COMS W1004: Introduction to Computer Science and Programming in Java

Course Notes V. 1.0.1

By Griffin Newbold

Introduction

My name is Griffin Newbold, I am a TA for COMS W1004 for Spring 2022 under Professor Adam Cannon. I myself am part of the SEAS class of 2025 and plan to major in Computer Science. These course notes reflect a culmination of my efforts to provide extra resources to people taking this course. As the curriculum changes so will these notes, nothing will be removed but things will be added and objectives that are removed from the syllabus will be put towards the end and labeled as bonus content. While these were made primarily under Cannon's 1004 style, these should work fine for Blaer's 1004 as well. Currently the required textbooks are:

1. Invitation to Computer Science by Schneider and Gersting
2. Big Java: Early Objects by Cay Horstmann

I hope you enjoy your semester in COMS W1004, now let's begin!

Course Contents:

Click a link to skip to that Section

Java Building Blocks

To understand programming in Java, you must understand the basic building blocks that make up the language itself. The first thing that needs to be understood is the concept and application of ***variables***

Variables: Variables are how you store data in Java, when dealing with **primitive non-object types** (which will be discussed later) variables directly hold the data they are pointing to. When dealing with **non-primitive object types,** variables hold references to the data they are pointing to.

There are two distinct ways to declare and assign values to variables in java and depend on whether they are primitives or objects. They are as follows:

For primitive types it is as follows:

```
dataType variableName = valueOrExpressionOfSameType;
```

For object types it is as follows:

```
ObjectType variableName = valueOrExpressionOfSameType;
```

We will see concrete examples of the preceding two statements later on but for now to remember variables and what they do think of the memory balls from Inside Out, the ball itself is the variable and the memory is the data we are storing in the variable.

Primitive Data Types: In Java when you are dealing with data, there are different forms that the data can take and these forms determine what type of data can be stored. While there are many primitive data types (which I will still list for the sake of consistency the ones in bold are the ones most relevant in the actual class and the ones in italics will be relevant in an actual program):

1. **byte:** stores 8 bits of data and can hold integer values ranging from -128 to 127
2. char: stores 16 bits of data and can hold single characters ex: 'a' must use single quotes
3. ***boolean*:** stores 8 bits of data and is used to hold one of two values true or false
4. short: stores 16 bits of data and can hold integer values ranging from -32768 to 32767
5. ***int*:** stores 32 bits of data and can hold integer values ranging from -2147483648 to 2147483647
6. **float:** stores 32 bits of data and can hold floating-point values
7. ***double*:** stores 64 bits of data and can hold floating-point values
8. long: stores 64 bits of data and can hold integer values -9.22E+18 to 9.22E+18

The following is an example of how to declare a variable named myFirstInt as type int and we will assign it a value of 1004:

```
int myFirstInt = 1004;
```

We can then reassign myFirstInt to have a different value the following way:

```
myFirstInt = 1661;
```

Notice that we do not have to put the dataType when we are reassigning the variable to a different value, this is because the variable already exists you only need to put the dataType when you first declare the variable. To illustrate this the following code is identical in what it does:

```
int mySecondInt; //declaring the variable
mySecondInt = 2021; //assigning a value to the variable
int mySecondInt = 2021; //declaring and assignment
```

Now that we have covered primitive types we will now move onto Classes, Objects, and Constructors

Classes, Objects, Constructors

Classes are the main structural unit for Java programs, they house all the code for a particular file and the structure looks as follows (this should look familiar):

```
public class F2C {

     //main method
     public static void main(String[] args){
          //insert code here
     }
}
```

Files in java must follow the convention of ClassName.java so if the file you created was called 'HelloWorld.java' the name of the public class must be HelloWorld otherwise you will run into issues. When you think of a Class in relation to objects and constructors, think of the class as a blueprint for a product.

Time to make note of the main method, only briefly we will come back to it later when we talk about methods, in order for you to run a .java file directly it must have a main method, otherwise the JVM won't know where to start.

Objects are instances of a class, so think of an object as the final product that was designed using the blueprint, it therefore is an instance of that blueprint. The blueprint itself doesn't change but the specific attributes of the final product might be slightly different. For example you may have a red car or a blue car that were both designed using the same blueprint. Objects need to be instantiated into existence and can be done in the following way:

```
ClassName variableName = new ClassName(arguments);
```

Let us break down what it means to instantiate an object using the above syntax with something you are hopefully familiar with by now: The Scanner

It is important to note that the Scanner class is not available for use off the bat in java and you'll need to use a special statement to gain access, we will discuss that after we conclude with Objects and Constructors.

The Scanner is in the class named Scanner so in order to instantiate an object of type Scanner the first part of the syntax will be (note I will be naming the Scanner sc):

```
Scanner sc
```

That is the left hand side of the assignment out of the way, now time for the right hand side with only a handful of exceptions when you're dealing with objects you will always use the new keyword (since you are creating a new instance of an object) For Scanner it will be as follows:

```
= new Scanner(System.in);
//combining the previous two segments yields the following

Scanner sc = new Scanner(System.in);
```

It is important to realize that just like with primitive data types the declaration and assignment can occur on separate lines as follows:

```
Scanner sc;
sc = new Scanner(System.in);
```

For the Scanner class, it requires an argument in order to function properly, for now the only argument needed is System.in this allows for input to be collected from the user at the terminal during runtime.

Now is the perfect time to discuss constructors, constructors are special methods that are used to construct an object and are called immediately

after the new keyword and can take any amount of arguments as needed. They are formatted in the following way:

```
public ClassName(Param1, Param2,...){
     //insert code here
}
```

It is important to note that there is NO return type on constructors, we will cover return types more when we talk about methods in Session 2, when you think of a constructor think of the assembly line process in a factory, it constructs the final product (Object). To recap here is how you should view Classes, Objects, and Constructors metaphorically:

The Class is the blueprint … the design of the final product if you will. Once you have your design, you need to assemble the product on an assembly line, this is the Constructor it *constructs* the object and at the end of it all, you have your final product the Object that has just been constructed.

