# C#

Notes from: <https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/intro-to-csharp/hello-world>

<https://www.codecademy.com/learn/learn-c-sharp>

<https://docs.microsoft.com/en-us/dotnet/api/>

C# is a strongly and statically typed language. Strongly typed means that all variable types must be stated when the variable is declared.

Statically typed means that the data type of the variable is known at compile, rather than runtime.

## Console

The Console class contains various methods that allow the program to write or read to the console (cmd).

Console.WriteLine allows a message (or variable) to be written to the console.

Console.ReadLine will open a dialogue box and wait for the user to press enter before the value is taken to the program. It can then be processed such as assigned to a variable or used in calculations.

## Variables

Variables allow values to be stored, and assigned a name. This name can be referenced in code and the value will be used in output.

Variables are assigned by declaring a datatype (string, int, decimal, Boolean, etc.), a name, an = sign and then a value.

String ship = “Titanic”.

### Strings

Strings are groups of characters. This can be either a word, or sentence, or a paragraph. String values are enclosed in double quotes “”.   
  
Strings can be manipulated in a variety of ways in output.

Variables can be inserted in an output string by putting a + sign to a string.

“The name of this ship is “ + ship = The name of this ship is Titanic

“The name of this ship is “ + Titanic +”. And it will sink” = The name of this ship is Titanic. And it will sink

An easier way to add variables to strings is to use string interpolation. This is done by pre-facing a string with an $ before the “”, and then encasing the variables within {}. Multiple variables can be inserted using this method.

String ship = “Titanic”;

String ship2 = “Olympic”;

String ship3 = “Britannic”;

Console.WriteLine($”The three ships of the {ship2} care {ship2}, {ship}, and {ship3}”); = The three ships of the Olympic class are Olympic, Titanic, and Britannic.

### String Properties and Methods

Properties refer to characteristics of the string such as length, begin, end, etc.

Methods are actions that can be taken on a string.

Both of these are called by the appending them to the string name via a “.” and begin with a capital letter.

The difference is that a method has () after it’s name.

For example, .Length is a property, and .Replace is a method.

Ship.Length – Property

Ship.Replace() – Method.

String methods and properties can also be called in interpolation.

The property .Length can be used to display how many characters make up a string. It can be called by typing.Length after a string.

Ship.Length = 7;

Trimming a string.

There are three specific methods that can be used to removed extra spaces from a trim.

String.TrimStart() removes all blank spaces from the beginning of the string.

String.TrimEnd() removes all blank spaces from the end of the string.

String.Trim() is a shorthand method that removes blank spaces from the start and end of the string.

String.Replace() will search a string for a given substring, and then replace it with a new given substring.

The String.Replace takes two parameters, the string to find, and the string to insert.

Ship.Replace(“Titan”, “Olymp”) will change the string Titanic into Olympic.

There are two methods that modify the case of a string.

String.ToUpper() will write the string in ALLCAPS.

String.ToLower() will write the string in all lowercase letters.

Alongside the Replace method, there are methods to search a string and return results.

All the results below return Boolean (true or false) values indicating if the provided string exists within the string variable.

String.Contains(“Cheese”) will return if the variable or value contains the substring “Cheese” and false if it does not.

String.StartsWith(“Cheese”) will check the beginning of the string for the specified phase. If the phrase is found, it will return true, if it is not found it will return false.

String.EndsWith(“Cheese”) will search the end of the string for the specified phrase, returning true if found and false if not.

String.indexof(“a”) will return an integer showing what position the character occupies in the string. If a substring is used, it will return the index where the string starts (“anic” in the string “Titanic” will return 3. If the giving string occurs more than once, the position of the first occurrence is returned (“t” in “Titanic” will return 0 even though it also occurs as the third letter). This method will always return a -1 if the parameter is not in the string.

String.Substring(1) This method takes an int parameter and will return a sub-string starting at the given position and going to the end of the string. It is often combined with the index of property to find the index of the beginning of the string and then passed to the substring method for extraction.

#### Escaping Characters

There are times when certain characters reserved by the C# language need to be interpreted as literal characters. The act of denoting that a character should be read literally instead of for it’s reserved purpose is called escaping.

Escaping is typically done by pre-facing the character with an “\”.   
  
