/////////////// Fields ////////////////////////////////////////////////////

[private](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/private)  
The type or member can be accessed only by code in the same class or struct.

[protected](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/protected)  
The type or member can be accessed only by code in the same class, or in a class that is derived from that class.  
[internal](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/internal)  
The type or member can be accessed by any code in the same assembly, but not from another assembly.

[protected internal](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/protected-internal) The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly.1

[private protected](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/private-protected) The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class.

///////////////C# Basics /////////////////////////////////////////////////

mcs -debug Program.cs

mono --debug Program.exe

* **Namespace.Class.Method**
* Keywords <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/index>
* C#
  + Is a compiled language
    - Needs a compiler to take Human Readable Code and transfer it into Computer Readable Code
      * *mono* Compiler is a popular compiler
        + Run the compiler by typing: ***mcs Program.cs*** (mono-c-sharp) in terminal

Compiler creates an executable file out of c# code

* + - * + ***ls*** (list) to see everything in the file directory

i.e. Program.cs (Code file) to **Program.exe** (Executable file)

* + - * + ***Mono Program.exe***

Run the .exe file

* + - ***System.Console.Write(“Hello World!”); //Write script***
    - ***mcs Program.cs //Type this in terminal to start program***
    - ***ls***
    - ***Program.cs Program.exe***
    - ***Mono Program.exe***
    - See program in terminal: Hello World!
* Popular language
  + Web servers
  + 3D games
  + Applications
  + Easy to write
  + Doesn’t break as often
  + Not only on Windows

//////// Namespaces /////////////////////////////////////////////////////

* Namespaces
  + 2 reasons c# uses them
    - .Net Framework uses namespaces to organize its many classes
    - Used to provide the compiler a context for all the named information in your program
      * Variable names
      * Class names
  + Usually companies will use their name as their main Namespace
    - i.e. namespace Treehouse
  + Can use multiple namespaces to create an app or project with your company
    - i.e. namespace Treehouse.FitnessFrog
  + **Using Directive**
    - *using*: at beginning of program, don’t have to write the username before every *class* in a *method*
      * i.e. using System;
        + don’t need to write System.Console.Write (can write Console.Write instead)
    - More readable
    - Write less code

//////// Classes //////////////////////////////////////////////////////////

* Class (Classification)
  + If a file only has one class = name the class the same name as the file
  + Have multiple methods inside them
    - Used to organize the code into smaller chunks
* Base Class
  + The class whose members are inherited is called the base class
    - Parent Class
    - Super Class
* Sub Class = Derived Class (Child Class)
  + Class that inherits those members is called the derived class
    - Derived class can have only one direct bas class
  + Gains all the members of the base class, except for its constructors and finalizes
  + Can reuse the code in the base class without having to re-implement it
  + Can add more members and extends the functionality of the base class

