

# THE HAWK COLLECTIVE

JAVA PROGRAMMING TUTORIAL

# Setting Up Your Environment for Java Development

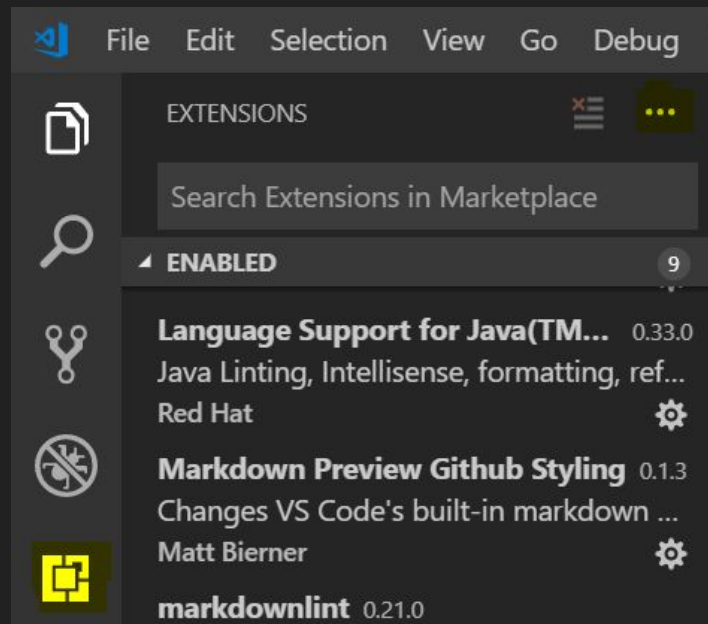
## Download and Install Visual Studio Code

Create a folder to use as a workspace, this is where all of your projects will be stored.

In Visual Studio Code, go to the extensions tab (pictured right) and install the following extensions:

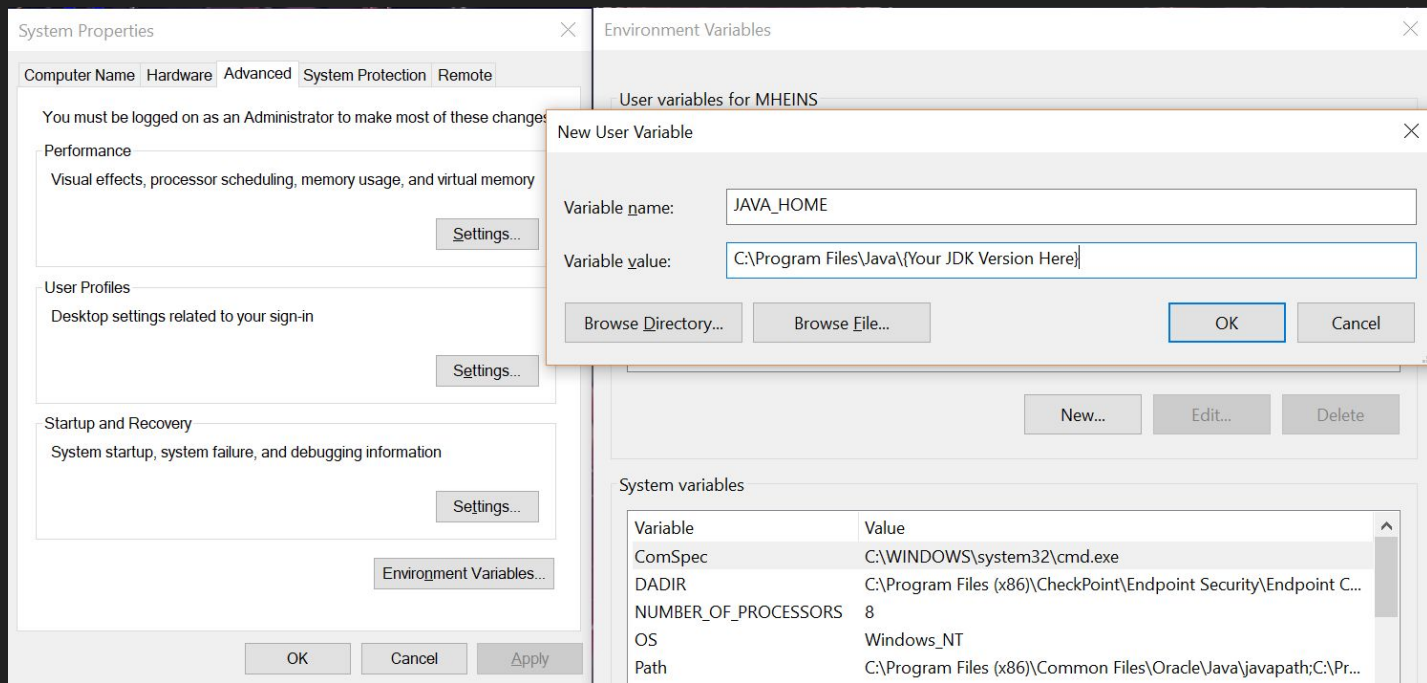
- Java Extension Pack from Microsoft
- Code Runner from Jun Han
- WPI Extension (Select “Install from VSIX from the context menu (...) in the extensions tab