For example, quotation marks are used to denote the beginning and ending of a string, but if outputting dialogue (say as a quote or as a snippet of a story), the quotations need to be just that.   
  
string quote = ‘Darth Vader’s famous line is not \”Luke, I am your father\”. It is actually \”No, I am your father.\” in response to Luke’s declaration that Vader murdered his father.”

Escaping characters can also be used to insert formatting commands within strings. For example, \n will begin the text following the character on a new line of text.   
  
Console.WriteLine(“The saber cackled malevolently, bathing Vader’s helmet in blood-red light. \nLuke’s eyes widened in horror as the Sith’s words pierced his pain-addled brain. \nVader hadn’t murdered Anakin. Vader was Anakin.”);

This will print as

The saber cackled malevolently, bathing Vader’s helmet in blood-red light.

Luke’s eyes widened in horror as the Sith’s words pierced his pain-addled brain.

Vader hadn’t murdered Anakin. Vader was Anakin.

All string methods are case sensitive meaning that if the phrase “cheese” is in the value, but not “Cheese”, the method will return false.

### Numbers

C# has various numerical data types.

One of the most common is integers which represents a positive or negative whole number.

Any operation with integers will always result in integers regardless of what the result should be.

Integers are defined by the datatype int.

Int test = 1;

Integer variables can be defined as operations performed on other variables.

Int test2 = 3;

Int test3 = test1 + test2;

Console.WriteLine(test3); = 4.

There are various operations that can be performed on integers corresponding to the four mathematical operations in real-life arithmetic. Each is defined by a specific symbol.

+ defines addition operations.

* Defines subtraction operations.

/ defines division operations.

\* defines multiplication operations.

These operations have an order of precedence similar to real-world mathematics.

Multiplication and division are performed first, where as addition and subtraction are performed second.

a + b – c \* b / a = 1 + 3 – 4 \* 3/1 = 4 – 12 = -8

These operations can be pre-empted by encasing a specific operation(s) in ().

Interger division will always result in integers. If it would result in a decimal value, the decimal value is discarded only the whole number presented.

There is also a fifth type of operation known as modulo denoted by a %.

This operation is a form of division which returns the remainder instead of the quotient.

i.e. 14/5 = 2

14 % 5 = 4 as the remainder of 14/5 is 4.

Operations can only be performed on numbers of the same numeric type.

### Number Methods

The Math class offers several methods that can be performed on numbers of any data type.   
  
These methods are invoked through Math.MethodName(parameters);

Math.Abs() will return the absolute value of any given number (this will always be positive regardless of the sign and is a good way for countering methods that can’t take negative input).

MathAbs(-88) will return 88 as will Math.Abs(88)

Math.Sqrt() will return the square root of a number. This method will not accept negative numbers but in that case, the Math.Abs function can be used within the call.

Math.Sqrt(4); returns 2.

Math.Sqrt(-9) returns NaN (short for ‘Not a Number’)

Math.Sqrt(Math.Abs(-9)); returns 3.

Math.Min takes two parameters and returns the smallest of the two.

Math.Min(8, 18) returns 8.

Math.Max will return the max of two numbers.

Math.Max(8, 18) returns 18.

Math.Floor() will round down a number to the nearest whole number. If the number is a decimal, it will truncate the decimal.

Math.Floor(8.888) will return 8.

Math.Ceiling will round up to the nearest whole number. If the number is a decimal, it will be truncated.

Math.Ceiling(0.0888) will return 1.

Math.Pow takes two numbers and raises the first number to the power of the second (essentially multiplying the number by itself x times where x is the second number).

Math.Pow(2, 2) will raise the number to the power of 2 (multiplying 2 by itself one time) and will result in 4.

Math.Pow(4, 6) will multiply 4 by itself six times (4 \* 4 \*4 \*4 \*4 \*4) and will return 5696

### **Integer Methods and Properties**

Like strings, integers have both methods and properties.

Also, like strings these are appended to the end via a “.” and are capitalized.

Unlike the string examples which where called on variables with string values, some int properties can be called on an int object itself.

For example, Max value and MinValue.

Int.MaxValue is the maximum value an integer value can have which is 2,147,483,647.

Int.MinValue is the minimum value an integer value can have which is -2,147,483,648.

If a value exceeds the maximum value of an integer then there is an overflow condition.

If a value is below the minimum value of an intger, we have an underflow condition.

In the event that either of these conditions is achieved, the value displayed will starting counting from the corresponding opposite limit and go from there.

