**
* **Lambda Expressions**
* **Boolean simplification**
* **Digital Electronics**
* **Generic collections**
  + **Collections**
  + **List**
  + **Set**
  + **Map**
  + **Stack**
  + **Queue, PriorityQueue**
  + **LinkedList, ArrayList**
  + **HashSet, TreeSet**
  + **HashMap, TreeMap**

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

There are a variety of use cases for this app:

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

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

**Primitive Data Types**

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

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

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

**Boolean**

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

//Declare a boolean with the value *true*

boolean isSunny = true;

//Declare a boolean with the value *false*

boolean isCloudy = false;

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

int num = 10;

boolean isNumGreaterThan5 = num > 5;

System.out.println(isNumGreaterThan5);

**Byte**

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

//Declares a byte with the value *12*

byte dozen = 12;

//Declares a byte with the value -1

byte num = 1;

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

byte num = 200; //Will not compile!

**Char**

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

//Declares a char with the value ‘A’

char firstNameInitial = ‘A’;

//Declares a char with the value ‘5’

char onesDigit = ‘5’;

//Declares a char with the value ‘\*’

char asterisk = ‘\*’;

**Short**

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

//Declares a short with the value 5000

short num = 5000;

//Declares a short with the value -100

short negativeNum = -100;

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

short num = 800000; //Will not compile!

**Int**

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

//Declares an int with the value 30

int num = 30;

//Declares an int with the value -8125123

int negativeNum = -8125123;

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

int pi = 3.14; //Will not compile!

For those who like visualizing large numbers:

Maximum value: - 1 = 2147483647

Minimum value: - = -2147483648

**Long**

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

//Declares a long with the value 31415926535

long bigNumber = 31415926535;

//Declares a long with the value 314159265358

long biggerNumber = 314159265358;

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

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

**Float**

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

Thus, floats represent numbers involving decimals.

//Declares a float with the value 5.39

long iMadeThisDecimal = 5.39f;

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

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

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

float wholeNumber = 5f;

**Double**

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

//Declares a double with the value 123456789.987654321

double iMadeThisDecimal = 123456789.987654321;

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

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

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

double wholeNumber = 5;

**Declaring and Initializing Variables**

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

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

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

int num;

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

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

int myAge;

int qwotiqwrog;

int q;

Anyways, back to the original code:

int num;

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

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

int myAge;

myAge = 21;

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

We can declare and initialize a variable in one line:

int myAge = 21;

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

//myAge is 21

int myAge = 21;

//Now myAge is 30

myAge = 30;

//Now myAge is 35

myAge = 35;

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

int examOneScore = 90;

int examTwoScore = 95;

//combinedExamScores is the sum of examOneScore and examTwoScore

int combinedExamScores = examOneScore + examTwoScore;

//This should output *185*

System.out.println(combinedExamScores);

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

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

double costOfApple = 0.75;

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

boolean isWeekday = false;

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

char letterGrade = ‘A’;

**Introduction to Strings**

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

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

So… what exactly is a *String*?

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

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

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

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

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

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

(remember, a space is also a char!)

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

Alright, cool. So, what does this mean?

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

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

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

String greetings = “Hello world!”;

System.out.println(greetings);

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

Here are some important things to remember:

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

**Print Statements**

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

Let’s get started:

In Java, we print using the command:

//Put whatever you want to print inside the parenthesis

System.out.println();

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

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

System.out.println(“Hello, World!”);

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

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

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

System.out.println(“Hello”);

System.out.println(“World!”);

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

Hello

World!

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

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

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

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

//Output: Hello, World!

System.out.print(“Hello, World!”);

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

System.out.print(“Hello”);

System.out.print(“World!”);

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

HelloWorld!

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

**Arithmetic Expressions and Type Casting**

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

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

int i = 10 + 5;

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

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

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

//The output should be 8

System.out.println(3 + 5 \* 3 – 10);

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

System.out.println(5 / 2);

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

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

5/2 = 2.5 = 2

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

Here’s some more examples:

//Will output 3 (3.75 -> 3)

System.out.println(15 / 4);

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

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

//Will output 0 (0.2 -> 0)

System.out.println(1 / 5);