## Install the JDK.



# Setting Environment Variables

You may need to set the `JAVA_HOME` Environment Variable (Search 'Environment Variable' to open the window on the left).



# Creating a New Java Project in VSCode

# 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 word 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

```
int x = 1;  
x + 5;  
// x == 1  
x = x + 5;  
// x == 6
```

# 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
```

# 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
}
```

# 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)

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)

```
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
```

# 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.

# 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.

# 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
```

# 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

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.

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

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

# 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 `IndexOutOfBoundsException` 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.

```
int[] list = {1,2,3};  
int[] list2 = new int[3];  
list2[0] = 1;  
list2[1] = 2;  
list2[2] = 3;  
list2[3] = 4;
```

Array index is out of bounds [more...](#) (Ctrl+F1)



# 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++;
}
```

# 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;
}
```

# 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](#)

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

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.

# 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();
```

# ArrayList

//Arrays vs ArrayList, Mutable, Helpful Methods, Generics

# Writing Classes

# Exceptions 2

//Throwing an exception, Try/Catch blocks

# 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++;  
    }  
}
```



# Static Classes

# Enumerated Types

# Inheritance (Subclasses)

# Interfaces

# Git and Version Control

Git is a Version Control System (VCS) which can be used to backup and manage large projects with numerous contributors.

A repository, or repo, is the collection of folders and files which are tracked as part of a project.

A commit is a change, or collection of changes, that have been made to the repository. By looking at the commit history of a project, developers can learn what changes were made when, by who, and why.

Commits can also be easily reverted if the changes are later found to cause bugs, or have other unintended side effects

# Git 2

A branch is a separate version of the same codebase. For example typically projects have a 'master' branch alongside various branches for different features that are in development.

The 'master' branch should contain only fully functional code and features. This may also be considered a 'production' branch.

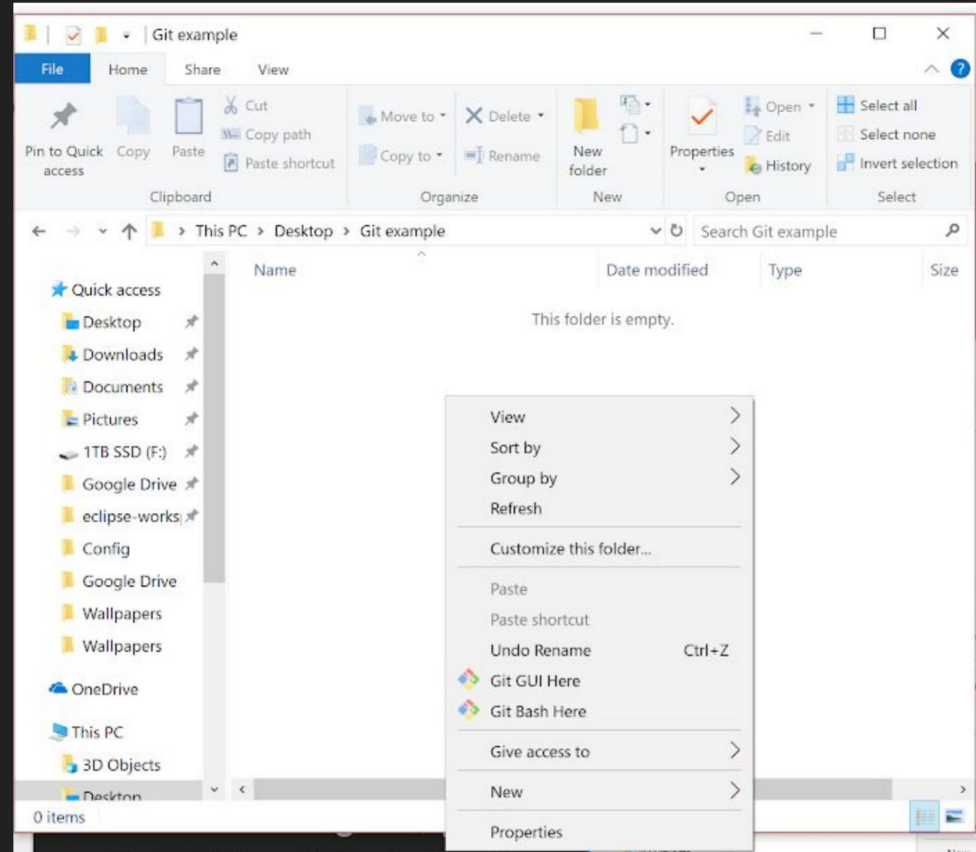
'Development' branches are used to track changes while a specific feature is being worked on. Code in these branches may not always be completely Functional.

A 'merge' is the process by which changes made in one branch are combined with those of another branch

# Setting Up Git

To setup a new Git repo, you will first need to [download and run the installer](#) (default settings are fine).

After installing Git, you should be able to right click and select the option “Git Bash Here” (pictured to the right). This will open a Command Line Interface (CLI) through which you can run git commands

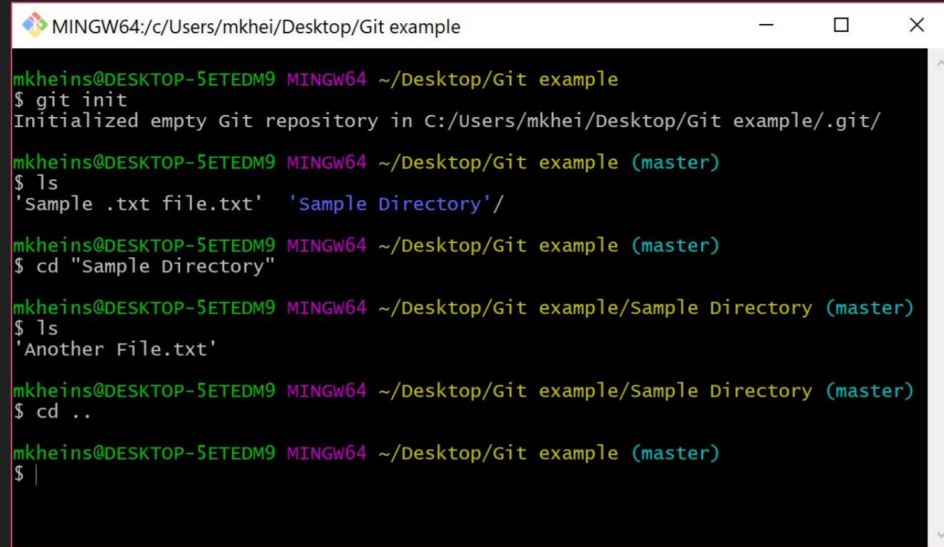


# Useful Git Bash Commands

‘git init’ will initialize a new repository in the current directory (shown in yellow)

Every new project will have a single branch, ‘master’ (shown in blue)

‘Cd’, ‘ls’, and ‘..’ are used to change directory, display what files/folders are in the current directory, and refer to the parent directory of the current directory