Import Statements

In Java there are classes that can provide functionality to our programs that are not included in the java.lang package (the set of classes that java imports automatically, no explicit declaration) the import statement syntax is as follows:

```
import java.util.Scanner;
```

Sometimes, you will need multiple classes from a package so you can use the wildcard "*", this allows you to import all the classes within a package and can be used in the following way:

```
import java.util.*;
```

More Java Fundamentals

Strings: Strings are a special data type in Java, they are not a primitive type and they are objects but they can be declared like primitive types and objects. Strings can be declared in Java using either method as follows:

```
String myFirstString = "This is my first String.";
String my2ndString = new String("This is my Second String");
```

Since Strings are objects there are methods that can be called upon the Strings we created, these methods will be covered in detail in a bit but for now there are a few things to know about Strings:

1. Strings are simply an array of chars meaning the following is the same as the first String object:
   char[] data = {'T','h','i', 's'....};
   String myFirstString = new String(data);
2. Strings are immutable; you cannot change a String object once you make it, if you do concatenation on a String it creates a new String Object.
3. String manipulation is probably the most common thing you will do in Java

Methods in the String Class

The String Class has many classes, most of which are irrelevant for this class but we will cover the most used ones and the ones we need to know

```
//used for String comparisons, returns a boolean
equals(Object obj);
//used to find the index that a substring occurs at within a
//larger String, returns an int
indexOf(String str);
//takes the entire string to UPPERCASE
toUpperCase();
//takes the entire string to lowercase
toLowerCase();
```

```
//used to find the length of a string, returns an int
length();
//used to create strings from the contents of a string,
//returns a new String object
substring(int start, int end);
//used to see if a substring is found within a larger String
contains(String str);
//used to get the specific character at a distinct index in
//a string
charAt(int index);
```

Methods in Java

In Java it is really sloppy and unconventional to write all of your code in the main method so you can write your own methods to abstract away the details of the code. The formal definition of a method is as follows:

```
<access specifier> <modifier> returnType
methodName(argumentList){
     //code goes here
}
```

Here is a proper example of creating a method by using the most used method that is required for Java programs to run: The Main Method:

```
public static void main(String[] args){
     //code goes here
}
```

Let's break it down, in this case "public" is the access specifier, meaning it can be accessed by any class that wants to use it. "static" is the modifier in use here, for now do not worry about what it means we will cover that when we talk about object oriented programming. The return type is "void" meaning this method returns nothing, the name of the method is "main" and the argumentList is a single String array called args.

With methods there are two key things that we can do: ***override and overload.*** We will talk about overriding methods when we discuss object oriented programming. For now we will stick to overloading methods. Let us assume that for some reason you wanted two versions of a method called "add" that simply adds values together so let us assume that your original method looks as follows:

```
public int add(int a, int b){
     return a + b;
}
```

You want a method, also called add, that can take three integers as input as well keeping the standard two integer method as well, well in this case we find the first way you can overload a method and that is by adding more parameters to the argument list, that being said the following is legal in Java:

```
public int add(int a, int b, int c){
     return a + b + c;
}
```

The other way to overload methods in Java would be to change the data types of the parameters that being said the following is legal in Java:

```
public double add(double a, double b){
     return a + b;
}
```

You can of course change both the number of parameters and their data types but it is never enough to just change the return type of the method, the following is not legal in Java and will not compile:

```
public double add(int a, int b){
     return a + b;
}
```

Repetition Structures (Loops)

In Java there will be times where you will need to repeat a set of instructions. Rather than copying and pasting the instructions several times we can use a loop to repeat a set of instructions until a certain condition is met. There are 2 key types of loops in Java: The While Loop and the For Loop

The While Loop

The while loop is a repetition structure that repeats a set of instructions until a boolean condition is met. The syntax for a while loop is as follows:

```
while(booleanExpression){
     //code to run
}
```

Beware of creating infinite loops, you must make sure when you are programming that you make sure there is something in the code of your loop that eventually breaks you out of the loop.

The do-While Loop

The do-while loop is exactly like a while loop except that it is GUARANTEED to run at least once the syntax for a do-while loop is as follows:

```
do{
     //code to run
}while(booleanExpression);
```

Just like with regular while loops, be careful you do not create an infinite loop.

The For Loop

The for loop is a repetition structure that repeats a set of instructions a set number of times, it is the loop most commonly used to iterate through arrays. The syntax for a for loop is as follows:

```
for(initialization, booleanExpression, updater){
     //some code to run
}
```

Control Structures

In Java sometimes we need to do different things depending on what conditions have been met. There are two main types of control structures: If-else statements and Switch-Case statements, let us begin with the If-else statement and to start here is the traditional if statement:

```
if(booleanExpression){
     //code to run
}

//code here runs regardless of the conditional above
```

The above statement basically says if the boolean expression is true, run the code within the if statement. The code outside an if statement runs regardless of, but what if that was not the intent we wanted well now sounds like a perfect time to introduce the else statement. Else statements can only exist with an if statement. The syntax is as follows:

```
if(booleanExpression){
     //code to run if booleanExpression is true
}else{
     //code here now only runs if the above doesn't
}
```

Finally, what if we want to check multiple different conditions? Well there is an Else If statement to help us with that, you are not required to have a standard else with this but it is recommended. The syntax for it is as follows:

```
if(booleanExpression1){
     //code to run if booleanExpression1 is true
}else if(booleanExpression2){
     //code to run if booleanExpression2 is true
```

```
}else{
     //code here now only runs if the above doesn't
}
```

The next type of conditional is the Switch-Case statement. It is important to note that the Switch-Case only works with the following types: byte, short, char, and int primitive data types, the String class, and the wrapper classes for the previously mentioned primitive types (do not concern yourself with this for now). The syntax for a switch-case is as follows:

```
int month = 2;
String monthString;
switch (month) {
    case 1:  monthString = "January";
             break;
    case 2:  monthString = "February";
             break;
    case 3:  monthString = "March";
             break;

    default: monthString = "Invalid month";
             break;
}
System.out.println(monthString);
```

It is important to note the break statements, if you do not put the break statements, then if one of the cases executes all the ones below it will execute until you either hit a break statement or exit out of the switch statement, so be careful!