//////// Methods ////////////////////////////////////////////////////////

* **Methods** are the moving parts of an object
  + Its where the code does stuff lives
  + Use the fields in some way
* **Methods** have **Four** Parts
  + Name
    - Give it a name that makes sense
  + Body
    - Everything in between curly braces
    - List of instructions to tell the computer what to do
  + Parameters
    - Inside parentheses of method
    - Runs a particular type of information designated as a parameter
  + Return Type
    - Takes the parameters and returns information
* Computer program consists of
  + Files
    - Can have multiple classes
    - Must have a .cs (C-Sharp) at the end of the file in order to run (program.cs)
  + Classes
    - Can have multiple methods
  + Methods
* **Main()**
  + First method that is run when the program is started
  + Computer looks for this method in C#
  + It’s a rule that must be used when running a C# program
* **Parse();**
  + Parses Strings into Integers
    - i.e. int.Parse(“55”);
      * answer: 55
    - i.e. int x = int.Parse(“55”);
      * x = 55
  + Can also parse “double”
* **ToLower()**
  + Even if the user capitalizes all of the letters in the program this method will make them all lowercase so its easier to run the whole program and not write a bunch of code to accepts Lowercase and Uppercase letters
* **GetType()**
  + Used with object to find the type of variable or class in a program
    - int x = 5;
    - double y = 2.5;
    - (x + y).GetType();
    - System.Double
* **Math.Sqrt()**
  + Returns the square root of a specified number
  + return (int)Math.Sqrt(xDiffSquared + yDiffSquared);
  + <https://msdn.microsoft.com/en-us/library/system.math.sqrt(v=vs.110).aspx>
* **Math.Pow()**
  + Returns a specified number to the specified power
    - Math.Pow(value, power);
  + <https://msdn.microsoft.com/en-us/library/system.math.pow(v=vs.110).aspx>
* **Static Methods**
  + They are called directly on the class
    - Don’t need to construct an object before we can use them
    - i.e. public static double Sqrt( )
    - i.e. Math.Sqrt
      * There is no Math object being used
      * Square root method is being called directly on the class
    - i.e. WriteLine()
    - i.e. Main()
      * First method called in the program
      * Has to be a static method
      * That’s why it has the “static” keyword
        + i.e. public static void Main()
      * Allows the Main method to be called without first creating a game object
  + **Static Methods don’t have access to instance methods**
  + Static Methods only have access to what is inside its curly braces and **other Static Methods**
* **Instance Methods**
  + They are called directly on objects
    - Objects are known as instances of a class
* **Overloading a Method**
  + To overload a method means to have two methods with the same name but take different parameters

//////// Variables ////////////////////////////////////////////////////////

* **Variable**
  + Types = <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/built-in-types-table>
    - **String**
      * string
      * Name but cant start with a number and must be descriptive
    - **Integer**
      * Int
      * Pseudonym for System.Int.32 and can be used interchangeably between int and System.Int.32
      * Number that can be written without a decimal component
      * Can be positive or negative
      * Computer prefers integers because its faster to run
      * If you add/subtract/multiply/divide an integer to a double it will always return as a double
    - **Double**
      * double
      * used with decimals
      * Pseudonym for System.Double but can just use *double* and *System.Double* interchangeably
      * If you add/subtract/multiply/divide an integer to a double it will always return as a double
    - **Boolean**
      * bool
      * true or false
  + C# assigns values to the right of the equal sign to the variable
  + Variable Declaration
    - Declare a variable without putting a value inside of it
      * string firstName;
  + Variable assignment
    - Assign a value to variable
      * firstName = “Brennan”;
  + **Initializing the Variable**
    - Declare a variable then assign a value to the variable
      * string firstName = “Brennan”;
  + **Compile Time Inference**
    - When you initialize a variable at the same time you declare it, you don’t need to specify the type of the variable and can use var as a placeholder
      * The compiler can determine by looking at the right side of the equals sign and infer what the type is when it compiles the code
    - **Var**
      * Acts as a placeholder for the actual type
      * If you want to use a double and it is the integer 0 you can use 0.0 to infer that you want to use it as a double and not an integer
        + i.e. var runningTotal = 0.0;
  + Syntactic Sugar
    - i.e. runningTotal = runningTotal + minutes;
    - refractor it to look like:
      * runningTotal += minutes; //use += operator
      * “+=, -=, \*=, /=, %=”
  + **Instant Level Variable**
    - Exist as long as the object exists
    - Can be used by other methods and other classes
    - Standard convention to name **Methods and Public Instance Variables** start with a **UPPERCASE** letter
  + **Method Level Variables**
    - Only exist in a method
    - Can only be used within a method
    - Standard convention to name **Method Parameters and Method Variables** starting with a **lowercase** letter

/////////Strings /////////////////////////////////////////////////////////

* String
  + String Literal
    - string entry = “22”;
    - String that literally contains “22”

///////// Operators //////////////////////////////////////////////////////

* Addition
  + “+”
* Subtraction
  + “-“
* Multiplication
  + “\*”
* Division
  + “/”
* Modulus
  + “%”
* Comparison
  + “<, <=, >, >=, ==, !=”
  + Can only compare integers and not strings
  + Used a lot in Conditional if/else statements

