# The Hawk Collective

## Java Programming Tutorial

Set up your development environment by following the instructions here

# Java Program Execution Flow

Java Programs begin execution at a ‘main’ method.

The main method will always have the following method header

```
public static void main(String[] args)
```

Java programs will run each line of code sequentially starting with the first line from the main method.

# Java Syntax (Grammar)

Java is case-sensitive, ‘test’ and ‘Test’ are processed as two completely distinct words

Each line of code ends in a semicolon ‘;’

By convention, variable and method names are written in what is known as Camel-Case.  This means that the first work is lowercase, with following words being capitalized (with no spaces between them) e.g. variableName

Curly braces ‘{}’ are used to denote sections of related code

Lines that begin with ‘//’ or blocks of text bookended with ‘/*’ and ‘*/’ are comments and will not affect program execution

# Data Types

Data can be stored in a variety of formats within any programming language.

Java’s basic or ‘primitive’ types:

- Int (Integer values)
- Boolean (true/false)
- Double (decimal values)
- Char (character values, e.g. ‘a’)
- Long, short, byte, float (These aren’t really that important)

Data can also be stored in Objects and arrays, which will be covered later

# Variables

There are two steps to making a variable in Java, Declaration and Initialization.

Declaration is simply stating the type of the variable, followed by the name

```
int variableName;
```

This tells the computer to make a variable of type int (integer) called ‘variableName’

Initialization is assigning a starting value to a variable.  This can be (but does not have to be) done on the same line as the declaration.

```
int variableName = 4;
variableName = 4;
```

# Constants

Constants are variables that are declared with the ‘final’ keyword

```
final int CONSTANT_VALUE = 42;
```

By convention, constants are named in all caps, with underscores ‘_’ used to separate words

Once initialized, constants cannot be changed

# Mathematical Operators

Basic math operators (+, -, *, /) will function as expected including order of operations (When dividing two integers, the result is always rounded down)

A single equals sign ‘=’ is used as the assignment operator. It will make the left side of the statement equal to the right side.

Two equals signs ‘==’ are used to check equality, i.e. is the left side equal to the right side.

In the code below, the first line makes the value of x equal to 5 + 5 (AKA 10), whereas the second line checks to see if x is equal to 5 + 5 (and will evaluate to either true or false)

```
x = 5 + 5;
x == 5 + 5;
```

# Mathematical Operators 2

The modulo operator ‘%’ is used to take the remainder from dividing two numbers

```
7 % 3 == 1;
```

The increment operator ‘++’ and decrement operator ‘--’ can be used to increase or decrease the value of a variable by 1

```
int x = 1;
x++;
//x == 2;
x--;
//x == 1;
```

# Assignment Operators

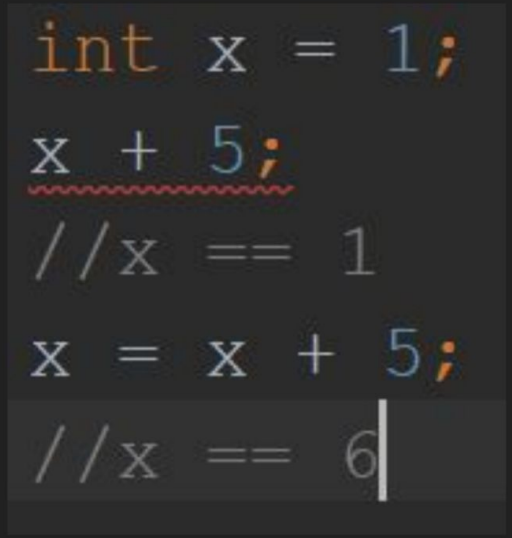
You can combine the basic math operators with the assignment operator ‘=’ to change the value of a variable.  The statement ‘x += 1’ is equivalent to ‘x = x + 1’

```
int x = 1;
x += 1;
//x == 2;
x *= 3;
//x == 6;
```

```
x /= 2;
//x == 3;
x -= 1;
//x == 2;
x %= 2;
//x == 0
```

# Mathematical vs Assignment Operators

Mathematical operators do not change the value of the variables involved, unless an assignment operator is also used



# The Print Method

The below method will print a single line to the standard output, typically the command line.

```
int x = 3;
System.out.println();
System.out.println(3);
System.out.println(x);
```

The above code will have the following output

```
3
3
```

# Example

Sarah and Taylor are both moving today and they have thirty nine identical boxes between them. The moving company charges $15.10 per box.

Write a program that calculates then prints the following:

What is the total price to move all boxes?

A different company charges $50 per five boxes, what would be their cost to move Sarah and Taylor? (Assuming Sarah and Taylor would take the leftover four boxes themselves)

What would the total be if they used the first company for the remaining four boxes?

# Example Solution

```java
public class ExampleOne {

    public static void main(String[] args){

        // Given in prompt
        double numBoxes = 39;
        double boxCost = 15.10;

        // Calculate the total price using the original cost
        double totalPrice = numBoxes * boxCost;

        // Calculate the using the new rate per 5 boxes
        double newPrice = 35 / 5 * 50;

        // Print Info
        System.out.println("Total Cost: $" + totalPrice);
        System.out.println("New Cost: $" + newPrice);

        // Add old moving companies rate for 4 boxes to other companies total
        newPrice += boxCost * 4;

        // Print Info
        System.out.println("Final Cost: $" + newPrice);

    }

}
```

Output should look like this:

```
Total Cost: $588.9
New Cost: $350.0
Final Cost: $410.4
```

# Exercise 1

John, Jack, and Jude are given 65 quarters to share among the three of them.

Write a program that calculates then prints the following:

What is the total value of all the quarters?

If each friend receives an equal amount of quarters, how many are leftover?

In the same case, what is the total value of a single friends quarters? (Excluding any leftover)

If John gives his pile to Jack, what is the value Jack possesses?

If Jack then adds the leftover quarters to his pile, what is its new value?

# Exercise 1 Solution

Total Quarter Value: $16.25
Leftover Quarters: 2
Leftover Quarters Modulo: 2
Single Quarter Pile Value: $5.25
Jack's Value: $10.5
Jack's New Value: $11.0

# Type safety

Variables all have type. Ex: boolean, int, double, char, String

When you try to assign a variable of one type to another, Java will throw an error.