I.e. in an underflow condition it will go to the maximum limit and countdown, and the overflow condition will go from the minimum limit and count up.

### Doubles

Double times use double precision to represent floating point numbers. Floating points are useful for representing non-integral numbers that can be very large or very small in magnitude.

Double numbers can be decimals and values defined via the double data type will display decimal results for operations even if they are defined as integers.

Doubles also have a minimum and maximum value.

Double.MinValue = -1.79769313486232E+308

Double.MaxValue = 1.79769313486232E+308

The E+308 is Scientific Notation and denotes how many places to the right the Decimal place needs to be moved to find that value. In this case, the decimal would need to be moved 308 places to the right.

### Rounding Errors

C# can have rounding errors with Decimals. This is often indicated by repeating decimals.

For example 1/3 is written as 0.333333 ad infinitum. This is typically rounded to 1/3.

### Decimal

Decimals are similar to doubles in that they represent values with a radix point (.). However, they differ in that a Double value can move (hence the scientific notation) or float, while a decimal will remain fixed.

Decimal values are noted by using M or m to distinguish them from doubles.

Double a = 1.0

Decimal a = 1.0M

The value range of a decimal is also significantly smaller than double, but comes at greater precision.

Decimal.MaxValue = 79228162514264337593543950335

Decimal.MinValue = -79228162514264337593543950335

Console.WriteLine(1.0/3.0); = 0.333333333333333

Console.WriteLine(1.0M/3.0M); = 0.3333333333333333333333333333

### Converting

In order to perform operations on values of differing types, they must be converted.

There are two types of conversion, implicit and explicit.

Implicit conversion occurs by the system when it notices a conversion is either. An implicit conversion only occurs only when there is no data loss. If there would be data loss, the conversion fails, the program returns an error, and an explicit conversion must be used.

The following implicit conversion will not use.

Double test = 882.5;

Int test2 = test;

This fails because converting to int will cause the decimal to be dropped and the number truncated. C# will not allow a loss of data because this could potentially cause issues later on in the code.

The follow conversion will be successful.

Int test = 882;

Double test2 = test;

This works because no data loss will occur between the int and double.

### Explicit Conversion

There are two ways explicit conversion can occur.

The first is through parenthetical conversion and the second is through the Convert method.

Parenthetical conversion is invoked by specifying the type to be converted to in front of the value being converted.

The following example show parenthetical conversion:

Int cheese = (int) 882.5;

This can be done through the System.Convert class.

Type System.Convert.To[NumericType](ValuetobeConverted)

Int length = 882;

Decimal lengthRemainder = 0.5M;

Console.WriteLine($”The length of the Titanic was {System.Convert.ToDecimal(length) + lengthRemainder} feet.”);

You can also assign the conversion to a new variable.

Decimal length = System.Convert.ToDecimal(length);

## Control Flow and Repetition

C# offers several ways of controlling the conditions which a certain block of code will run and for the repeating of certain actions either under set conditions or a specific number of times without having to write the code multiple times.

These statements are generally grouped under if statements and loops.

### If Statements

If statements specify the condition(s) under which a specific block of code should run. For example if the user inputs a value, and an action should only be taken when it exceeds a certain amount, an if statement can be created for that condition.

An if statement consists of three major parts, the declaration (if), the condition (when this block should run) and the block (what to do when the condition is met). The declaration is always if, the condition is placed between two () and the conditions encased between {}.

If (value > 10) {

Console.WriteLine(“You are correct”);

}

The above condition checks that the value of a variable “value” if it exceeds 10, then it writes “You are correct.” to the console. If the value is less than 10, the statement is ignored and execution continues with the next block after the if.

### Conditions

Conditions are the most important part of an if statement as they control if a statement runs. An if statement must have at least 1 condition, but can have any number.

Conditions are a Boolean test, meaning that they evaluate to true or false. If the required condition(s) are met, the statement evaluates to true and the code runs. If not, it evaluates to false and the statement does not run.

Conditions consist of three major assets. The value, the comparison, and the check.

The value is what the condition should check and is most often a variable. This part is often a variable name.

The comparison indicates how the value should relate to the check. It can be any of several conditions but the most common are below:

|  |  |
| --- | --- |
| Symbol | Test |
| == | If the values are equal |
| <> or != | If the values are not equal |
| > | The value is greater than or exceeds the check |
| < | The value is less than the check. |
| >= | The value equals or exceeds the check |
| <= | The value is equal to or less than the check. |

The final part is the check which is the target the value must match as described by the condition for the if statement to activate.