/////////// Conditional Statements if/else ///////////////////////////////////

* **Conditional if / else**
  + Used to perform different actions based on different conditions
    - Use **if**
      * Specify a block of code to be executed if condition is true
    - Use **else**
      * Specify a block of code to be executed if the same condition is false
    - Use **else if**
      * Specify a new condition to test, if the first condition is false
* **Continue** 
  + Works like the break statement
    - Instead of forcing termination, force the next iteration of the loop to take place, skipping any code in between
* **“try” / “catch” constructs**
  + used to handle exceptions
* **Break**
  + Tells loop to exit immediately

/////////// Casting and Type Conversions ///////////////////////////////////

* **Cast**
  + A way of explicitly informing the compiler that you intend to make the conversion and that you are aware that data loss might occur
  + Must specify the type that you are casting to in parentheses in front of the value or variable to be converted
  + It takes away all numbers after the decimal point: neither rounding up nor rounding down. Called truncation
    - int z = (int)(5 / 2.5);
      * answer = 2 //converted a double into an integer
    - int x = (int)2.9;
      * answer = 2//converted a double into an integer
  + **Truncation**
    - Removing everything after the decimal
      * Doesn’t round up nor round down, just removes the decimal numbers
        + 2.9 = 2
        + 3.1 = 3
    - Division with two integers
      * Result of an arithmetic operation between two integers will always be an integer, even if it isn’t a whole number
        + double y = 7 / 3

answer = 2

should be 2.3333333333

decimal value is truncated

* + - * + C# assigns all values to the right of the equal sign to the variable
    - **2 solutions to work around Truncate to create a result with decimals**
      * Create a double you must change one integer into a double
        + double y = 7 / (double)3

answer = 2.33333333333333333

* + - * Append .0 to an integer :
        + Y = 7 / 3.0

answer = 2.3333333333333333333

* + - **Tips to avoid truncation**
      * If using a literal number and intend the result to be a double
        + Use .0 decimal point at the end of the integer
        + Casting

Mixing integers and doubles in arithmetic operations:

| **Left Side of Operator** | **Right Side of Operator** | **Message** |
| --- | --- | --- |
| int | double | double |
| int | int | int *(beware of truncation!)* |
| double | int | double |

//////////// Problem Solving //////////////////////////////////////////////

* **Four P’s**
  + Preparation
    - Understand the problem, and think of the solution
    - Use comments to go through each step
  + Plan
    - Plan out the solution
  + Perform
    - Perform the actions required for the solution
  + Perfect
    - Don’t have to perfect it, just work on the 4 p’s to make the program better and better
* **Input Validation**
  + Create rules about what the program can and cant accept
    - If user types something in, it should always make sense
* **Refactoring**
  + Change the code of a program without changing the apparent functionality
  + Makes the code easier to
    - Read
    - Maintain
    - Test
    - More efficient and run faster
    - Add additional code later
    - Make compliant with best practices
  + Compile Time Inference
    - Use var for variables when you can
  + Syntactic sugar
    - Features in the language that are completely redundant
    - Only thing they do is allow you to shorten the code
      * += operator