Bad Example:

```
// If you multiply an int by a double the result is a double
int intResult = myIntExample * myDoubleExample;
```

Good Example:

```
// This stores the value of the multiplication into a new variable
double doubleResult = myIntExample * myDoubleExample;
```

# Boolean (Logical) Operators

Boolean operators are those which can affect statements which are evaluated as either true or false

The NOT operator '!' will reverse the true/false state of its operand (what the operator acts on)

```
boolean example = true;
//!example == false
```

```
boolean example = false;
//!example == true
```

The AND operator '&&' will evaluate to true if and ONLY if both operands are true

```
boolean example = true;
boolean trueBool = true;
//example && trueBool == true
```

```
boolean example = true;
boolean falseBool = false;
//example && falseBool == false
```

# Boolean Operators 2

The OR operator '||' will evaluate to true if either of its operands is true

```
boolean example = false;
boolean trueBool = true;
//example || trueBool == true
```

```
boolean example = false;
boolean falseBool = false;
//example || falseBool == false
```

The not equals operator '!=' will evaluate to true if the operands are not equal to one another

```
boolean example = true;
boolean falseBool = false;
//example != falseBool == true
```

```
int x = 5;
//x != 5 == false
//x != 7 == true
```

# Relational Operators

Relational operators (<, >, <=, >=) will work as you would expect when used with numerical values

```
int x = 5;
//x < 5 == false
//x <= 5 == true
//x > 2 == true
//2 > x == false
```

```
int x = 5;
int y = 2;
//x + y < 4 == false
//x + y >= 7 == true
```

# Boolean Expressions

A Boolean Expression is any statement which can be evaluated into true or false. These expressions often chain together multiple boolean operators

```
int x = 5;
int y = 2;
int z = 4;
((x + y > 4) && (z - x != 1))
// true      AND   true  == true
((x / y > 4) || !(z - x == -1))
// false     OR  NOT(true)
// false     OR    false  == false
```

# If Statements

If statements are used to make decisions while the program is running.  If there is not an open brace ‘{‘ after the if statement, the code block is assumed to only include the line immediately following the condition.

```
boolean condition = true;
if(condition)
{
    //if condition is true go here
}
```

A boolean expression, or condition, is placed within the parenthesis ‘()’ and if the expression is true, the block of code is executed.  If the condition is false, then the if statement is skipped and the program will resume execution on the line after the closing brace.

# Else Statements

Else statements are blocks of code which are executed only if the condition of the if statement they are preceded by is false.

```
boolean condition = false;
if(condition)
{
    //if condition is true go here
}
else
{
    //if condition is false go here
}
```

# Nested If and Else Statements

If and Else statements can be placed within other if and else statements. These are referred to as ‘nested’ statements.

It is important to keep paired braces at the same indentation level so you can easily see where blocks of code begin and end.

```
boolean condition = false;
boolean secondCondition = true;
if(condition)
{
    if(secondCondition)
    {
        //both conditions are true
    }
    else
    {
        //condition is true
        //secondCondition is false
    }
}
else
{
    //condition is false
}
```

# Example 2

- Create a new boolean
- Create an if statement that checks whether or not the boolean is true or false
- If true, print out “The boolean is true.”
- If false, print out “The boolean is false.”
- Then create another boolean variable and have another if statement compare the two
- Make a nested if statement that prints something different for each case of the two variables
    - Var1 = true, var2 = true
    - Var1 = true, var2 = false
    - Var1 = false, var2 = true
    - Var1 = false, var2 = false

# Example 2 Solution

```java
public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        boolean booleanExample;
        booleanExample = true;

        if (booleanExample == true) {
            System.out.println("The boolean is true");
        } else {
            System.out.println("The boolean is false");
        }

        boolean booleanExample2 = false;

        if (booleanExample == booleanExample2) {
            System.out.println("The two booleans are the same");
        } else {
            System.out.println("The two booleans are different!");
        }

        if (booleanExample) {
            if (booleanExample2) {
                System.out.println("Both variables are true");
            } else {
                System.out.println("booleanExample is true, but booleanExample2 is false");
            }
        } else {
            if (booleanExample2) {
                System.out.println("booleanExample is false, but booleanExample2 is true");
            } else {
                System.out.println("Both variables are false");
            }
        }
    }
```

Output should look something like this:

```
The boolean is true
The two booleans are different!
booleanExample is true, but booleanExample2 is false
```

# Exercise 2

The normal year contains 365 days, but a leap year contains 366 days. This extra day added to the February month, that is why we get February 29.

As per Mathematics, **except century years**, years perfectly divisible by four are called Leap years.

Century year's means they end with 00 such as 1200, 1300, 2400, 2500 etc (Obviously they are divisible by 100). For these century years, we have to calculate further to check the Leap year.

If the century year is divisible by 400, then that year is a Leap year
If the century year is not divisible by 400, then that year is not a Leap year.

Write a program that determines if a given year is a leap year.

# Exercise 2 Solution

```java
// Code Source: https://www.programiz.com/java-programming/examples/leap-year
public static void main(String[] args) {

    int year = 1900;
    boolean leap = false;

    if(year % 4 == 0)
    {
        if( year % 100 == 0)
        {
            // year is divisible by 400, hence the year is a leap year
            if ( year % 400 == 0)
                leap = true;
            else
                leap = false;
        }
        else
            leap = true;
    }
    else
        leap = false;

    if(leap)
        System.out.println(year + " is a leap year.");
    else
        System.out.println(year + " is not a leap year.");
}
```

# Robot Example

```java
package frc.robot;

import edu.wpi.first.wpilibj.GenericHID.Hand;
import edu.wpi.first.wpilibj.PWMVictorSPX;
import edu.wpi.first.wpilibj.TimedRobot;
import edu.wpi.first.wpilibj.XboxController;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;

/**
 * This is a demo program showing the use of the DifferentialDrive class.
 * Runs the motors with tank steering and an Xbox controller.
 */
public class Robot extends TimedRobot {
  private final PWMVictorSPX leftMotor = new PWMVictorSPX(0);
  private final PWMVictorSPX rightMotor = new PWMVictorSPX(1);
  private final DifferentialDrive robotDrive = new DifferentialDrive(leftMotor, rightMotor);
  private final XboxController driverController = new XboxController(0);

  boolean tankDrive = true;

  @Override
  public void teleopPeriodic() {
    // Drive with tank drive.
    // That means that the Y axis of the left stick moves the left side
    // of the robot forward and backward, and the Y axis of the right stick
    // moves the right side of the robot forward and backward.
    if (tankDrive == true) {
      robotDrive.tankDrive(driverController.getY(Hand.kLeft), driverController.getY(Hand.kRight));
    }
    else {
      robotDrive.arcadeDrive(driverController.getY(Hand.kRight), driverController.getX(Hand.kRight));
    }

    if(driverController.getAButton()) {
      tankDrive = true;
    }
    if(driverController.getBButton()) {
      tankDrive = false;
    }

  }
}
```

# Switch and Break Statements

An alternative to using multiple if statements is to use a switch statement.