The check can be any valid entry for the value’s data type.

### Multiple Conditions

A single if statement can have multiple conditions and those conditions can be joined in one of two ways, AND or OR.

AND (denoted by &&), means that all conditions must be met in order for the condition to activate.

If (value < 10 && value > 5)

This condition will trigger only if the value is between (but not including) 5 and 10.

OR (denoted by ||) means that any condition must be met for the condition to activate.

If (value > 10 || or value < 5)

This condition will trigger on any number less than 5, or over 10 but not on 5, 6, 7, 8, 9, or 10.

Conditions can also be grouped via (). This can be useful when for combining conditions in different ways such as a value between a certain threshold or a particular value

If(powerlevel = 9000 || (powerlevel > 18000 && powerlevel < 24000){}

This condition will activate if powerlevel is 9000 or if powerlevel is between 18000 and 24000.

An if statement can have conditions based on multiple variables as well.

IF(Class = ‘Olympic’ || Name <> ‘Lusitania’)

This will activate if the class value is Olympic or the name value is Lusitania. This can be useful when dealing with classes that have multiple attributes.

### Else

If Statements will only activate if their conidition(s) are met. However, if no conditions are met the statement will not execute.

Sometimes an action should always be taken whether the condition is met or not. C# provides this option with the keyword “Else”.

Else can be paired with if to specify that an action should be taken when the condition is not met.

Else is placed after an if, and covers in any cases that do not trigger the proceeding statement.

If (powerlevel > 9000){

Console.Write(“It’s over 9,000!”);

}

Else{

Console.Write(“It’s not over 9,000!”);

}

The above statement will write “It’s over 9,000” if the value of powerlevel exceeds 9000 but will write “It’s not over 9,000!” if the value is 9000 or less.

### Nesting Ifs

If’s can be nested inside each other to allow for a number of different processing options If certain requirements are met.

If statements are considered nested if they appear between the {}of another if statement.

If(weight<= 46.508){

If (weight = 46508){

String name = “Titanic”;

}

If(weight = 45504){

String name = “Olympic”;

}

If(weight = 30000){

String name = “Lusitania”

}

}

Else{

Console.WriteLine(“This weight is not one of any great liner of 1912 or earlier”)

}

### Ternary Operators

While if-statements are helpful, they can be cumbersome to write if there are several to write or if the condition is small.

Ternary operators act as a sort of short-hand for simple if statements that be quickly written on one line.

Ternary expressions are assigned to a variable and start with a variable declaration and assignment.

After the declaration the assignment is made to a Boolean condition in () just as the condition for an if-statement.

After the condition, a ? is used to denote the start of two possible outcomes. The value to be assigned if the statement is true comes first, followed by a colon, and then the value if the condition is false.

The following example shows a ternary expression that will display the status of an ocean liner as sunk if the ship is Titanic and afloat if anything else.

String ship = “Titanic”;

String status = (ship == “Titanic”) ? “sunk” : ”afloat”;

## Loops

Loops work similar to if statements in that they check conditions and execute code while those conditions are true.   
  
Loops differ in that they will continuously execute their code block while the condition is satisfied.

Depending on the type of loop, the loop may check the condition before or after execution. It is important to put check in a loop to track when the condition is no longer valid, otherwise the loop will go on forever and the program will never terminate or continue.

### While Loop

While loops will execute the code within as long as the condition is met. The program will only process this loop and other actions will be suspended.

While Loops consist of three main parts, the condition, the execution, and the check.

The condition is what continues the while loop. As long as the conditions evaluates to true (similar to if statements) the loop will execute.

The execution refers to the code that is run while the condition is met. This can be any valid C# statement.

The check is where the loop updates the condition. This will change based on the loop. For example, it might be updating a number’s value if the execution should only happen while it’s under a certain value, or if the user is supposed to enter a new value, it might mean rechecking the value.

A sample while loop is below.

Int counter = 0;

While(counter <= 10){

Console.WriteLine($”The current counter is {counter}.”);

Counter++;

}

The ++ operator will increase the counter variable by 1.

### Do While Loop

Another form of loop similar to wow is do while.