```
MINGW64:/c/Users/mkhei/Desktop/Git example
mkheins@DESKTOP-5ETEDM9 MINGW64 ~/Desktop/Git example
$ git init
Initialized empty Git repository in C:/Users/mkhei/Desktop/Git example/.git/

mkheins@DESKTOP-5ETEDM9 MINGW64 ~/Desktop/Git example (master)
$ ls
'Sample .txt file.txt' 'Sample Directory'/

mkheins@DESKTOP-5ETEDM9 MINGW64 ~/Desktop/Git example (master)
$ cd "Sample Directory"

mkheins@DESKTOP-5ETEDM9 MINGW64 ~/Desktop/Git example/Sample Directory (master)
$ ls
'Another File.txt'

mkheins@DESKTOP-5ETEDM9 MINGW64 ~/Desktop/Git example/Sample Directory (master)
$ cd ..

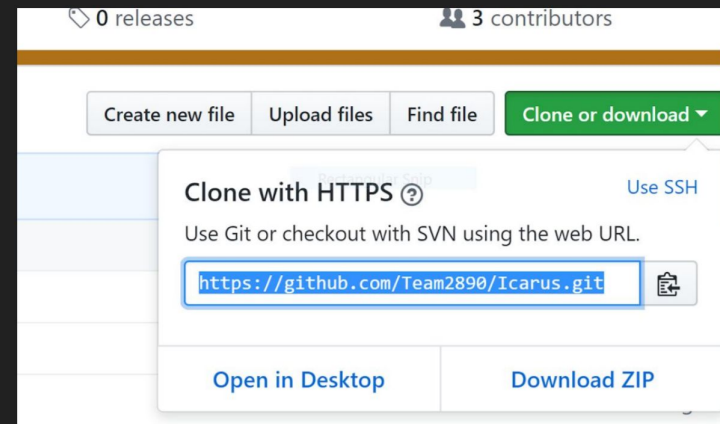
mkheins@DESKTOP-5ETEDM9 MINGW64 ~/Desktop/Git example (master)
$ |
```



# Useful Git Bash Commands 2

'git clone' will copy the repository located at the https or ssh location specified by onto the local machine, creating a subdirectory of the current directory

'git config user.name' will set the name that is recorded for all commits made on the local machine. This command can be rerun at any time to change the username



```
MINGW64/c/Users/mkhei/Desktop/Git example
mkheins@DESKTOP-5ETEDM9 MINGW64 ~/Desktop/Git example (master)
$ git clone https://github.com/Team2890/Icarus.git
Cloning into 'Icarus'...
remote: Counting objects: 4109, done.
remote: Total 4109 (delta 0), reused 0 (delta 0), pack-reused 4109
Receiving objects: 100% (4109/4109), 11.25 MiB | 8.03 MiB/s, done.
Resolving deltas: 100% (2137/2137), done.