A switch statement evaluates the expression then runs all statements following the matching case label.

A break statement can be used to exit the switch statement.  After hitting a break statement, program execution will resume with the first statement following the switch statement.

```
String expression = "dog";
/**The expression can be a char, byte, int,
 *  short, String, or an enumerated type*/
switch(expression){
    case "cat":
        System.out.println("Meow");
        break;
    case "dog":
    case "wolf":
        System.out.println("Woof");
        break;
    default:
        System.out.println("???");
        break;
}//Execution resume here after break
```

# Switch and Break Statements 2

If no case matches the value of the given expression, then execution will move to the default case if present, otherwise it will move to the first statement following the switch statement.

# Exercise 3

Open ExerciseThree.java located at the following link.

Write a program that performs the same functionality as ExerciseThree.java but utilizes switch statements instead of if-else blocks.

# Exercise 3

Exercise 3 Code Solution

Expected output:

```
4 days until the weekend...
The current month is:
June
```

# A Variable’s ‘Scope’

‘Scope’ is the part of a program from which something is accessible.  For example, if you are within the scope a variable was defined, you can see and use that variable, but if you are outside of the scope you cannot interact with the variable in any way.

In general scope is defined by curly braces ‘{}’, so anything that was defined within a set of braces is visible anywhere else within those braces, including within other nested braces.

This means that variables which are declared within an if statement cannot be used outside of that if statement.

# Scope Example

The variable y is defined in scope A, and is therefore usable anywhere within scope A

Variable x is defined in scope B, so is only usable within scope B (The red text color means that there is an error as that line is not within scope B)

```
public static void main(String[] args)
{ //Scope A
    int y = 4;
    if(true)
    { //Scope B
        int x = 7;
        y = x + y;
    }//End Scope B
    y = x + 2;
}//End Scope A
```

Scope B itself is within scope A, therefore anything defined in scope A is usable in scope B (This is why there is no error when using variable y within scope B)

# Garbage Collection

When a program reaches the end of a certain scope (end of a loop, if statement, method, etc.) any references to variables which were created in that scope are deleted.

Any variables which do not have any references pointing to them are automatically deleted, this process is known as garbage collection.

```
if(true)
{
    int i = 4;
}
//i has been garbage collected
```

# Syntax Errors vs Logic Errors

A Syntax Error is something that breaks the rules of the language and cannot be executed.  Referencing a variable that has not been defined, or leaving out a semicolon are examples of syntax errors.

A Logic Error is code that does not break the rules of the language, but which will not produce the desired output or behavior.  A calculator which says ‘2 + 2 = 5’ or a robot that moves backwards when told to move forwards are examples of logic errors.

# Arrays

An array is a data structure that can hold a number of variables/objects of the same data type.  The type of an array is specified by the type of the elements within the array, followed by an empty pair of square brackets ‘[]’ e.g. int[]

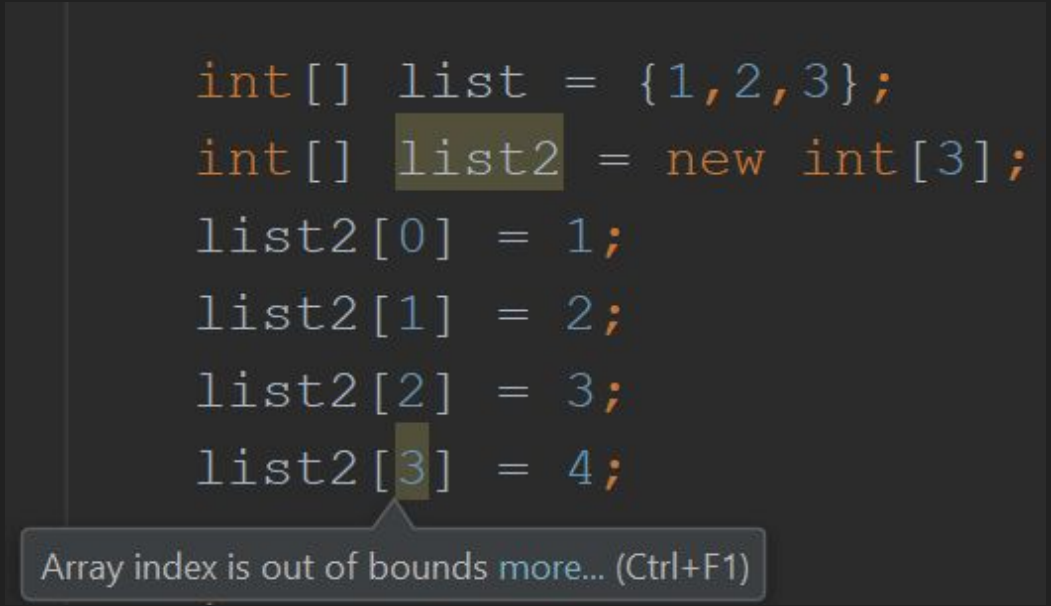
To access an individual ‘element’ of an array, you use square brackets ‘[]’ to specify the ‘index’, or offset from the beginning of the array, starting with 0.

Arrays are of a fixed size, if you try to access an index that is beyond the predefined size of the array (length - 1) an IndexOutOfBounds exception will be thrown.

# Arrays 2

You can initialize an array by using comma separated values (CSV) between a pair of curly braces ‘{}’.

You can also use the ‘new’ keyword to create an empty array, as shown in the code to the right.



# Exercise 4

A small pet store has an inventory of animals. The store has 5 cats, 3 dogs, 10 birds, 2 turtles, and 4 frogs.

The owner wants a program to tell customers how many of each animal they have available. Create a program with 2 arrays, one for the animal names, and one for how many of each corresponding animal is in inventory. Print the name of each animal and how many of that animal the store has.