Before we can move on to the basics of object oriented programming it is important to talk about operators and their precedence in Java. The following is the list of operators that are relevant and their precedence, if they are on the same line they are evaluated from left to right. The further down the list you go the lower the precedence:

Operator Type	Symbols
Post-Unary	expression++, expression--
Pre-Unary	++expression, --expression
Other Unary Operators	-,+,~,!
Multiplication/Division/Modulus	*,/.%
Addition/Subtraction	+,-
Relational Operators	<,>,<=,>=. instanceof
Equation	==, !=
Short Circuit Logical Operators	&&, ||
Ternary Operator	booleanExpression ? expression1 : expression2
Assignment Operator	=, +=, -=, *=, /=, %=

Object Oriented Programming 101

Java by nature is an object oriented programming language, this means that the true scale of the things you can do in Java can only be done using objects. We saw last week we could create an object of a certain type in the following way:

```
ClassName variableName = new ClassName(argumentList);
```

We also talked about constructors and learned that they are special methods with no return type (not even void) There is a special keyword when talking about object oriented programming called the ***this*** keyword, it is used to refer to the current object, simply think to yourself “I want this object’s variable” etc You do not actually have to explicitly define a constructor, if you don’t then the default constructor is called and looks like the following, important to note once you define a constructor you no longer get the default constructor:

```
public ClassName(){
     super();
}
```

The super keyword will be covered in detail in a future session when more detailed discussion on object oriented programming is necessary.

This is also the perfect time to properly define the static modifier, anything with the static keyword means it belongs to the class not the object. So if you define a static method, you do not need to declare and instantiate an object to use it. You can simply just call the method directly. If a variable has the static method then it also belongs to the class, so if you were to use a method that changed the value of that variable it would change for all of the objects of that type present and future.

If you are using objects, then an important concept to know is the dot operator "**.**" This is the operator used to call methods on an object and access its attributes. See the following example:

```
int value = 450;
public void updateValue(){
     //code to run
}
obj1.updateValue();
obj1.value
```

Objects in Java can inherit methods and the likes from other classes that they extend, all classes implicitly extend Object. Think of this inheritance like with you, you inherit certain genetic traits from your parents and your parents from your grandparents and so on. Like previously mentioned certain methods are inherited by default, but what if you want your own implementation of that method? This is where method overriding comes in. To do this all you have to do is define your own method with the exact method signature as the one you are trying to override. It is as simple as that!

The Binary System

When you were growing up you were taught to count and perform mathematical operations with the Decimal System. It turns out there are many more numbering systems that exist, usually denoted as Base-# so Decimal is base 10 since there are ten digits used to express values, these being 0-9. When we started creating computers we needed a new system to work with so Base 2 or Binary was created and it is what Computers use to process data. As the Base 2 probably gives away there are only 2 valid digits for binary, those being 0 and 1. So the following is an example of a binary number:

10101101

Converting from Binary to Decimal and Vice Versa

An important skill to have is the ability to translate between binary numbers and decimal numbers and the reverse too. To go from binary to decimal you simply start from the right of the number and add up powers of two whenever there is a 1 in its place with the right most value being 2^0 and the power increasing every time we move to the left. Let's try a couple of examples!

First let's convert the following binary numbers to decimal:

10101101 -> 173

Starting from the rightmost bit and adding the powers of two where there is a 1 we get:

$$2^0+2^2+2^3+2^5+2^7 = 173$$

11000101 -> 197

Same process as the last one we get:

$$2^0 + 2^2 + 2^6 + 2^7 = 197$$

Now let's go from a Decimal Number to Binary! The process is a little different but still straightforward. First find the value n that gives closest power of 2 greater than or equal to your number so if 63 the value will be n = 6 if the value is 65 then n = 7, this value n denotes the number of bits your binary number will have and after this you find the biggest power of 2 that fits in your number and make its equivalent bit a 1 and continue this process until the number is 0. Let's try a couple examples:

117 -> 01110101 (the leading zero is only there to keep our 8-bit values consistent)

Using the process described above we first must find n, which in this case is 7 since 2^7 is the closest power of 2 that is greater than or equal to our number. The closest power of 2 that will fit into our number is 2^6 so we will turn the 7th bit from the right to a 1 117-64 is 53 with the next power of 2 being 2^5 so same process, the 6th bit from the right is turned into a 1 and 53-32 is 21 repeat until the value reaches 0 and you get the value above.

127 -> 01111111 (the leading zero is only there to keep our 8-bit values consistent)

Using the process described above we first must find n, which in this case is 7 since 2^7 is the closest power of 2 that is greater than or equal to our number. The closest power of 2 that will fit into our number is 2^6 so we will turn the 7th bit from the right to a 1 127-64 is 63 with the next power of 2 being 2^5 so same process, the 6th bit from the right is turned into a 1 and 63-32 is 31 repeat until the value reaches 0 and you get the value above.

Two's Complement

Originally when we first started using binary representation the leftmost bit denoted the sign of the binary number with a 1 denoting a negative number

and a 0 denoting a positive number. The problem with this is that there were now two different zeroes 1000 being -0 and 0000 being +0 which is fundamentally incorrect. One trick to getting around this was a system known as Two's Complement. Two's Complement is a simple concept and it's easy to grasp:

Start at 000 (or however many bits you have I am using 3 for the simplicity) the amount of values you can represent after the 0 is 2^3-1 which is 7 take that number divide it into 2 and give the extra value to the negative subset of values so we get the following:

000 -> 0 | 001 -> +1 | 010 -> +2 | 011 -> +3 | 100 -> -4 | 101 -> -3 | 110 -> -2 | 111 -> -1

In order to give the two's complement representation of a negative number, all we do is start from the positive version of the number in binary, then we invert all the bits and simply add 1. Let's try a couple of examples!