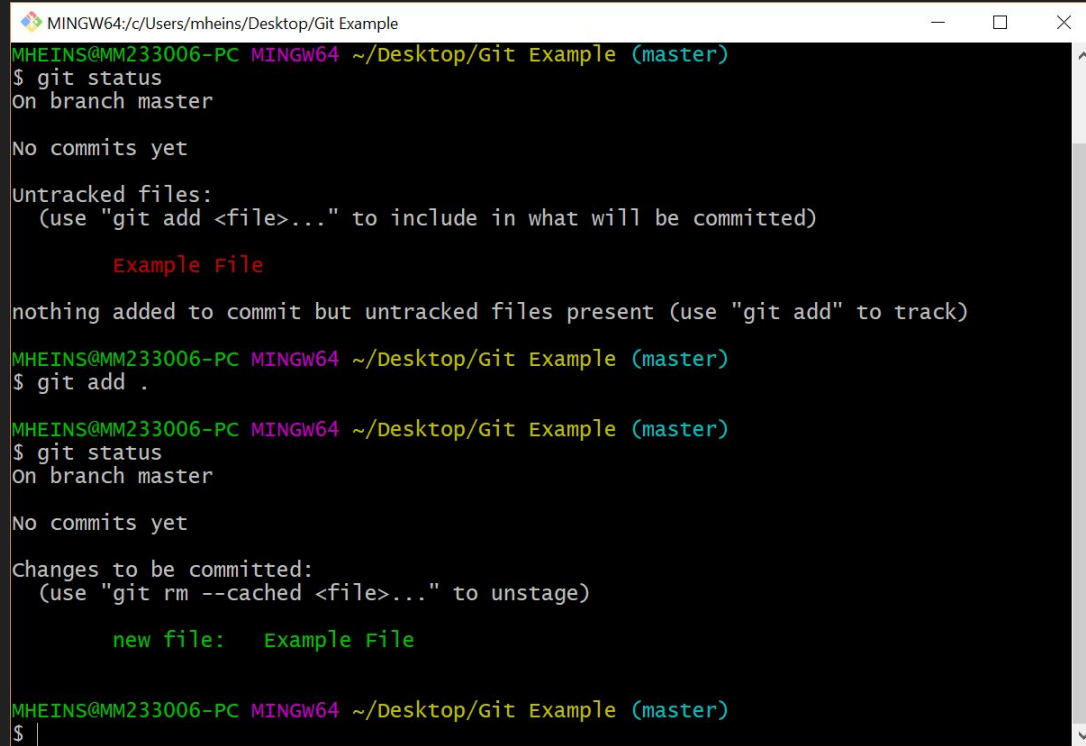
mkheins@DESKTOP-5ETEDM9 MINGW64 ~/Desktop/Git example (master)
$ git config user.name 'Matt Heins'

mkheins@DESKTOP-5ETEDM9 MINGW64 ~/Desktop/Git example (master)
$
```

# Useful Git Bash Commands 3

'git status' will show any staged or unstaged changes that have been made to the current directory. Unstaged changes will not be recorded if a commit is made.

'git add .' will stage all changes in the current directory, as well as all subdirectories. This should be done prior to running



```
MINGW64:/c/Users/mheins/Desktop/Git Example
MHEINS@MM233006-PC MINGW64 ~/Desktop/Git Example (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Example File

nothing added to commit but untracked files present (use "git add" to track)

MHEINS@MM233006-PC MINGW64 ~/Desktop/Git Example (master)
$ git add .

MHEINS@MM233006-PC MINGW64 ~/Desktop/Git Example (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