The owner decides to trade the turtles for ferrets from another pet store. After printing the initial inventory, replace the turtles in the first array with ferrets and re-print the inventory.

# Exercise 4 Solution

```java
public class ExcerciseFour {
    Run | Debug
    public static void main(String[] args) throws Exception {
        String[] animals = {"Cats", "Dogs", "Birds", "Turtles", "Frogs"};
        int[] inventory = {5, 3, 10, 2, 4};

        System.out.println("We have " + inventory[0] + " " + animals[0]);
        System.out.println("We have " + inventory[1] + " " + animals[1]);
        System.out.println("We have " + inventory[2] + " " + animals[2]);
        System.out.println("We have " + inventory[3] + " " + animals[3]);
        System.out.println("We have " + inventory[4] + " " + animals[4]);

        animals[3] = "Ferrets";

        System.out.println("We have " + inventory[0] + " " + animals[0]);
        System.out.println("We have " + inventory[1] + " " + animals[1]);
        System.out.println("We have " + inventory[2] + " " + animals[2]);
        System.out.println("We have " + inventory[3] + " " + animals[3]);
        System.out.println("We have " + inventory[4] + " " + animals[4]);
    }
}
```

Expected Output:

```
We have 5 Cats
We have 3 Dogs
We have 10 Birds
We have 2 Turtles
We have 4 Frogs
We have 5 Cats
We have 3 Dogs
We have 10 Birds
We have 2 Ferrets
We have 4 Frogs
```

# While Loops

```
int x = 1;
while(x < 3) //First condition is checked
{            //is x less than 3
    x++; // If so Add 1 to x
    //Then go back to the beginning of the loop
    //and check the condition again
} //This loop will run twice
```

A while loop is a block of code that will continue to repeat itself ‘while’ a condition (boolean expression) is true.

The condition is checked for ‘truthiness’ prior to each iteration of the loop.

Similarly to an if statement, if the condition is false the block of code within the while loop will never be run.

# Infinite Loops

It is important to make sure that the condition of a while loop can become false while iterating through the loop.

If the condition cannot become false then the loop will run forever.

```
int x = 1;
while(x < 3)
{
    x--; //x is decreasing
    //therefore it will always
    //be less than 3
}
```

# For Loops

```
int sum = 0;
for(int i = 0; i <= 10; i++)
{
    sum += i;
    //This sums all integers 0-10
}
```

The for loop is essentially just a while loop with a built in counter.  You use a for loop when you know ahead of time exactly how many times you want to go through the loop.

The condition of the for loop consists of three statements

- Initialization - This is where you create the variable that will be used to keep track of which iteration of the loop you are on (i.e. first time through the loop i == 1, second time i == 2, etc.)
- Termination - Boolean expression which terminates the loop when false
- Incrementation - increments the counter variable

# Exercise 5

Recreate the solution to Exercise 4 so the solution uses a **while loop** to print the inventory of the petstore the first time (while turtles are present in the array).

Additionally, change the solution so the solution uses a **for loop** to print the inventory of the petstore after the turtles are traded for ferrets.

Each loop should contain no more than 1 print statement.

# Exercise 5 Solution

```java
public class ExcerciseFive {
    Run | Debug
    public static void main(String[] args) throws Exception {
        String[] animals = {"Cats", "Dogs", "Birds", "Turtles", "Frogs"};
        int[] inventory = {5, 3, 10, 2, 4};

        int x = 0;
        while(x < 5) {
            System.out.println("We have " + inventory[x] + " " + animals[x]);
            x++;
        }

        animals[3] = "Ferrets";

        for(int i = 0; i < 5; i++) {
            System.out.println("We have " + inventory[i] + " " + animals[i]);
        }
    }
}
```

Expected Output:

```
We have 5 Cats
We have 3 Dogs
We have 10 Birds
We have 2 Turtles
We have 4 Frogs
We have 5 Cats
We have 3 Dogs
We have 10 Birds
We have 2 Ferrets
We have 4 Frogs
```

# For Each Loop

A for each loop is a shorthand way of defining a for loop which iterates through each element of an array (or other iterable object, such as ArrayList)

The two loops shown to the right both iterate through each int in the list ‘sampleList’ and add it to the sum variable

```
int[] list = {1,2,5,9,1,4,11,9};
int sum = 0;
int sum2 = 0;
for(int i = 0; i < list.length; i++)
{
    sum += list[i];
}
for(int element: list)
{
    sum2 += element;
}
```

# For/While Loop Example

Both of the below programs do the same thing, one is implemented with a for loop, the other with a while loop

```
int sum = 0;
for(int i = 0; i <= 10; i++)
{
    sum += i;
    //This sums all integers 0-10
}
```

```
int sum = 0;
int i = 0;
while(i <= 10)
{
    sum += i;
    i++;
}
```

# Generics

Instead of declaring a specific type, classes can instead declare a ‘generic type’ which can be replaced with a specific type when an object is declared.

A generic type is declared after the class name using diamond braces ‘<>’.  To declare multiple generic types separate them with commas (<E, T>)

In the example to the right, the GenericTest class is declared with ‘E’ representing the generic type.

```
public class GenericTest<E>
{
    E instanceobj;
    public GenericTest(E obj)
    {
        instanceobj = obj;
    }
    public E getInstanceObj()
    {
        return instanceobj;
    }
}
```

# Generics 2

Generic types are specified when the object is declared

Once the object has been initialized, the generic type cannot be changed, and passing in a different type will result in an error (shown below)

```
GenericTest<String> stringTest;
GenericTest<Scanner> scanTest;
stringTest = new GenericTest<String>("Test");
scanTest = new GenericTest<Scanner>(new Scanner("Test"));
scanTest.setInstanceObj("obj");
```

# ArrayList

An ArrayList is a data structure that behaves similarly to an ordinary array, but is resizeable and has a number of additional methods which make it much easier to use. ArrayLists must first be imported above the class declaration by using “import java.util.ArrayList;”

ArrayLists use generic types, and can be used to store any object (only 1 type per list)

```java
import java.util.ArrayList;

public class ArrayListExample {
    Run | Debug
    public static void main(String[] args) throws Exception {
        ArrayList<String> list = new ArrayList<String>();
        list.add("SampleString");
        list.get(0);//"SampleString"
        list.isEmpty();//false
        list.size();//1
        list.contains("SampleString");//true
        list.contains("samplestring");//false
        list.add("SampleString");
        list.remove("SampleString");
        list.contains("SampleString");//true
```

# ArrayList Methods

size() returns the number of elements store in the ArrayList

remove(int index) removes the element at the specified index, moving all following elements left 1 to compensate

remove(Object o) removes the first instance of the given object

add(E e) appends the object to the end of the ArrayList

add(int index, E e) adds the given object at the specified index, shifting any displaced elements to the right

# ArrayList Methods 2

indexOf(Object o) returns the index of the first instance of the given object that is found in the ArrayList, or -1 if the given object is not in the ArrayList

contains(Object o) returns true if there is an object in the ArrayList that is equivalent (using o.equals) to the given object.

get(int index) returns the object in the specified index

set(int index, E element) replaces the element at the given index with the given object.  This does not affect the size of the ArrayList, and index must be >0 && <ArrayList.size()

# Exercise 6

Create a new ArrayList that will hold double objects.