Give the Two's Complement Representation of -124
First we must write out the binary representation of 124 as practiced above:

01111100

Now we invert all the bits: 10000011 and then we add 1: 10000100 so our final answer is: 10000100

Give the Two's Complement Representation of -35
First we must write out the binary representation of 35 as practiced above:

0100011

Now we invert all the bits: 1011100 and then we add 1: 1011101 so our final answer is: 1011101

Sign-Mantissa Form for Fractional Numbers

Eventually we will have had to mention fractional numbers and whether or not they can be represented in binary and the answer is yes, there is just a bit more work involved. The first step of which is to convert the number to scientific notation: $\pm M \times B^{\pm E}$ We will detail this process using the value +5.75 and also assume that we will use 16 bits to represent the value with 10 bits allocated for representing the mantissa and 6 bits for the exponent.

Let's start with converting 5 to binary: 101 now to add on the fractional bit we use the exact same logic insert a binary point and continue on, except we now have the halves, quarters and eighths places rather than the tenths, hundredths and thousandths place

.75 = .11 since (½ + ¼ = ¾) so our final value for 5.75 is 101.11 but we do not want the decimal point there so we must normalize the number so the first significant digit is immediately to the right of the binary point

$$\begin{aligned} 5.75 &= 101.11 \times 2^0 \\ &= 10.111 \times 2^1 \\ &= 1.0111 \times 2^2 \\ &= .10111 \times 2^3 \end{aligned}$$

We now have our exponent which is 3 and we have our mantissa which is 10111 now using the placeholders described above with the leftmost bit of each fragment representing the sign we get the following:

0|101110000|0|00011

I have added the bars to express the different segments here, the first segment being the sign of the mantissa, in this case it is positive so it has a 0 in its spot. The second segment is the mantissa itself. All you do is plop the number down there and add zeros after it until there is a total of 9 bits in that segment. The third segment is the sign of the exponent which in this case is positive so a 0 is in its spot. The final segment is the value of the

exponent and this you treat like a regular binary value, starting from the right and moving left.

Circuits, Logic Gates and Truth Tables

Circuits are an integral part of Computer Engineering and theoretical Computer Science and as such it is important for you to understand how circuits work and how to construct them. Circuits have 4 key components, those being inputs, wires, logic gates, and the output.

Let's focus on logic gates first. There are plenty of logic gates within the realms of Computer Science and Computer Engineering but here are the three most important ones to know:

1. AND: Represented by && in Java
2. OR: Represented by || in Java
3. NOT: Represented by ! in Java

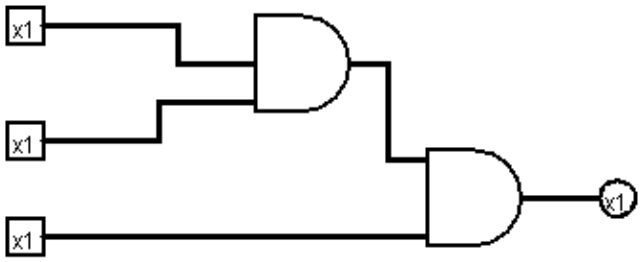
All of these should be familiar to you at this point as you've used all of them in prior assignments. All of this is part of Boolean Logic and an addition to Boolean Logic that we will discuss now are Truth Tables which allows us to display the result of boolean expressions. Let's try one now!

Give the truth table for the following:(x || z) && (y || !z) First write out all columns needed then break the boolean into chunks and solve them from there, the following is how you should approach this problem:

x	y	z	x || z	y || !z	(x || z) && (y||!z)
0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	1	1	1
0	1	1	1	1	1
1	1	0	1	1	1

1	1	1	1	1	1
0	0	0	0	1	0

Now we can go back to Circuit design, below is an example of a constructed circuit, let's dissect the components and then do some practice converting booleans to circuits and vice versa.



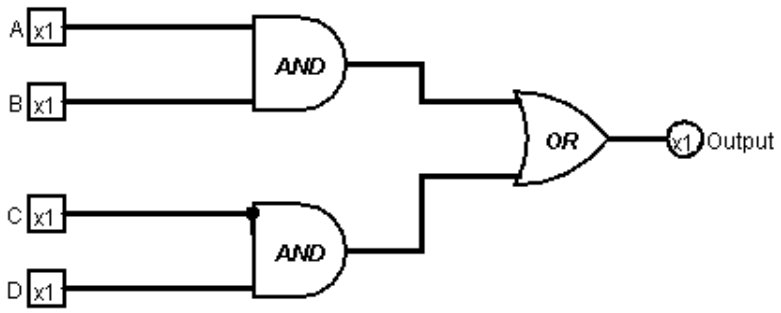
Equivalent Boolean Expression: ((A&&B)&&C)

In this schematic we can see 3 square nodes which represent our inputs, a single circular node that represents the output and two logic gates, these two gates are AND gates. Let's practice converting between circuits and booleans and vice versa:

Given the following boolean expression construct a circuit representation: (A&&B) || (C&&D)

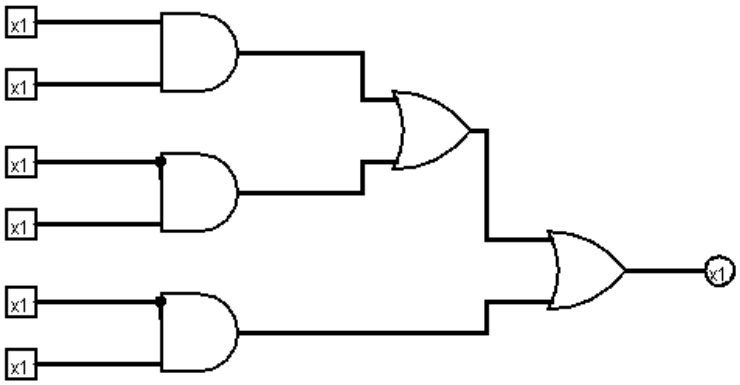
1. **First Step: Determine the number of Inputs you have (In this case we have 4: A, B, C, and D)**
2. **Second Step: Determine what logic gates you'll need (In this case we will need 2 AND Gates and an OR gate)**
3. **Third Step: Split into smaller circuits, construct and then combine results together**

Using this logic lets see what the final circuit will look like!