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

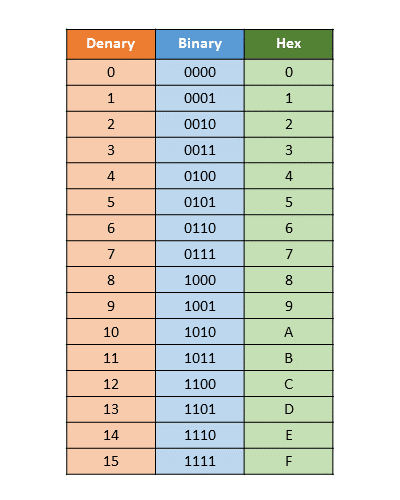
The UIL Segment

**Number Base Concepts**The first thing we have to learn is what a base is. We do calculations every day in base 10. That means numbers are {0,1,2,3,4,5,6,7,8,9} in any position. Binary on the other hand is base 2. That means it can only be {0,1} 2 options. Furthermore when doing other bases greater than 10 even though we do not have more than 10 numbers, we use letters. Hexadecimal fits this requirement as it is base 16. That means that 10 base 10 = A base 16. Hexadecimal can be {0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F}

Binary Breakdown



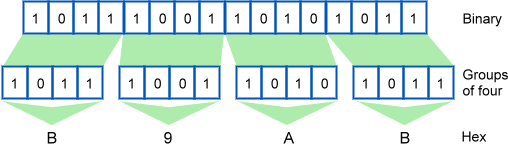
Hexadecimal Breakdown



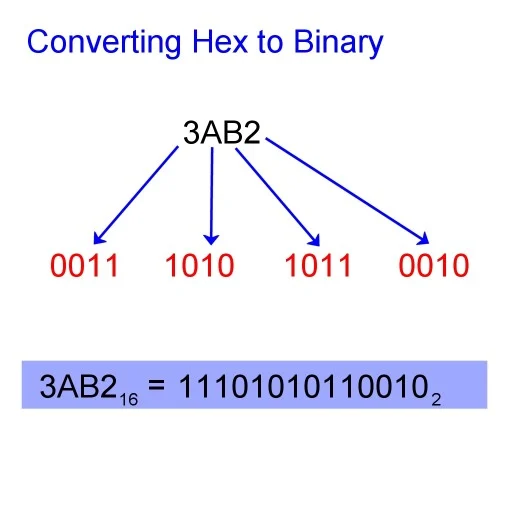
Conversion Binary to Decimal



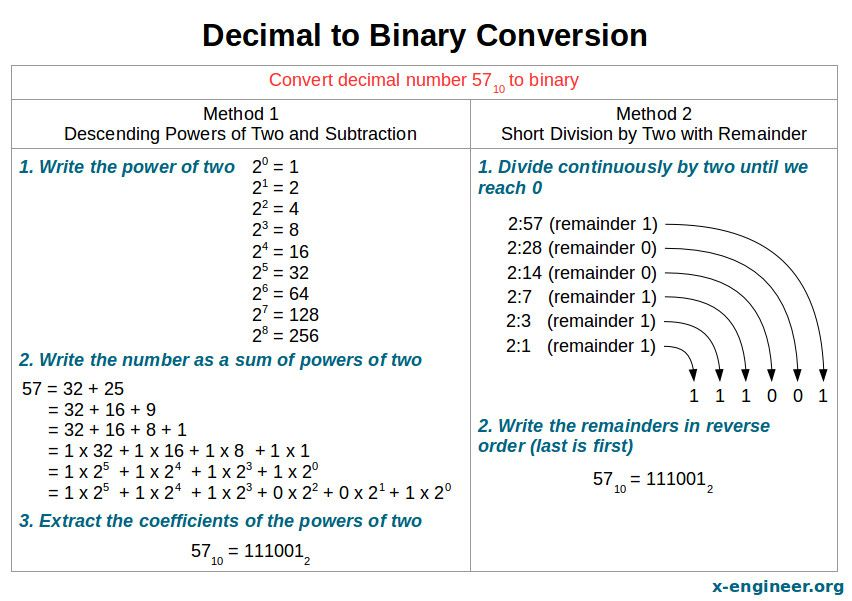
Conversion Binary to Hexadecimal



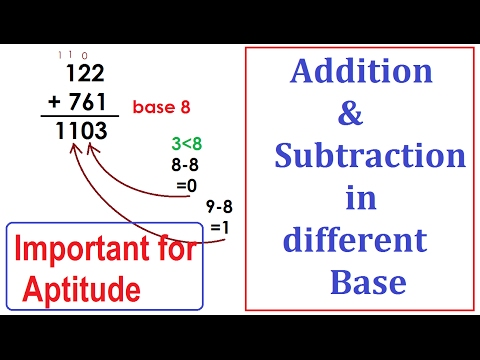
Conversion Hexadecimal to Decimal



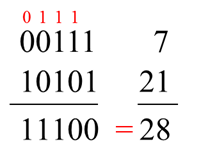
Conversion Decimal to Binary



Base addition

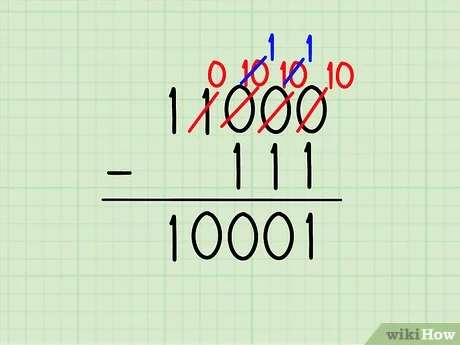


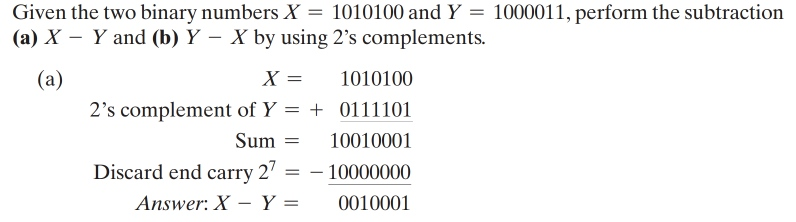
Binary addition



Binary Subtraction

There are 2 ways to do a binary subtraction

First one is like a normal subtraction method

The second one is using 2’s complement [Link to refer to the advance method] 

**Simple math expressions**

* **Simple output**
  + **Print, println, print f(%d, %f, %s, \”, \\, \n)**

There are 3 types of printing methods, System.out.print(), System.out.println(), and System.out.printf();

System.out.print(); Prints out on that line but if you were to call another print line it would show up following your last print

System.out.println(): Different from print, println adds a /n to ever print for you. That means everything is returned lower. It is equivalent to C++ cout << endl; or “\n”

Example

System.out.print(“This is my first line”);

System.out.print(“This is my second part”);

System.out.println(“I think it is time to move to another line”);

System.out.println(“Yes, I believe it is time”);

Output

This is my first lineThis is my second partI think it is time to move to another line

Yes I believe it is time

Printf has a few different parameters but first we need to go on to the format

System.out.printf(“String to be formatted”, each, variable, used);

Need to add printf here for that is what I am missing

* **String class methods**

Strings are very useful and below I will explain some methods that you must know. One thing to keep in mind is that a string is just an array of characters. It is an object.