Add these numbers to your arraylist: 4.3, 27.8, 1.0, 3.45, 99.99

Print the length of your ArrayList

Use a method to add 2.5 in between 1.0 and 3.45

Use a method to remove 27.8 from the ArrayList

Use a method to replace 4.3 with 5.55

Print out each double in the arraylist using a for each loop

# Exercise 6 Solution

```java
import java.util.ArrayList;

public class ExcerciseSix {
    Run | Debug
    public static void main(String[] args) throws Exception {
        ArrayList<Double> list = new ArrayList<Double>();

        list.add(4.3);
        list.add(27.8);
        list.add(1.0);
        list.add(3.45);
        list.add(99.99);

        System.out.println(list.size());

        list.add(3, 2.5);

        list.remove(27.8);

        list.set(0, 5.55);

        for(double num: list) {
            System.out.println(num);
        }
    }
}
```

Expected Output:

```
5
5.55
1.0
2.5
3.45
99.99
```

# Practice Problems 3

1. Define an array of integers with 5 entries in it
2. Put a value at each index of the array
3. Write a for loop that prints out all the entries in the array on their own line
4. Write a while loop that does the same thing
5. Write a for each loop that does the same thing
6. Define an ArrayList object that stores integers
7. Add all the integers from your original array to the ArrayList
8. Add one more
9. Print them out using the loop of your choice

# Methods

Methods are sequences of code statements which perform a single operation and can be called or invoked by other sections of code.

Each method has what is known as a method signature which defines visibility, return type (output), name, and parameters (input).

There are three visibility modifiers in Java

- Public - This is visible to anyone
- Private - This is only visible within the same class
- Protected - This is only visible within this class or subclasses of this class*

Classes will be covered later

# Methods 2

The return type for a method can be any primitive data type, array, or class.

Methods can also not return a value, in this case the method signature will contain the keyword ‘void’ in place of a return type.

By convention, method names are written in camel case, and should describe what the method does.

Similarly to mathematical functions, the parameters (input) for Java methods are specified as comma separated values which are declared within parentheses following the method name. (e.g. f(x,y) )

# Methods 3

The below code defines a publically visible method ‘average’ which takes doubles ‘a’ and ‘b’ as parameters, and returns a double which is the average of ‘a’ and ‘b’

```
public double average(double a, double b)
{
    return (a + b) / 2;
}
```

# Exercise 7

Write a method that takes 3 integers as parameters and returns the smallest number

Write a method that takes 3 integers as parameters and returns the largest number

Write a method that takes 3 integers as parameters and returns the average number as an integer

Write a method that takes 3 integers as parameters and prints each integer and returns nothing

In the main method, create 3 integer variables called smallNum, largeNum, and avgNum and call the three methods created using 21, 43, and 7 as parameters, and store the results in the corresponding variables. Then call the last method created and use the variables created as parameters.

NOTE: When creating each method, add static after public since it is in the main class

# Exercise 7 Solution

Expected Output:

```
7
43
23
```

```java
public class ExcerciseSeven {

    public static int smallest(int a, int b, int c)
    {
        if(a < b && a < c)
            return a;
        else if(b < a && b < c)
            return b;
        else
            return c;
    }

    public static int largest(int a, int b, int c)
    {
        if(a > b && a > c)
            return a;
        else if(b > a && b > c)
            return b;
        else
            return c;
    }

    public static int average(int a, int b, int c)
    {
        return (a + b + c) / 3;
    }

    public static void printVars(int a, int b, int c)
    {
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
    Run | Debug
    public static void main(String[] args) throws Exception {
        int smallNum;
        int largeNum;
        int avgNum;

        smallNum = smallest(21, 43, 7);
        largeNum = largest(21, 43, 7);
        avgNum = average(21, 43, 7);

        printVars(smallNum, largeNum, avgNum);
    }
}
```

# Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Most recursive methods will follow the template of **if(condition){ return data }, else { return callFuncAgain() }**

# Recursion

The example function will keep dividing the parameter by 2 until the result is less than 10, at which point it will return the result.

Just like with loops, there is a chance an infinite recursion may happen, where the function will call itself for infinity and never return anything. Make sure there is a case where your recursive function will stop calling itself and return data.

```
public int RecurseExample(int a)
{
    if(a < 10)
    {
        return a;
    }
    else
    {
        return RecurseExample((a / 2));
    }
}
```

# Exercise 8

A factorial is a function in math where a number is multiplied by all the numbers lower than it. It is represented using an exclamation point. So the factorial of 5 (5!) is 5! = 5 x 4 x 3 x 2 x 1. Another way to look at 5! is that 5! = 5 x 4!. Which means that 4! = 4 x 2! and so on. 1! will always equal 1.

Write a method that will find the factorial of an integer given as a parameter using recursion.

Use your function to print the factorial of 12.

# Exercise 8 Solution

```java
public class ExcerciseEight {

    public static int factorial(int a)
    {
        if(a == 1)
        {
            return 1;
        }
        else
        {
            return (a * factorial(a-1));
        }
    }

    Run | Debug
    public static void main(String[] args) throws Exception {
        System.out.println(factorial(12));
    }
}
```

Expected Output:

```
479001600
```

# Exceptions

An exception is an event that disrupts or prevents program execution at runtime.

When an exception is created, or thrown, by a program certain information such as the type of exception and where it occurred are passed up to the next level of execution (This will be explained later, see Exceptions 2).

Exceptions are typically used to prevent undefined or unpredictable behavior from occurring, such as using null values or dividing by 0.