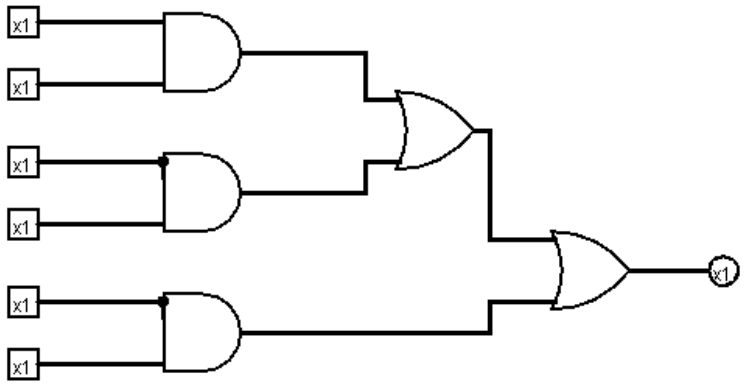


Equivalent Boolean: (A&&B) || (C&&D)

Now it is to go in reverse, given the following circuit provide the proper boolean expression:



Step 1.) Recognize there are 3 pairs of inputs: A,B,C,D,E, and F
Step 2.) Connect each pair with their logic gate: A&&B C&&D E&&F
Step 3.) Connect each connections with their logic gate: (((A&&B) || (C&&D)) || (E&&F))



Equivalent Boolean Expression: (((A&&B) || (C&&D)) || (E&&F))

Assembly Language

Assembly is a low level language that was used early on in the development of Computers and Computer Programming, there are many different versions of assembly language each with their own set of commands the list of which is below along with a snippet of code of what they do:

1. **LOAD X - Loads the value at address x to Register R**
2. **STORE X - Stores the value in Register R to address X**
3. **MOVE X,Y - Copies the contents of X and puts them in Y**
4. **ADD X,Y,Z - Performs X+Y and stores on Z**
5. **ADD X,Y - Adds X to the value at Y**
6. **ADD X - Adds X to the value currently in Register R**
7. **COMPARE X,Y - Compares the values at X and Y**
8. **JUMP X - Performs an unconditional jump to location X**
9. **JUMPGT/LT/EQ/GE/LE/NEQ - Performs a conditional jump**
10. **HALT - Program Stops**

These commands can allow you to write a lot of programs, and we will use these commands to translate Java code to Assembly language. Let's try a couple of examples. (v,w,x,y,z = 200,201.202,203, and 204 respectively)

Write out the Assembly Instructions for the following code segment: if(x < y) then set v to z else set v to y

Memory Location	OP Code	Address Field	Comment
50	COMPARE	202, 203	Compares the values of x and y
51	JUMPLT	54	Jump to Location 54 if x < y
52	MOVE	203, 200	Copies contents of y onto v
53	JUMP	55	Jump to location 55
54	MOVE	204, ,200	Copies the contents of z onto v

Write the Assembly code instructions for the following code segment: while(y >= z) set y to the value of w+y set z to the value of z+y End of Loop

Memory Location	OP Code	Address Field	Comment
50	COMPARE	203, 204	Compares the values of y and z
51	JUMPLT	59	Jump to Location 59 if y < z
52	LOAD	203	Register R now has the contents of y
53	ADD	201	Register R now has the contents of y+w
54	STORE	203	The value has been stored to y

55	**LOAD**	**204**	**Register R now has the contents of z**
56	**ADD**	**203**	**Register R now has the contents of z+y**
57	**STORE**	**204**	**The value has been stored to z**
58	**JUMP**	**50**	**Jump to location 50 to repeat loop**
59	**….**	…	**….**

Computer Organization: Von Neumann Architecture

The organization and structure of all modern computer systems are based on a theoretical model called the Von Neumann architecture. The Von Neumann architecture is based on the following principles:

1. The stored program concept, in which the instructions to be executed by the computer are represented as binary and stored in memory
2. Four major subsystems, memory, Input/Output, Arithmetic Logic Unit, and the Control Unit
3. The sequential execution of instructions, in which one instruction is taken from memory and passed to the control unit where it is decoded and executed.

Now each of the four major subsystems will be described, more information can be found in Chapter 5 Section 2 of the Invitation textbook.

Memory

Memory is the functional unit of a computer that stores and retrieves instructions and data. All information stored in memory is represented internally using binary. Computer memory uses an access technique called random access, you know it as random access memory (RAM) which has the following characteristics:

- The time it takes to store or retrieve contents of a single cell is the same for all cells in memory
- All accesses to memory are to a specific address, you must always store or retrieve a complete cell.
- Memory is divided into fixed-size units called cells and each cell is associated with an unique identifier known as an address which comprises unsigned integer values.

Memory is made up of cells that contain a fixed number of binary digits, the number of bits per cell is called the memory width and we denote this with a W the standard width for a cell nowadays is 8 with there being 8 bits in a byte! When considering how many possible addresses there can be in memory let's consider N to be the number of bits available to represent the address, that would mean you have 2^N-1 possible addresses you have to subtract 1 since 0 is considered a valid address. The memory unit contains two special registers, the MAR which holds the address of the cell to be accessed, and the MDR which contains the data value being stored or retrieved the size of the MDR are usually multiples of W usually being nW traditionally nW holds the values of 32 and 64 (hopefully these trivial values ring some bells)

Input/Output and Mass Storage

The input/output (I/O) units are the devices that allow a computer system to communicate and interact with the outside world as well as store information for the long term. Of all the components of a Von Neumann machine, the I/O and mass storage subsystems are the most ad hoc and the most variable. Input/output devices come in two basic types: those that represent
information in human-readable form for human consumption and those that store information in machine-readable form for access by a computer system. The former includes such well-known I/O devices as keyboards, both physical and virtual, screens, and printers. The latter group of devices includes flash memory, hard drives, DVDs, and streaming tapes.