* charAt - Returns the character at position x (“the”.charAt(1) returns h)
* compareTo - Compares two strings lexicographically (That is important) Example(a.compareTo(b))
  + If a is equal to b Then return 0
  + If a is alphabetically before b return value < 1
  + If a is alphabetically after b return value > 1
* compareToIgnoreCase - Same as compare to, and you guessed it, it ignores if a character is uppercase or lowercase
* concat - A fancy way of adding two strings together (“Fat”.concat(“ bob”) returns Fat bob
* contains - checks if a string contains the parameter of another string(AKA char sequence) (“that man jumped a fence”.contains(“fend”) returns )
* equals - Compares two strings and returns true if they are the same else false
* equalsIgnoreCase - Same as equals, just ignore Them “A”
* indexOf - returns the index of the first matching character (“that man”.indexOf(“t”) returns 0)
* lastIndexOf - returns the last index that it occurred (“that man”.lastIndexOf(“t”) returns 3)
  + Note on both index of I used an example with 2 same characters so I could use them both times
* length - returns the length of the string (“ThatString”.length() returns 10)
* replace - Replaces all instances in the string of a specific char with another char. If there is none it won’t change anything
* replaceAll - Replaces all matching substrings with a new string. Note that replaceAll is different than replace as replace only deals with char
* substring - This returns a string of position a to b of the original string. (“that is my line”.substring(4) returns “is my line”)
* toCharArray - Returns a char array of each character. Very useful
* toLowerCase - Turns the string to all lowercase letters (“THAT IS IT”.toLowerCase(); returns “that is it”)
* toUpperCase - Turns the string to all uppercase letters (“that is it”.toUpperCase(); returns “THAT IS IT”)
* trim - Removes extra spaces at the beginning and end of a string
* **Simple boolean logic (And (&&), Or (||), Xor (^), Not (!))**

The boolean logic deals with boolean which means true and false. There are 4 need to know operators (gates) that you need to know. Additional Extra ones are built off of these but for the most part this is it. Below is a table with all possibilities.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | A && B | A || B | A ^ B | !A |
| False | False | False | False | False | True |
| False | True | False | True | True | True |
| True | False | False | True | True | False |
| True | True | True | True | False | False |

Quick way to remember it.