The next slide details some common exceptions you are likely to encounter.

# Common Exceptions

IllegalStateException - Usually used when a method is invoked without fulfilling the prerequisite conditions

NullPointerException - Thrown when an application attempts to use an object that is null

ArithmeticException - Thrown when illegal math operations are performed (e.g. divide by zero)

IOException - Thrown when an IO operation fails or is interrupted

IllegalArgumentException - Thrown when a method is called with invalid input parameters

# Using Objects and Classes

A ‘class’ is an entity in programming which represents and describes a type of thing.

An ‘object’ is a specific instance of a class

The concept of a book is an example of a class, whereas a specific book would be an object.

Classes can have numerous ‘fields’ which can store variables or objects, as well as methods which perform some operation.

By convention, class names are capitalized and object names use camel case

# Using Objects and Classes 2

Objects must be declared and initialized just like variables.

Every class has what is known as a ‘constructor’ method which must be called with the ‘new’ operator to create an object of that class.

Constructor methods share the name of the class and do not specify a return type. (e.g. The constructor method for the String class has the following method signature)

```
public String()
```

The below code demonstrates how to create an Object of the String class

```
String firstObject = new String();
```

# Testing Object Equality

When used with objects, the '==' operator compares the memory addresses associated with the objects.  This will only return true if both operands are the exact same object.

To test whether two objects have equivalent values, you should use the equals() method.  This will return true if the two objects are deemed equivalent by whatever criteria is defined for that class.

```
String testObj1 = new String("Hello World");
String testObj2 = new String("Hello World");
if(testObj1 == testObj2)//This is false
if(testObj1.equals(testObj2))//This is true
```

# Strings

A String is a sequence of characters and can be used to represent text or store data such as a name.

A String object can be created like any other object, using ‘new’ and calling the constructor, or by assigning it to text within quotes (“).

```
//Creates a blank String object
String exampleString = new String();
//reassigning exampleString to be equal to "Example String"
exampleString = new String("Example String");
//Creates a string using a string literal
String literalString = "literal string";
```

Text that is within quotes (“”) is known as a ‘string literal’

# String Operators

Strings in Java have some unique behavior when used with common operators.

The ‘+’ operator can be used to concatenate (Append one to the other) two Strings. Similarly the ‘+=’ operator can be used to concatenate and assign strings in one statement.

```
String helloWorld = "Hello";
//helloWorld == "Hello Wo"
helloWorld = helloWorld + " Wo";
//helloWorld == "Hello World"
helloWorld += "rld";
```

# String Methods

The String class contains a number of extremely useful methods, only some of which will be covered here. Here is the complete API for Strings in Java 10.

toUpperCase() and toLowerCase() return a string with all upper/lowercase characters, “Hello World” becomes “HELLO WORLD” or “hello world”

```
String sampleString = "Hello World";
sampleString.toUpperCase();//"HELLO WORLD"
sampleString.toLowerCase();// "hello world"
```

# String Methods 2

length() returns the number of characters in a string as an int

charAt(int index) returns the char at the specified index of the string (starting at 0)

compareToIgnoreCase(String str) compares 2 strings ignoring upper/lowercase. Returns 0 if the Strings are equal

```
String str = "Hello World!";
str.length();  //returns 12
str.charAt(4); //returns 'o'
str.compareTo("hello world!");//-32
str.compareToIgnoreCase("hello world!");//0
```

# String Methods 3

```
String str = "Test:One,Two,Three";
str.substring(13);//"Three"
str.substring(4,9);//":One,"
str.contains("hey");//false
str.contains("Two");//true
str.split(",");//{"Test:One" , "Two" , "Three"}
```

substring(int beginIndex, int endIndex) returns a new string which is a substring of the original string from beginIndex (inclusive) to endIndex (exclusive).  endIndex can be omitted, in this case the substring continues to the end of the original string

contains(String s) returns true if s is a substring of this string

split(String regex) returns an array of Strings, splitting the original string around matches of the given regular expression

# Exercise 9

Create a new string variable that holds the following text: “The quick brown fox jumps over the lazy dog”

Print the length of the string

Print the substring that contains “the lazy dog”

Compare the string to “The lazy dog jumps over the quick brown fox” and print the difference. Ignore casing for this.

Split the variable using “ ” (a single space) as the separator. Store this new array in a new variable called words. Print how many words are in the array.

# Exercise 9 Solution

```java
public class ExcerciseNine {

    Run | Debug
    public static void main(String[] args) throws Exception {
        String sentence = "The quick brown fox jumps over the lazy dog";

        System.out.println(sentence.length());

        System.out.println(sentence.substring(31));

        System.out.println(sentence.compareToIgnoreCase("The lazy dog jumps over the quick brown fox"));

        String[] words = sentence.split(" ");
        System.out.println(words.length);
    }
}
```

Expected Output:

```
43
the lazy dog
5
9
```

# Importing Classes

Java classes are organized into packages. To use classes that are from a different package* you must first import them.

The code to the right shows how to import the Scanner class, which can read text from files or user input

*The java.lang package is always available. This contains classes like Object, String, Math, Integer, etc.

```
import java.util.Scanner;

public class JavaExamples{
    ▶ Run | Debug
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        String str = scan.nextLine();
        System.out.println(str);
        scan.close();
    }
}
```

# The Scanner Class

The Scanner Class is a powerful utility that can be used to read a source of data as text.  This is useful when reading from a file, or user input from a command line.

The Scanner constructor takes in a ‘source’ parameter of the File, InputStream (e.g. System.in), or String types (among a few others).  This parameter specifies where the Scanner will be reading from.

# The Scanner Class 2

When you are finished using a Scanner you need to invoke the close() method. This is due to IO functions relying on OS level resources which do not have reliable garbage collection.  Failure to close a Scanner or similar IO objects will result in a resource leak that can negatively impact system performance.

Scanners will break their source into ‘tokens’ which are separated by a ‘delimiter’. The default delimiter is a space or line break, however this can be changed if necessary.

# Scanner Methods

hasNext() returns true/false if the scanner has another token in its source

next() returns the next token from the source as a String

nextLine() returns the next line from the source as a String

useDelimiter(String pattern) changes the delimiter to use the given pattern instead of the default whitespace

```
Scanner scan = new Scanner("Testing, one two, three four");
scan.hasNext();//true
scan.next();//"Testing,""
scan.useDelimiter(",");
scan.next();//" one two"
scan.nextLine();//, three four
scan.hasNext();//false
scan.next();//Throws noSuchElementException
scan.close();
```

# Exercise 10

Create a program that accepts positive integers from user input one at a time and stores the integers in an ArrayList.