Mass storage devices themselves come in two distinct forms: direct access storage devices (DASDs) and sequential access storage devices (SASDs).

A direct access storage device (DASD) is one in which requirement number 2, equal access time has been eliminated. That is, in a direct access storage device, every unit of information still has a unique address, but the time needed to access that information depends on its physical location and the current state of the device A magnetic disk stores information in units called sectors, each of which contains an address and a data block containing a fixed number of bytes. A fixed number of these sectors are placed in a concentric circle on the surface of the disk, called a track.

A sequential access storage device (SASD) does not require that all units of data be identifiable via unique addresses. To find any given data item, we must search all data sequentially.

One of the fundamental characteristics of many (although not all) I/O devices is that they are very, very slow when compared with other components of a computer.

The Arithmetic Logic Unit

The arithmetic/logic unit (ALU) is the subsystem that performs such mathematical and logical operations as addition, subtraction, and comparison for equality. The one-bit adder that was in the review session and the one-bit subtractor that was completed on the problem set last week, are just a fraction of what makes up the ALU.

The ALU is made up of three parts: the registers, the interconnections between components, and the ALU circuitry. Together these components are called the data path.

The Control Unit

The most fundamental characteristic of the Von Neumann architecture is the stored program—a sequence of machine language instructions stored as binary values in memory. It is the task of the control unit to (1) fetch from memory the next instruction to be executed, (2) decode it—that is, determine what is to be done, and (3) execute it by issuing the appropriate command to the ALU, memory, or I/O controllers. These three steps are repeated over and
over until we reach the last instruction in the program in your case this command is HALT

Together the Control Unit and the ALU construct what we all know as the Central Processing Unit or CPU

Class Design Overview

Class design is a very important aspect of programming as it allows us to organize and properly think out how we want to write our code before we even touch the keyboard. When I was in high school taking my first Computer Science course, my instructor told me to think twice and code once, and that is the mindset I want you to have going forward.

There are two key design principles that you should be aware of: Top-Down Design and Bottom-Up Design I will now give a brief description and advantages of each principle

Top-Down Design

In the top-down model, an overview of the system is formulated without going into detail for any part of it. Each part is then refined into more details, defining it in yet more details until the entire specification is detailed enough to validate the model. In top down design we perform the following actions:

- We break the problem into parts,
- Then break the parts into parts soon and now each of the parts will be easy to do.

Some of the key advantages of Top-Down Design are:

- Breaking problems into parts help us to identify what needs to be done.
- At each step of refinement, new parts will become less complex and therefore easier to solve.
- Parts of the solution may turn out to be reusable.
- Breaking problems into parts allows more than one person to solve the problem.

Bottom-Up Design

In this design, individual parts of the system are specified in detail. The parts are linked to form larger components, which are in turn linked until a complete system is formed.

Some of the key advantages of Bottom-Up Design are:

- You know each individual part works
- Redundancy is minimized by using data encapsulation and data hiding.

Java Specific Design

Regardless of which design methodology you decide to go with for your project there are some java specific design principles that you should follow, most of these are already laid out in the style guide on courseworks but this will be a brief overview

Be sure that all .java files contain a header at the top of the file detailing the author's name, the date, the filename, and a brief description of what the file's functionality is. For university students it is important to put your unique identification tag

When you open code blocks you use the '{' to denote the beginning of a block and the '}' however there are two key styles that you should follow when writing your code, I will detail each below but please choose one style and stick with it through the entire file to keep the file readable and to maintain consistency.

K&R Style (my personal preference) braces go attached to the preceding line:

```
for(int i = 0; i < arr.length; i++){
    //some code to run
}
```

Horstmann Style braces are detached from the preceding line:

```
for(int i = 0; i < arr.length; i++)
{
    //some code to run
}
```

Indentation and whitespace are important components of a program, if you are writing in a language like Python then you are required to have proper indentation and whitespace in order for the program to run in the first place. In java it is purely style, but style is important so it is relevant, use whitespace to separate 'thoughts' in your code so if you have 5 lines devoted to setting up a single variable put a white space afterwards to improve readability. For every codeblock you are deep into a program, there is one additional indentation that is needed.

Class names are written in UpperCamelCase; Variable and Method names are written in lowerCamelCase, please provide useful and informative names. Constants in Java should be written in CONSTANT_CASE All instance variables should have the private access modifier on them and have corresponding mutator and accessor methods as you deem necessary for the proper encapsulation of data and program functionality. All methods that you want the client to be able to call should have the public access modifier, if you do not want the client to have permission to call the method directly (for example helper methods) they should have the private access modifier.

Introduction to Arrays

An array in Java is a group of like-typed variables referred to by a common name. Arrays in Java work differently than they do in C/C++. Following are some important points about Java arrays.

- In Java, all arrays are dynamically allocated. (discussed below)
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++, where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered, and each has an index beginning from 0.
- Java arrays can also be used as a static field, a local variable, or a method parameter.
- The size of an array must be specified by int or short value and not long.
- The direct superclass of an array type is Object.