The while loop will always check the condition before executing. In the above loop, if counter is already over 10 before the loop is reached, the loop will not run.

Do while will always run the loop at least once since the condition isn’t checked until after the execution.

The difference between a do while and a while is that the do keyword appears before the execution (still contained in {}), with the while condition appearing immediately after. The check will still need to appear within the execution.

Int counter = 0;

Do

{

Console.WriteLine(counter);

Counter++;

} While(counter <= 10);

If the counter is 11 or higher, the above loop will print 11, increase the counter value to 12, and then check the condition, breaking the loop since counter is over 10.

### For Loops

A final type of loop is a for loop. A for loop works similar to a while loop in that it defines a conditions, an execution and a check, only runs while the condition is active, and runs the check to update the value after every run (if updated correctly). It differs in how it is defined.   
  
A for loop has a three part declaration encased in () after the keyword for. These parts are the for loop initializer, condition, and iterator.

The initializer declares the variable to keep track of the loop (loop variable) and defines it’s data type and initial value.

The condition determines for how long the loop should run (value is over or under a certain threshold, etc.).

The iterator determines how to change the value after each run.

The initializer and condition must be followed by semi-colons.l The iterator does not need to be.

After the for statement, the code is placed between {};

Like a while loop, the condition is checked before each execution meaning that once an iterator exceeds the condition, it will not run.

Unlike a while loop, an iterator is not needed within the body of the for loop since it is contained in the for loop definition.

For(int counter = 0; counter <= 10; counter++) {  
  
Console.WriteLine(Counter);

}

## Switch Statements

In the event there are specific code blocks we want to run for a number of different values, it can be wearisome writing an if else condition for each value.   
  
That’s where the Switch Statement comes into play. This statement allows a structure to be used that can be used to quickly specify cases and actions to take when they are met. It allows a default case to be specified in the event no cases are present.

The structure of switch statement opens with the switch statement and it’s parameter declared before an opening {

Each individual case is defined by the word case, and the value of the parameter this case should handle and followed by a :.

After the case statement, the code to be followed is written, and the case is ended with a break statement. Break is a special keyword which tells the program to exit the specified code block and move on to the next statement.

An example of a case statement is below. This will take the name of an ocean liner and display the year of its maiden voyage.

String oceanLiner = “Titanic”;

Switch (oceanLiner){

case “Olympic”:

Console.WriteLine(“1911”);

Break;

Case “Titanic”:

Console.WriteLine(“1912”);

Break;

Case “Britannic”:

Console.WriteLine(“1915”);

Break;

Case Default:

Console.WriteLine(“unknown”);

Break;

}

## Lists

Variables in C# are extremely useful but come with a major limitation. They can only store one value at a time. In order to list multiple values, even those that are logically related, a variable would be needed for each individual item.   
  
Enter Lists. Lists are a data structure that can be of any type and hold various values of that type.

A list must have it’s type defined when it is created. Unlike variables which are prefaced by their data type, Lists are pre-faced with the keyword var. The list is then given a name followed by an equal sign. After the name the new keyword proceeds “List” with a data type in <> (Often times examples use List<T> to refer to a generic type). Once the type is defined values are assigned to the list within {} (similar to loops and if statements). Values must be of the specified type, defined as properly for their type (decimals appended by M, strings in quotes, etc.) and are separated by commas.

Var ships = new List<string>{“Olympic”, “Titanic”, “Britannic”};

The above example creates a new list of type string, and assigns it three values Olympic, Titanic, and Britannic. Lists can have as many or as few entries as required.

### List Iteration

The easiest way to iterate through a list is to use yet another type of loop known as foreach.

Foreach loops are defined by defining a foreach condition of the type var. The var is named and a list is given to iterate through. Essentially, the variable assigns itself the first value in the list, executes the code in the loop, assigns itself the next value in the list, and repeats until all items in the list have been processed.

No conditions, or increments are necessary as the loop will only execute while there are unprocessed values.

Foreach(var ship in ships){

Console.WriteLine(ship);

}

In the above example, the loop will write the names of the three ships, and then stop.

### List Methods

Sometimes it can be impossible to fully define at list at creation. A list may grow as the program runs, or need to be created and then populated later (via a database or user input). Lists may also need to have values removed as they are no longer relevant (data may change, or the value is discarded after it has been processed). This can be accomplished through the use of methods.

Two important methods for lists are Add and Remove.