//////////// Exceptions //////////////////////////////////////////////////

* **Exceptions**
  + Occur because of programming mistakes
    - Exceptions are used to communicate to the programmer that there is a bug that needs to be fixed
  + Methods to say they weren’t able to complete successfully
    - Asked the method to do something it cant do
    - Method will decide it can’t do something successfully so it will throw an error
      * i.e. Unhandled Exception
      * i.e. System.FormatException
  + Read from the bottom up to follow the classes and to find the root of the exception
  + Is helpful when writing a program because it tells you where to create Input Validation
  + Shows you your weaknesses and helps you run an efficient program that wont crash as much
    - Smart to handle exceptions when stumbled upon
* **Handling Exceptions**
  + “try” / “catch()” constructs
    - Very powerful way to catch an exception and run the program with input validation
      * i.e. try { } catch (FormatException) { }
* **Variables in Exceptions**
  + i.e. throw new System.Exception(x + “,” + y + “ is outside the boundaries of the map.”)
    - In Main() you can access the exception message by declaring a variable for the exception: “ex”
    - i.e. catch(Exception ex) { }
* **System.Exception**
  + Provided by the .net framework
  + Means error occurred but shows nothing of the type of error
* **System.Format** 
  + Provided by the .net framework
  + Thrown from methods such as ent.parse
    - Which is expecting a string to have a certain format so that it can convert it to an integer
* **Can create custom Exceptions types**
  + Easy as creating a new class
  + They are typically very small
    - Can put them in the same file
    - All convention types in C# end with the word exception
    - i.e. class TreehouseDefenseException
  + All exception types must inherit from System.Exception
    - System.Exception is the most general exception that can be thrown, which is why any other Exception is inherited from System.Exception
    - All Exceptions thrown besides System.Exception are Subclasses of System.Exception
  + Can create an Exception **with no parameters (Default Constructor)**
    - **Automatically calls the default constructor of its base class**
    - i.e. public OutOfBoundsException()
  + Can create an Exception **with parameters and displays a message(must use “base”)**
    - i.e. public OutOfBoundsException(string message) : base(message)

//////////////// .Net Framework //////////////////////////////////////////

* .Net Framework
  + A large collection of code that can be used to build certain types of software
    - .Net Framework
      * Made up of hundreds of classes and methods
  + Reduces time to build software
* **System.Console.Write();**
  + **Writes to Console**
  + Print to screen
    - Provide information inside of parenthesis
    - Surround in double quotes “”
      * String
  + System:
    - Is an example of a Namespace
    - Name of the namespace the class is contained in
  + Console:
    - Is an example of a Class
    - Class the method is contained in
  + Write:
    - Is an example of a Method
* **System.Console.WriteLine();**
  + Always appends a new line character to the end of the string
    - Any subsequent output will start on a new line
* **System.Console.ReadLine();**
  + **Reads from the Console**
  + Console will start recording what the user types in until the user hits the “Return” key
    - Returns what the user typed back to the program
    - You can save what they typed back into a variable
  + Doesn’t need any parameters to run in this method
    - i.e. string name = System.Console.Readline();
      * This holds the value of what the user writes and hitting the “Return” key

////////// Exercise App ///////////////////////////////////////////////////

using System;

namespace Treehouse.FitnessFrog

{

class Program

{

static void Main()

{

var runningTotal = 0.0;

while (true)

{

// Prompt the user for minutes exercised

Console.Write("Enter how many minutes you exercised or type \"quit\" to exit:

string entry = Console.ReadLine();

if (entry.ToLower() == "quit")

{

break;

}

else

{

try

{

var minutes = double.Parse(entry);

if(minutes <= 0)

{

Console.WriteLine(minutes + " is not an acceptable value.");

continue;

}

else if(minutes <= 10)

{

Console.WriteLine("Better than nothing, am I right?");

}

else if(minutes <= 30)

{

Console.WriteLine("Way to go hot stuff!");

}

else if(minutes <= 60)

{

Console.WriteLine("You must be a ninja warrior in training!");

}

else

{

Console.WriteLine("Okay, now you're just showing off!");

}

// Add minutes exercised to total

runningTotal += minutes;

}

catch(FormatException)

{

Console.WriteLine("That is not valid input.");

continue;

}

// Display total minutes exercised to the screen

Console.WriteLine("You've entered " + entry + " minutes.");

// Repeat until the user quits

}

}

Console.WriteLine("Goodbye");

}

}

}

To Run Program using Mono Compiler:

**mcs Program.cs** //mono c-sharp compile Program.cs

**ls** //Shows both Program.cs and Program.exe files in directory

**mono Program.exe** //mono compiler to run new executable file

This final code challenge will require you to use almost everything we've covered in this course. You’ll write a program to celebrate completing this course by printing “Yay!” to the screen a given number of times.

Write a program that:

* Prompts the user to enter the number of times to enter “Yay!”. I've done this for you. Leave this as it is.
* Prints “Yay!” to the screen that number of times (Hint: You’ll need a loop for this. You’ll also need to keep track of how many times the loop has run).