An array can contain primitives (int, char, etc.) and object (or non-primitive) references of a class depending on the definition of the array. In the case of primitive data types, the actual values are stored in contiguous memory locations. In the case of class objects, the actual objects are stored in a heap segment (heaps are not relevant information to you at this time so that is all that will be said).
To declare and instantiate an array in java it can be done in one of two ways

```
int[] arr = new int[5];
int arr[] = {0,1,2,3,4};
```

It is important to note that the [] can be placed after the type or after the name. It makes no difference, the first method to instantiate declares an int array that can hold 5 elements, the second method does the exact same thing except it gives initial values to each slot in the array. An important

thing to be able to do with arrays is to iterate through them we can do this one of two ways:
The first is with a traditional for-loop:

```
for(int i = 0; i < arr.length; i++){
        //some code to run
}
```

Using a traditional for-loop allows us to modify the data structure without running into errors. You can access elements at index i with arrayName[i] syntax.
The second way is with an enhanced for-loop otherwise known as a for-each loop

```
for(int i : arr){
  System.out.println(i);
}
```

When using the enhanced for-loop be careful not to modify the structure of the data structure you should only be accessing the elements using a for-each loop

To add items to the array you just specify the index in assignment:

```
arr[2] = 5;
```

This line means that we are assigning the value 5 to index 2 of the array

Introduction to ArrayLists

Now that we have covered arrays, which are static, we need to address the possibility that we may have an unknown amount of data, so an array which cannot change length past the initial allocation, an ArrayList can solve this issue because it is dynamic and can expand when needed.

The main caveat with ArrayLists is that primitive data types cannot be stored within the structure and the autoboxed object types must be used (we will cover this next time)

To declare and instantiate an ArrayList, you must have imported the java.util.ArrayList package and then you write the following style of code:

```
ArrayList<Integer> myArrayList = new ArrayList<Integer>();
```

The usage of 'Integer' on the right hand side is redundant and you actually do not need it meaning the following line of code is just as valid:

```
ArrayList<Integer> myArrayList = new ArrayList<>();
```

ArrayLists have dedicated methods for adding, removing, getting and setting elements within the Array, please go see the Important Classes and Methods document in the files section on Courseworks.

You can iterate through ArrayLists the same way as with arrays except you'd use the proper methods of the ArrayList class rather than the square bracket syntax style like you would for Arrays.

While I will not detail how the methods work here, the methods that are most important from the ArrayList class are: add(), get(), set(), and remove()

Array vs ArrayList

Here are some key things to keep in mind when you are deciding whether to use an Array or an ArrayList for your program:

1. Flexibility
    a. Arrays are static data structures
    b. ArrayLists are dynamic data structures
2. Primitive Data Type
    a. Arrays can store primitive data types and object types
    b. ArrayLists can only store object types
3. Adding Elements
    a. Arrays can insert elements via assignment
    b. ArrayLists can add elements via methods such as add()

Higher Dimensional Arrays

In the last section we discussed 1-Dimensional Arrays and their uses, now it is time to introduce the idea of multidimensional arrays. Hypothetically you should be able to have N-dimensional arrays but we will stick to 2 dimensions for now

When you think of multidimensional arrays, simply think an array of arrays like this:

```
int[][] a = [[1,2,3],[4,5,6]];
```

Java and other C-based languages are ***row major*** this means for indexing purposes the first index you specify is for the row i.e. to access the 2 we would use the following notation:

```
a[row][column] => a[0][1];
```

We can iterate through every element in this collection using a nested for-loop as follows, remember its rows first then columns:

```
for(int r = 0; r<a.length;r++){
     for(int c = 0; c < a[r].length; c++){
          System.out.println(a[r][c]);
     }
}
```

It is important for the bounds to be correct as it could lead to problems, so please use the notation as given above.

Autoboxed Types

Last section I mentioned that primitive data types like int, double and char have Object type versions of themselves. This is true and goes for all the primitive types, this is what allows us to store primitive data types inside more advanced data structures like Arraylists.

These autoboxed types are called as such because when you pass in a primitive to where only objects are accepted, Java automatically turns the primitive into an Object version of itself. These object types allow us to perform methods to ease ourselves when programming, Integer.parseInt() and Character.toString() are methods that you should most likely already be familiar with and many more methods exist in these classes for you to use.

Inheritance and Polymorphism

One of the key concepts in any Object Oriented programming language is Inheritance and Polymorphism. In Java inheritance is in the form of subclass-superclass relationships. This can be shown through more Java keywords. Consider the following class definition:

```
public class subClass extends superClass {}
```

We use the extends keyword whenever we want to show a relationship between two classes, you always extend the superClass. When you extend a class the subclass inherits all public methods from the superClass that can then be reused in the subClass. This however is not necessary, in the event you want a class to have distinct functionality from its parent in regards to an inherited method, you can simply override the method by redefining it in your own class. Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance. Polymorphism allows us to use inherited methods in different ways. For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats etc

Files and Exceptions

Now we want to be able to consider data from outside files. To create a file object in Java:

```
File fileName = new File("filePath");
```

We can then use this File Object to construct a Scanner:

```
Scanner sc = new Scanner(fileName);
```

We can then use a loop to iterate through the contents of a file:

```
while(sc.hasNextLine()){//some code to run}
```

If you want to write to a file you must construct a PrintWriter Object:

```
PrintWriter out = new PrintWriter("outputPath");
```

From there just .println methods to write to the file. Be sure to close the Scanner and PrintWriter when you are done processing the file prior to the program quitting

When you detect an error condition, you should throw the appropriate exception object using the throw keyword:

```
throw new IllegalArgumentException("This shouldn't happen");
```

Sometimes you will want to handle an exception before it reaches the user for these consider a try-catch block

```
try{
    //statements that might throw an exception
}catch(//the exception you want to catch){
    //what you want to happen when exception is
    caught
}
```

You can also attach a finally block to the try-catch block in order to put some code that executes no matter what, a finally block is not required in the structure however!

In Java there are two key types of exceptions: Checked and unchecked exceptions, checked exceptions can occur beyond your control while unchecked exceptions are your fault in the event that you are potentially dealing with an exception in the current method that you cannot handle add the throws clause to the method signature to tell the compiler you are aware of the exception and you expect termination to occur when the exception occurs.

```
public void readData(String filename) throws FileNotFoundException {
      //some code to run
}
```

Finally it is important to understand you can create custom exceptions the same way you
make custom classes just extend RuntimeException in you Class definition when you do it!

Computer Networks

WAN- a collection of local-area networks (LANs) or other networks that communicate with one another. A WAN is essentially a network of networks, with the Internet the world's largest WAN.