The program should keep accepting integers from user input until a negative number is inputted. Once a negative number is inputted, the program should calculate the average of all the numbers and output the average of the numbers stored in the ArrayList.

Note: The negative number should NOT be stored.

[Exercise 10 Solution]

Expected Output:

```
Please enter positive integers one at a time.
You may exit by entering any negative integer.
5
11
99
2
23
40
-1
Average is: 30
```

# Writing Classes

- A class is the blueprint from which individual objects are created.
- Classes are instantiated by using the key word ‘new’ and then the name of the class.
- Classes should begin with an uppercase.
- Classes contain a constructor that defines variables.
- Classes implement methods and their own data.

# Class Example

Pull up the following Bicycle.java class from this link.

Over the next few slides we will look at each section of this class implementation. Including the class variables, constructor, and methods defined in the class.

# Class Level Variables

```
// the Bicycle class has
// three fields
public int cadence;
public int gear;
public int speed;
```

As you've previously learned, variables have scope within a program. Variables defined at the class level are in the scope of that class and can be accessed from anywhere within that class.

This can also apply to external classes. If you define public class level variables, you can access those variables from another class as long as you have an instance of your object.

You can avoid this situation by making your variables private. If you want to control access to a classes data members then be sure to write your own get and set methods in your class definition.

# Constructor

A class constructor can be seen as a method that is called when you want to create a new instance of an object. Just like methods they can be overloaded, however there is no return on a constructor. Notice how in the constructor you have the option to initialize class level variables by using parameters.

```
// the Bicycle class has
// one constructor
public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}
```

# Exercise 11

Open the exercise sheet posted at this link and complete the listed tasks for Holiday.java and Movie.java

Exercise 11 Code Solution (Holiday.java)

Exercise 11 Code Solution (Movie.java)

# Enumerated Types

An enum type is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.

Because they are constants, the names of an enum type's fields are in uppercase letters.

In the Java programming language, you define an enum type by using the enum keyword. For example, you would specify a days-of-the-week enum type as:

# Enumerated Types

```
public enum Day {

    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,

    THURSDAY, FRIDAY, SATURDAY

}
```

You should use enum types anytime you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and data sets where you know all possible values at compile time—for example, the choices on a menu, command line flags, and so on.

# Exercise 12

Implement the started code on this slide using enums instead of integers.

Starter Code

Solution

# Exceptions 2

There are situations in which you will want to throw an exception on your code, the most common of which are:

- A precondition is not met
- A postcondition cannot be met

Exceptions are thrown using the ‘throw’ keyword

The example code on the next slide shows an example of throwing an exception for both a pre- and postcondition

# Exceptions 2-2

```
boolean noAvailableInternetConnection;

/**
 * Loads a given Https url in a WebView
 * @param url - Https URL to be loaded
 * @return WebView object containing the page at url
 */
public WebView loadSecureURL(String url)
{
    if(!url.toLowerCase().contains("https"))
    {//Precondition of url being a secure URL was violated
        throw new IllegalArgumentException("Must be an https url");
    }
    if(!internetConnectionFound())
    {//Postcondition of loading the url into a WebView
    //cannot be fulfilled without an internet connection
        throw new NoInternetException();
    }
    return loadURL(url);
}
```

# Try/Catch

A try catch block is a logical structure that allows your program to continue to run even if an error is thrown. Consider the example code below:

```
public class MyClass {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

Because there is no index 10 in the myNumbers array, an IndexOutOfBounds exception is thrown. But instead of halting the program, the “Something went wrong message is printed to standard out.

# Exercise 13

Open the code file TryCatchStarter.java using this link. Then, without removing anything that is already there, alter the program so that all of the print statements are displayed in the console when run.

Try Catch Exercise Code Solution

Expected output:

```
Starting Point
Checkpoint 1
Checkpoint 2
Finished
```

# The ‘static’ Modifier

By default, methods, fields, and classes need to be instantiated into an object to be used.  This is because the values stored in fields and output from methods are often determined by data associated with the individual object.

The ‘static’ modifier tells us that the method/field/class belong to the class rather than a specific object, and that they can be used without instantiating an object. No matter how many instances of a class are created, there will only be one instance of any static fields.

# Using Static Fields and Methods

There are a lot of static methods which are used to provide general utility when programming.  The below code demonstrates calling a static method from the Math class to generate a random double between 0 and 1.  Note that Math is referring to a class, not an object.

```
public static void main(String[] args) {

    Math.random();
```

# Using Static Fields and Methods 2

Static fields are shared between all instances of a class.  This can be useful for constants that should have the same value, but can also be used in other ways.

The code to the right uses a static field to give each instance of the TestJava class a unique ID number.  Since the uniqueID field is static, it will be changed across all instances of TestJava whenever the constructor is called.

```
public class TestJava {

    private static int uniqueID = 0;
    public int ID;
    public TestJava()
    {
        this.ID = uniqueID++;
    }
```

# Exercise 14

Create a public class called President. The President class should have a constructor that takes a firstName (String), a lastName (String), and an ID (int). Make variables for each of the constructor parameters so that they can be accessed later.

Make a private variable that is static called inOffice. It should be a boolean and it should be used to make sure only ONE object of the President type is created (any subsequent constructor calls should set firstName and lastName to null, and ID to -1)

Create a static method called print that outputs the phrase “I am the President of the United States”. Call this method in the main method.

# Exercise 14 Solution

Excercise 14 Solution

# Static Classes

In Java, you can create custom classes within other classes. These are called nested classes.

These nested classes can be static classes or non-static classes.

A static class is a nested class that can have objects of it created without creating an instance of the outer class. In a non-static nested class, you must first create an instance of the outer class before you can create an instance of the inner class.

While nested classes can access the variables and methods of the outer class, a static nested class can only access **static** variables and methods of the outer class.

# Static vs Non-static classes

<- Static

Creating a nested static object ->

<- Non-Static

Creating a nested non-static object ->

```java
// Java program to Demonstrate How to
// Implement Static and Non-static Classes

// Class 1
// Helper class
class OuterClass {

    // Input string
    private static String msg = "GeeksForGeeks";