You can print them all on the same line or on separate lines.

using System;

namespace Treehouse.CodeChallenges

{

class Program

{

static void Main()

{

var entry = 0;

try {

Console.Write("Enter the number of times to print \"Yay!\": ");

var entryParsed = int.Parse(Console.ReadLine());

if (entryParsed < 0) {

Console.Write("You must enter a positive number.");

}

else

{

entry += entryParsed;

}

}

catch(FormatException)

{

Console.Write("You must enter a whole number.");

}

var x = 0;

while (true) {

if (x < entry) {

Console.Write("Yay!");

x++;

}

else

{

break;

}

}

}

}

}

////////////// Object Oriented Programming ////////////////////////////////

* C# has features built into the language to support OOP and its four core principles
* **Four core principles**
  + Encapsulation
  + Inheritance
    - Enables you to create new classes that reuse, extend, and modify the behavior that is defined in other classes
    - The class whose members are inherited is called the base class
    - Class that inherits those members is called the derived class
    - Derived class can have only one direct bas class
    - Subclasses that inherit the attributes and behaviors of the more general classes
  + Polymorphism
  + Abstraction
* **Objects**
  + One purpose is to combine data and the methods that are closely tied to the data together in one elegant package
  + Object is an instance of a class
    - You can use the terms object and instance interchangeably
* Breaking it up into smaller parts
  + Could be reused
  + Code reuse is important
    - Makes it possible to write so much software so quickly
  + Open source code
    - Can integrate code easily with OOP
* Every value that a variable can take on is an object
  + Int
    - 1, 2, 3
  + Bool
    - True, False
  + String
    - “Yes” “No”
* **Class = Starting with a Capital letter**
  + Template for making individual objects of a particular type
  + Each individual **object that a class makes is called an instance of that class**
  + Can use the words type and class interchangeably
  + Example: Class is like a cookie cutter and the cookie is the instance of that class
  + Can create different objects from a class by changing the attributes of that class
* **Object**
  + Is an instance (Instantiation) of a type
  + Can create different objects from a class by changing the attributes of that class
  + **Instantiation** 
    - Creating an object from a class
    - By creating an object you create an instance of the class
      * i.e. Tower tower = new Tower();
      * (Tower: class) (tower: “instance”) (new Tower(): object)
* **Nouns**
  + Find the nouns in order to start programming
* Make new files for every class
  + Easier to keep track of, read and use
  + A standard convention
    - Makes it look more professionally done
* To fill out a class need to figure out what sort of attributes it should have
  + Think about the most minimal set of attributes the objects should have
* **Is**
  + Keyword that evaluates type compatibility at runtime
  + Determines whether an object instance or the result of an expression can be converted to a specified type
  + <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/is>
  + i.e. if (obj is Person) { //do something }

/////////// Fields ////////////////////////////////////////////////////////

* **Fields = Variable of any type that is declared directly in a class or struct and are members of their containing type**
  + Sustenance of a class
    - Make two objects of the same type unique
    - Contain objects data
  + **Public Field**
    - i.e. public string day;
  + **Private Field**
    - i.e. private DateTime date;
  + Creating a Map
    - Simplest object in the game
    - Divide the map into a grid
      * Width (field)
        + Number of grid squares it is wide
      * Height (field)
        + Number of grid squares it is high
    - A map can be described by the attributes for width and height
    - Width and Height become two fields in the class
    - **Fields are declared inside the class**
      * int Height;
      * int Width;
      * Every object that is instantiated from the map class can store both width and height
    - **Need to make fields accessible to other classes**
      * Every field in a class is assigned an accessibility level
      * Accessibility level determines the value stored in these fields and can be accessed or modified by classes other than the map class
    - **2 Accessibility Levels**
      * **Public**
        + Public fields can be accessed by any method in any class
      * **Private**
        + Private fields are only accessible to methods in the same class that they’re declared in
      * Can specify accessibility level using an **access modifier keyword**
        + **i.e. public int Width;**
        + **i.e. private int Height;**