MAN- a network with a size greater than LAN but smaller than a WAN. It normally comprises networked interconnections within a city that also offers a connection to the Internet.

LAN- a collection of devices connected together in one physical location, such as a building, office, or home. A LAN can be small or large, ranging from a home network with one user to an enterprise network with thousands of users and devices in an office or school.

The Open Systems Interconnection (OSI) model describes seven layers that computer systems use to communicate over a network. There are seven layers to the model and attached are protocols associated with each layer:

1. Physical Layer: RS232, 100BaseTX, ISDN, 11
2. Data Link Layer: RAPA, PPP, ATM, Fiber Cables
3. Network Layer: IPv5, IPv6, ICMP, IPSEC, ARP, MPLS
4. Transport Layer: TCP, UDP
5. Session Layer: NetBIOS, SAP
6. Presentation Layer: MPEG, ASCH, SSL, TLS
7. Application Layer: HTTP, FTP, POP, SMTP, DNS

The extent of knowledge you should have over these is simply the layer names and some protocols

Dijkstra's Algorithm

Dijkstra's Algorithm is an algorithm that is used for finding the shortest distance, or path, from starting node to target node in a weighted graph
Here are the steps:

1. The very first step is to mark all nodes as unvisited,
2. Mark the picked starting node with a current distance of 0 and the rest nodes with infinity,
3. Now, fix the starting node as the current node,
4. For the current node, analyze all of its unvisited neighbors and measure their distances by adding the current distance of the current node to the weight of the edge that connects the neighbor node and current node
5. Compare the recently measured distance with the current distance assigned to the neighboring node and make it as the new current distance of the neighboring node
6. After that, consider all of the unvisited neighbors of the current node, mark the current node as visited
7. If the destination node has been marked visited then stop, an algorithm has ended, otherwise, choose the unvisited node that is marked with the least distance, fix it as the new current node, and repeat the process again from step 4.

This algorithm completes in $O(N^2)$ time

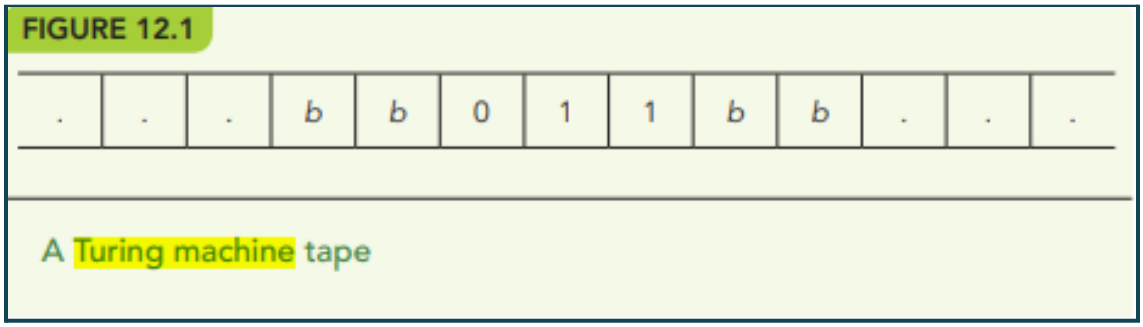
For purposes of this review guide, a fellow TA made a really good video explaining Dijkstra's Algorithm and how to go about solving problems that use it: [Dijkstra's Algorithm: An Overview]

Turing Machines

A Turing machine is a theoretical model of computation that includes a (conceptual) tape extending infinitely in both directions. The tape is divided into cells, each of which contains one symbol.

The Turing machine is designed to carry out only one type of primitive operation. Each time such an operation is done, three actions take place:

1. Write a symbol in the cell (replacing the symbol already there).
2. Go into a new state (it might be the same as the current state).
3. Move the "read head" one cell left or right.



The image above is an example of a turing machine tape. For purposes of Cannon's assignments you start on the leftmost blank spot. Turing machine instructions are formatted as such: (Current State, Current Symbol, Next Symbol, Next State, Direction). The Symbol is what is in the boxes, you will always start in state 1 (unless stated otherwise) and you will perform the given operations until you run out, or potentially run into a loop.

Let's do an example!

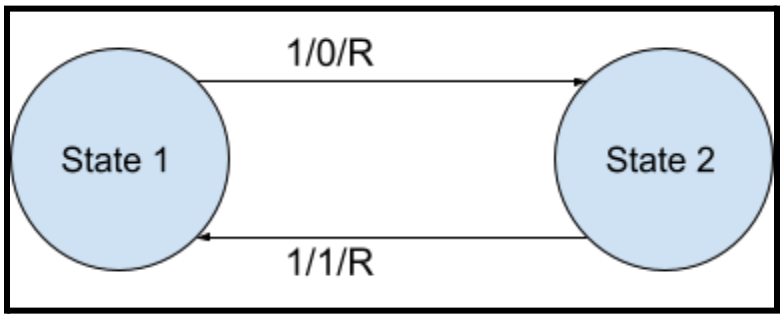
Suppose we have this tape: … b 1 1 1 b … and the following instruction set: (1,1,0,2,R) and (2,1,1,1,R) Determine its output:

Remember we start from the leftmost non blank and start in state 1
First set of instructions to execute is (1,1,0,2,R) so we will write a 0 in its place the state is now a 2 and we move to the right tape is now b 0 1 1 b. Now we execute (2,1,1,1,R) Write a 1 in its place go to state 1 and move right, tape is now b 0 1 1 b, now we execute (1,1,0,2,R) Write a 0 in its place go to state 2 and move right tape is now b 0 1 0 b and we have no more matching instructions! Final tape is: … b 0 1 0 b …

You'll be asked to draw State Diagrams that coordinate with the instruction sets from time to time.

1. Draw a bubble for each state, in the last example there are two state so 2 bubbles
2. Draw arrows to bubbles with labels that contain (current symbol, next symbol, direction)

So for the previous example here is the state diagram for these instructions (1,1,0,2,R) and (2,1,1,1,R):



Anytime the current state and next state match you will have an arrow looping back to the same state!