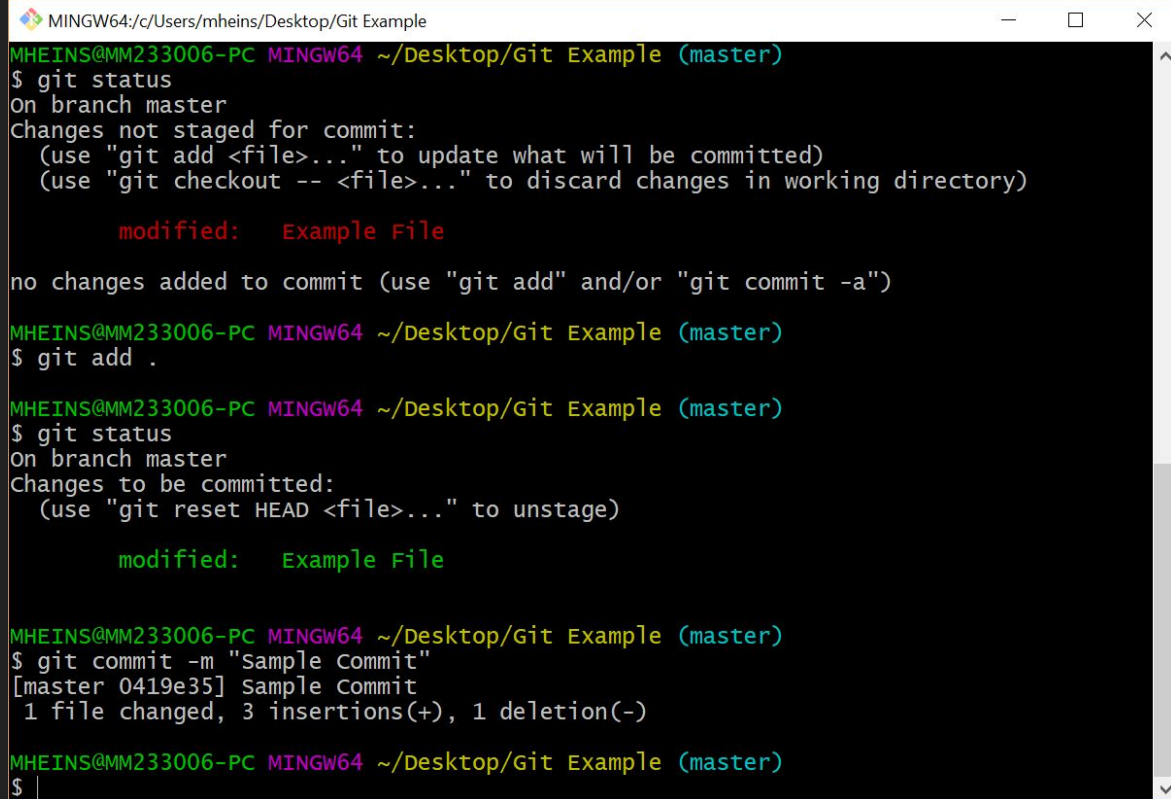
        new file:   Example File

MHEINS@MM233006-PC MINGW64 ~/Desktop/Git Example (master)
$ |
```

# Useful Git Bash Commands 4

'git commit -m "{x}"' will commit the currently staged changes, with the commit message of {x}. The message should explain all changes made by the commit.

'git push' will upload the commits made to your local repo to the remote repo

A screenshot of a Git Bash terminal window. The title bar shows the path 'MINGW64:/c/Users/mheins/Desktop/Git Example'. The terminal content shows a series of Git commands and their outputs. First, 'git status' is run, showing that 'Example File' is modified but not staged. Then, 'git add .' is run to stage the changes. Finally, 'git commit -m "Sample Commit"' is run, creating a new commit with the message 'Sample Commit' and showing that 1 file changed with 3 insertions and 1 deletion. The prompt '\$' is visible at the bottom, indicating the terminal is ready for the next command.

```
MINGW64:/c/Users/mheins/Desktop/Git Example
MHEINS@MM233006-PC MINGW64 ~/Desktop/Git Example (master)
$ git status
On branch master
changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Example File

no changes added to commit (use "git add" and/or "git commit -a")
MHEINS@MM233006-PC MINGW64 ~/Desktop/Git Example (master)
$ git add .
MHEINS@MM233006-PC MINGW64 ~/Desktop/Git Example (master)
$ git status
On branch master
changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Example File

MHEINS@MM233006-PC MINGW64 ~/Desktop/Git Example (master)
$ git commit -m "Sample Commit"
[master 0419e35] Sample Commit
 1 file changed, 3 insertions(+), 1 deletion(-)
MHEINS@MM233006-PC MINGW64 ~/Desktop/Git Example (master)
$
```

# 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

# JavaDocs

# Code Comments

# Robot Code

//TODO add tutorials for robot specific code, frequently used classes, structure diagrams, external utilities such as driver station etc.



# 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

# Useful Resources 2

[Git Handbook](#) - Additional reading material on Git

[Google](#) - Google is your friend