* And is true only when both variables are true
* Or is true if either variable is true
* Xor is true if either variable is true, but not both
* Not is just the opposite
* **Math class methods**

When coding you cannot add a (Insert Square root sign) into your code just by typing. They came up with a fix with that by making it represented by “sqrt”. The math class can come in use as you can see if you typed “sqrt(9)” in google and it will return 3. We have listed below the math classes that you just have to remember

* ABS - (Anti-lock breaking system JK) Absolute ( |x| )
* Ceil - Round number up (3.1 turns into 4)
* Floor - Round down (3.6 turns into 3)
* Max - Greater value number (Max(2,5) returns 5)
* Min - Least Value number (Min(2,5) returns 2)
* Pow - Power/Exponent (pow(a,b) is a^b)
* Random - Returns a random double between 0.0 and 1.0
* Round - Guess what… It rounds the number
* Cbrt - The cube root of a number
* Sqrt - Square root (My favorite sqrt(x))
  + Extra
    - If argument is NaN or <0 = Return NaN
    - If argument is +0 or -0 = Return argument (+ or - 0)
    - If argument is +Infinity = Return +Infinity
* toDegrees - Returns an angle in radians unit to degrees unit
* toRadians - Returns an angle in degrees unit to a radian unit
* **Simple variable expression**
* **Conditionals**
  + **if/else (Not ternary)**

Could summarize from above the thoughts about if else. Same thing just UIL needs it now

* + **switch**

Switch statement should have been taught at the top also

* **Simple output loop**

For loop output with for(int i =0;i < 10; i = i+2) { System.out.print(i); }

**Array**

* **ArrayList**
* **Input concepts**
  + **Scanner and File classes**
* **Accumulation loop**
  + **Factorial**

Factorial is multiplying numbers from 1 to x number and denoted most of the time as x!. This can be written in a simple for loop [“Include the code for it here with the comments about how it is multiplying each time until it gets to that number. Can be in while loop or in for loop”]

* + **Summation**

Factorial is multiplying numbers from 1 to n numbers with increments of k value (k is typically 1) and denoted most of the time as Sigma X. [Need better format here for this] This can be written in a simple for loop (“Include the code for it here with the comments about how it is adding each time until it gets to that number. Can be in while loop or in for loop”)

* **Order of operation**
  + **Beyond basic with mod included**

We all know the order of operations is PEMDAS and it goes left to right. The issue we have now is the mod variable %. The mod variable divides and returns the remainder. From that reason it is said that mod is the same presidency as Multiple and Divide. So now it is PEMMDAS

* **Java specific data types concepts**
  + **Memory size, max, min, wrap around, complements**

Memory Sizes [Used from earlier]

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*.

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.

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 (‘ ’)

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?)

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.

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.