The add method will append the given value to the end of the list.

Ships.Add(“Lusitania”);

The remove method will remove the specified value from the List.

Ships.Remove(“Lusitania”);

### Access Individual Items in a List

Lists are 0 indexed, meaning that the first item in a list is assigned to place 0, the second is place 1, and so on.

In order to find the individual value at a specific place in a list simply type the list name, and the position in [].

Console.WriteLine(ships[0]);

This will display the value “Olympic”.

Console.WriteLine(ships[3]); would print “Lusitania”;

When using the add function, the new value will be assigned to x+1 where x is the last position of the list. In the above example, the end of ships was at position 2 and Lusitania was added to ships[3];

Similarly, removing an item will shift all entries after that position to the left one (x-1 where x is the item’s current position).

Removing “Titanic” from ships would cause “Britannic” to move into position 1 and “Lusitania” into position 2, dropping position 3. Adding “Titanic” back in afterwards would put into position 3.

### Finding an Index

Similar to getting an value by knowing the index, it is also possible to get the index knowing the value.   
  
This is accomplished via the “indexof” method. Simply feed the value as a parameter and it will return the index if the value is in the list. If the value is not in the list, the index will be -1.

Ships.IndexOf(“Lusitania”) ; returns 4.

Ships.IndexOf(“Andrea Doria”); returns -1.

## Array

Arrays function similar to lists in that they can store multiple values of the specified data type at once.

However, their definitions are slightly different.

Arrays are defined by declaring the data type, followed by [], the array name, an = and then the values of the array within {} separated by commas.

String[] OlympicClassShips = {“Olympic”, “Titanic”, “Britannic”};

Arrays can also be defined with the new keyword before the data type [] signifying that it’s being defined as an instance of the array class

String[] OlympicClassShips = new string[] {“Olympic”, “Titanic”, “Britannic”);

If an array’s length is known, but it’s values are not, it can be declared with the new syntax and have its length specified in the [].

String[] OlympicClassShips = new string[3]

A distinguishing feature of arrays over lists is that arrays length is static on creation. Once specified, an array’s length cannot be changed. Values can be changed, or set to default values, but they cannot be added or removed.

It is important to note, that an array must be declared and initialized in the same statement once a length is specified. If the length is specified without values assigned, each item will be given the default value, and values must be updated.

The above example creates an Array called OlympicClassShips with the values [null, null, null].

Each value in an array is assigned to a specific “slot” or index. Each index has an individual number starting at 0 and going up to the array length – 1 (the array created above will have indexes up to 2).

To access a specific value simply call the Array name with the specific index in [].

Console.WriteLine(OlympicClassShips[1]); returns Titanic.

To assign, or update a value in an array, just reference the index as above and assign the new value to it.

OlympicClassShips[2] = “Gigantic”;

### Array Properties and Methods

Arrays are a class type of a C# and as such, each Array has various properties and methods associated with it.

One such property is the Array’s length. This is simply the number of items in the array. Though indexes are numbered from 0, length is always the total number of indexes (OlympicClassShips will have a length of 3).

This can be referenced simply by calling ArrayName.Length. This will return integer equal to the length of the array. This value can be used as a condition for a loop, or to access the last value of a rank (OlympicClassShips[OlympicClassShips.Length-1); will return Britannic.

### Index Of

If a value is known to be an array, but it’s position is not, it can be found through the Index Of Method.

This is called via Array.IndexOf(). The method takes two arguments, an array (which can be of any data type), and a specific value (The data type of this value must be the same as the array). If the value is in the array, it will return the index of the value. If the value is not present, it will return -1.

Array.IndexOf(OlympicClassShips, “Olympic”) will return 0.

Array.IndexOf(OlympicClassShips, “Lusitania”) will return -1.

### Find

Find allows a rudimentary search to be run on an array (one dimensional), based on a given condition.

This will return the first occurrence that satisfies that condition. This can be assigned to a variable or even used as a call for another method.

Find takes two arguments, an array (of any datatype), and a condition. The condition is often written used Lambda Definitions (explained in more detail in a later section). These typically take the form of a variable (which item is temporarily fed into), and the condition.

Array.Find(OlympicClassShips, ship => ship.Length > 7); This will return the first value in the array that has a length longer than 7, in this case “Britannic”.

Find can be thought of as a for each loop that iterates through each item in the array until it finds a value that satisfies a condition and returns it.