However if you don’t specify “private” then the field is private by default

Even though you don’t have to type out “private” it is best practice to, since it tells others you thought about the accessibility level of the field and you’ve decided it should be private

* **How to access fields**
  + **First create a map object**
    - i.e. Map map = new Map();
  + **Access fields using dot notation**
    - i.e. map.Width = 8;
    - i.e. map.Height = 5;
  + Now that we have assigned each field a value let’s use them
    - i.e. int area = map.Width \* map.Height;
  + **Compile all files at once**
    - **i.e. mcs \*.cs**
      * uses an \* as a wild card
    - Tells compiler to compile all the files in the current directory that end with .cs
    - After the compiler compiles all files it turns them into one executable file
      * i.e. Game.exe
      * It chooses the first file name alphabetically
    - You can change the name of the executable file by
      * **Delete file** by typing **rm** (Name of file)
        + **i.e. rm Game.exe**
      * Tell compiler to **name the program something** **else:**
        + **mcs –out: (Desired File Name) \*.cs**

i.e. mcs –out: TreehouseDefense.exe \*.cs

* + - * + This will create the executable file name you want and will compile all of the .cs files

**i.e. TreehouseDefense.exe**

* + - **Compile as often as you can to find errors along the way**
      * More manageable and saves time
* **Readonly Property = Can only be used on fields (not properties)**
  + Make it read only
    - Cant change the fields after its created
      * i.e. public readonly int Width;
      * i.e. public readonly int Height;

///////// Constructors ////////////////////////////////////////////////////

* **Constructor Method**
  + Used to construct new instances of a class
  + Named same as the class they are in
  + Constructor is called when the object is created
  + **Don’t have return types**
    - They are only used to initialize the object
    - They don’t return anything
  + If a method doesn’t return anything we would write a void
    - However we don’t do that with constructors because they don’t return anything
  + **Make a constructor public**
    - **Allows other classes to access the constructor**
      * i.e. public Map(int width, int height)
  + **Readonly Property = Can only be used on fields (not properties)**
    - Make it read only
      * Cant change the fields after its created
        + i.e. public readonly int Width;
        + i.e. public readonly int Height;
    - **Only occur as part of the declaration or in a constructor in the same class**
    - Set only during object construction
* **Constructor Example:**

namespace TreehouseDefense

{

class Map

{

public readonly int Width;

public readonly int Height;

public Map(int width, int height) //Constructor with parameters

{

Width = width;

Height = height;

}

}

}

**Example:**

namespace TreehouseDefense

{

class Point

{

public readonly int X; //Make accessible to other classes (public)

public readonly int Y; //Make readonly because points don’t change

public Point(int x, int y) //Added a Constructor to initialize X and Y

{ // Uses parameters to set the value of X and Y fields

X = x;

Y = y;

}

}

}

////////// Conditional Operators/Boolean Logic Operators/////////////////////

* && = AND
  + Conditional-AND operator performs a logical-AND of its “bool” operands, but only evaluates its second if necessary

| **X** | **Y** | **X && Y** |
| --- | --- | --- |
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

* || = OR
  + Conditional-OR operator performs a logical-OR of its “bool” operands

| **X** | **Y** | **X || Y** |
| --- | --- | --- |
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

* ! = Negation Operator
  + Logical-negation operator is a unary operator that negates its operand. It is defined for “bool” and returns true if and only if its operand is “false”

| **X** | **!X** |
| --- | --- |
| true | false |
| false | true |

* **Example:**

namespace TreehouseDefense

{

class Map

{

public readonly int Width;

public readonly int Height;

public Map(int width, int height)

{

Width = width;

Height = height;

}

public bool OnMap(Point point)

{

bool inBounds = point.X >= 0 && point.X < Width && point.Y >= 0 && point.Y < Height;

bool inBounds = !(point.x < 0 || point.X >= Width || point.Y < 0 || point.Y >= Height);

return inBounds;

}

}

}

**Can refractor the inBounds variable:**

**return point.X >= 0 && point.X < Width &&**

**point.Y >= 0 && point.Y < Height;**

* **Zero-base counting**
  + **Off by one error**