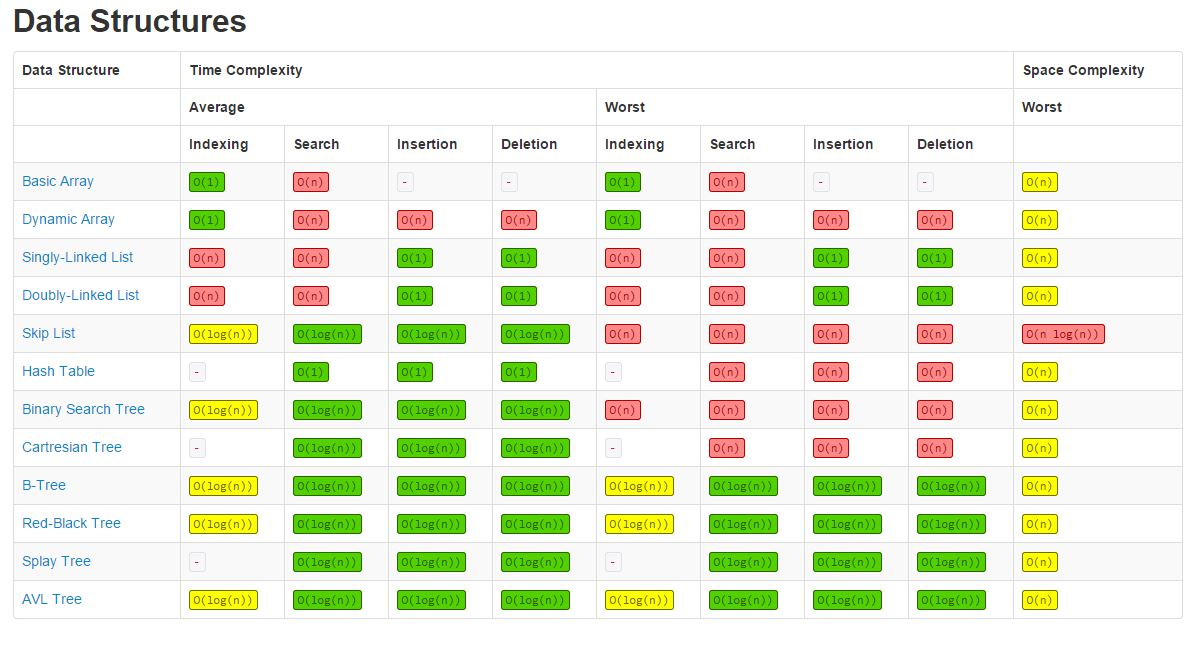
Float - Float comes from the term *floating-point numbers*, which is a fancy way of talking about numbers that are fractional. Some examples are π (3.1415…), the natural number e (2.71828), and 5.39, which is a decimal I made up on the spot. Floats end with *f*, to signify that it is a floating-point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

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.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Name | Boolean | Byte | Short | Int | Long | Float | Double |
| Size | 1 bit | 8 bits | 2 Bytes | 4 Bytes | 8 Bytes | 4 Bytes 7 decimal digits | 8 Bytes 16 Decimal Digits |
| Min | 0 | -128 | -32,768 | -2^31 (-2,147,483,648) | -2^63 | 1.17549435 E-38 | 2.22550738E-308 |
| Max | 1 | 127 | 32,767 | 2^31 - 1 (2,147,483,647) | 2^63 -1 | 3.40282346E+38 | 1.79769313E+308 |

**Advance Concepts**

**Big O notation**

****

Calculating the Big O notation of algorithms is based on the loops that it passes through.

From here on we are going to start referring to Big O notations

**-- Go through here on binary search and stuff like that..**

* **Enumerated Data types**
* **Parsing**
  + **String.split**
* split - splits the string into string[] creating a new entry at each split. Get from formatting more information
  + **Integer.parseInt**
* Integer.parseInt(x) - Turns parameter x from a string to an int there are more like
* Long.parseLong(x), Double.parseDouble(x), Float.parseFloat(x)
* **Regex**
  + **Pattern class**
* **Recursion, Dynamic programing**
* **Data Structures**
  + **Stacks**

Stacks are LIFO algorithms. That stands for Last In First Out. I always think of it as a stack of plates and you put an element on the top. Then the next time you grab a plate you grab the top plate.

Methods that are used

* Peek - Returns the top element’s value
* Pop - Removes the top element and return that value
* Push - Puts element into the stack
  + **Queues**

Queues are FIFO algorithms. That stands for Last In First Out where you think of a line is a queue. If you are the first person there you are the first one to leave. Sometimes I wish other aspects of life could be this way. The terminology is referring to the first element as being the head and the last element being the tail [Insert Photo Here]

Methods

* Peek - Returns the head element
* Element - Same as Peek but if Queue is empty returns an error
* Poll - Remove the head and return that value
* Remove - Same as Poll but if Queue is empty returns error
* Add - Puts element at the end of the Queue/ the Tail
  + **Binary Trees**
  + **LinkedList**
  + **Heaps**
  + **Hash Tables**
  + **Priority Queues**
  + **Graphs**
* **Sorting Algorithms**

**Two's complement**

Add a 1 to the front, flip all the bits and add 1 to the result

* **Polish notation**
* **Lambda Expressions**
* **Boolean simplification**
* **Digital Electronics**
* **Generic collections**
  + **Collections**
  + **List**
  + **Set**
  + **Map**
  + **Stack**
  + **Queue, PriorityQueue**
  + **LinkedList, ArrayList**
  + **HashSet, TreeSet**
  + **HashMap, TreeMap**