    // Static nested class
    public static class NestedStaticClass {

        // Only static members of Outer class
        // is directly accessible in nested
        // static class
        public void printMessage()
        {

            // Try making 'message' a non-static
            // variable, there will be compiler error
            System.out.println(
                "Message from nested static class: " + msg);
        }
    }

    // Non-static nested class -
    // also called Inner class
    public class InnerClass {

        // Both static and non-static members
        // of Outer class are accessible in
        // this Inner class
        public void display()
        {

            // Print statement whenever this method is
            // called
            System.out.println(
                "Message from non-static nested class: "
                + msg);
        }
    }
}
```

```java
// Class 2
// Main class
class GFG {

    // Main driver method
    public static void main(String args[])
    {

        // Creating instance of nested Static class
        // inside main() method
        OuterClass.NestedStaticClass printer
            = new OuterClass.NestedStaticClass();

        // Calling non-static method of nested
        // static class
        printer.printMessage();

        // Note: In order to create instance of Inner class
        //  we need an Outer class instance

        // Creating Outer class instance for creating
        // non-static nested class
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner
            = outer.new InnerClass();

        // Calling non-static method of Inner class
        inner.display();

        // We can also combine above steps in one
        // step to create instance of Inner class
        OuterClass.InnerClass innerObject
            = new OuterClass().new InnerClass();

        // Similarly calling inner class defined method
        innerObject.display();
    }
}
```

# Exercise 15

Create a public class called OuterClass. Create 3 private variables called x, y, and z. Then create a static variable called multiplier and set it to 1.

Create a non-static class called shooter. shooter should have a constructor that takes variables as parameters, and sets x, y, and z. Shooter should have a method called shoot that prints x, y, and z after they each have been multiplied by the multiplier variable.

Create a static class called Force. Force should have a static method called changeForce that takes a integer as a parameter, and sets the multiplier variable to it.

Finally, create a Shooter object in the main method and call the shoot method before and after changing the multiplier.

# Exercise 15 Solution

Excercise 15 Solution

# Inheritance (Subclasses)

Inheritance is a way in which you can reuse the properties of a class. With inheritance, you can create another class that extends the first one and inherits its data and methods. The subclass (or child class) can then access the methods and variables of the superclass and add its own methods and variables. Note: the superclass (or parent class) can NOT access the variables and methods of the sub class.

To create a subclass, simply use the extends keyword after the name of the subclass and specify the superclass.

# Inheritance (Subclasses) Example

Note that even though s1 is an object of Students, it can access the teach() method from the Teacher class.

```
class Teacher {
 void teach() {
  System.out.println("Teaching subjects");
 }
}
class Students extends Teacher {
 void listen() {
  System.out.println("Listening to teacher");
 }
}
class CheckForInheritance {
 public static void main(String args[]) {
  Students s1 = new Students();
  s1.teach();
  s1.listen();
 }
}
```

# Exercise 16

Create a class called Vehicle. Vehicle should have 3 variables to store it’s brand, yearCreated, and mode. These variables should be a String, an int, and a String respectively.

Create a class called Car that extends the Vehicle class. Car should have a String variable called model. Create a constructor for Car that takes brand, yearCreated, and model as parameters and sets the variables respectively. Set the mode to “drive”

Create a class called Airplane that extends the Vehicle class. Airplane should have an int variable called id. Create a constructor for Airplane that takes brand, yearCreated, and model as parameters and sets the variables respectively. Set the mode to “fly”

Finally, create a an object of type Car and an object of type Airplane and print every available variable from each.

# Exercise 16 Solution

Excercise 16 Solution

# Documentation

There are three types of documentation commonly used for Java programs

- READMEs - Anyone should be able to read this to know WHAT your program is.
- JavaDocs - Any developer should be able to read these to know WHAT each field/method/class in your code is used for.
- Code comments - Anyone familiar with the language should be read these to know HOW your code works.

Ideally you should have up-to-date versions of all three types of documentation at all times for any code that you are writing.

# READMEs

READMEs are markdown files (.md) which are included in software projects and provide documentation on what the software does and how to use it

Anyone (including non-devs) should be able to read your README and know what your code is doing

Markdown formatting guide

# JavaDocs

JavaDoc is a tool to generate documentation about classes, methods, and variables. It usually contains high-level information about each of these kinds of data and describes what they are used for, what information it has/returns, or what parameters it accepts.

JavaDoc is packaged with any modern JDK and can generate documents by using the javadoc command in the command line.

For example, you can generate documents called myDoc of your project by running the command “javadoc -d myDoc src*” inside your project location. After the command finishes, you should see a directory called myDoc. Inside is a file called index-all that you can open using your browser that will give an interactive way to view the information you created.

For what comments to include in the code that will be converted to the JavaDoc, please see this article.

# Code Comments

Code comments are comments of plaintext within the java code and should explain to developers how specifically your code works.

To add a comment line inside of java code, simply add // to the start of the line to turn the line into a comment. You can also create a multi-line comment by adding /* to the start of what you want to comment and adding */ to the end. Anything between /* and */ will be turned into a comment.

Example ->

```java
public double exampleMethod(int a, int b, int c)
{
    // Adds 5 to each parameter
    a = a + 5;
    b = b + 5;
    c = c + 5;

    /* Returns the average by adding a, b, and c together
    and then dividing by 3 */
    return ((a + b + c) / 3);
}
```

# Exercise 17

Take your solution to Exercise 16 and add comments to the code and create a JavaDoc for it.

The code comments should be above the class variables of each class and inside the constructor of each class (if there is one). The comments should say whether the variable below comes from the current class or which superclass it comes from.

The JavaDoc should tell an explanation of each class and what kind of class it is (superclass, subclass). Additionally, it should explain the parameters of each constructor and explain what mode is set to for each constructor.

# Exercise 17 Solution

Code Comments Solution

JavaDoc Solution

# Useful Resources

FIRST Programming Resources Page - Instructions on how to setup your environment, along with various guides on how to program the robot using Java

Java 8 API - Complete documentation for the standard Java library version 8

Java 10 API - Complete documentation for the standard Java library version 10

StackOverFlow - Q&A website used for crowdsourced troubleshooting, if you have a problem you can’t figure out 80% chance it’s on here somewhere

Team2890 Github Account - Official Team GitHub account

Team Github Page - GitHub team, containing team documentation/training materials