As each value in the array is taken as an object of that data type, properties of that object can be applied in the condition (such as .Length on string values above).

### Sort

There may be occasions when an array needs to be in a specific order. The array sort method will reorder the array based on the data type (alphabetical for strings, and ascending order for numbers).

This is called via Array.Sort() and takes one parameter, an array of any data type.

For example, Array.Sort(OlympicClassShips) will re-order the array so it reads Britannic, Olympic, Titanic

This will re-order the array, but will not change the actual values within.

### Reverse

Similar to the sort, is the reverse method. When given an array, this method will reverse the order of the indexes. The only argument it takes is a single array.

Array.Reverse(OlympicClassShips) will return Britannic, Titanic, Olympic.

### Clear

If an array ever needs to be “reset”, the clear method will take the specified indexes, and reset them to their default values (null for strings, 0 for numerics).

The clear method takes 3 arguments, an array (of any data type), the starting index to clear, and the stopping point index. All indexes starting at the first given index and up to but not including the second will be set to the default values.

Array.Clear(OlympicClassShips, 0, 2) will return null, null, Britannic.

### Copy

It is also possible to clone an Array to another using the copy method. This can be useful if we want to preserve an array’s original values, or if we want to keep a clean copy before performing operations.

The copy method takes three arguments, a source array, a target array, and an integer. The method will take the first x values (equal to the given int), of the source array and copy them to the target array.

Array.Copy(OlympicClassShips, OceanLiners, 3) will copy the first three values of OlympicClassShips into OceanLners.

## Methods

Methods refer to a collection of statements that can be called via a signature. Methods typically refer to a specific action that may need to be called over and over again. For example, a ship class may have a method called sail which takes an origin, a destination and a list of hazards and calculate if the ships make it. This method can be called, feeding it the right information, and then returning if the ship made it or not.

### Method Signatures

Methods are typically named with every word of their name capitalized (Ship, BuyADog), etc.

After the name, the method is followed by ().

These () may or may not have a list of variable declarations within them. These variables are known as parameters and are passed as input when the method is called (between the parentheses). Parameters are declared like variables with a data type and a name. A method can have multiple parameters, separated by commas.

The combination of a method name and parameter types in order is called it’s signature.

The signature for our sailing method may look like this:   
Sail(string origin, string destination, bool hazards)

To call a method, simply place it’s signature in code and provide any necessary parameters.

Sail(“Southampton”, “New York”, false);

To create a method, after declaring its signature, use {} to enclose the code blocks.

Sail(string origin, string destination, bool hazards){

if (hazards == false){

Console.WriteLine($“The ship sailed safely from {origin} to {destination} because it encountered no hazards.”);}

Else{

Console.WriteLine($”Unfortunately, because there were hazards on its journey, the ship sank while going from {origin} to {destination}”);

}

}

### Parameters

Method parameters are a list of pieces of data the method requires in order to process its code.

Normally, if a method is called without all its parameters it will not run properly. This is because parameters defined as described above are required. Or perhaps, it is more accurate to say they require values to be manually passed as they have no way within the method to assign themselves a value.

Parameters can also be made optional. This is done by assigning a value to it in the method declaration.

When a value is assigned, the parameter can be safely skipped when the method is called and the assigned value will be used by default.

When some parameters have default values, and some do not, all non-default parameters must be created first as the below example.

The below example shows what happens when the hazard parameter is given a default value of false.

Sail (string origin, string destination, bool hazard = false){}

Sail(“Southhamption”, “New York”);

Returns: The ship sailed safely from Southhampton to New York because it did not encounter any hazards.

### Calling Parameters

There are two ways to call parameters. They can be called in order, as has been shown up until now, or they can be called by name.

When parameters are called in order, it is not necessary to specify the name of the parameter the given value belongs to. This is because the values are assigned in the same order as the Parameters are defined in the list. For example, when calling the sail method the parameters are always assigned in orders of Origin, Destination, hazards. If a parameter of the wrong data type is placed in the wrong order then it will cause an error.

Sail(false, “New York”, “Southhamption”); will raise an error because the bool value of false will be assigned to the string parameter of Origin.

When assigning values based on position, be wary of which parameters have default values. If a value is not provided for a default parameter, the value corresponding to its position will still be assigned to the parameter. This can cause an error if the data types do not match.

Example: Sail(string origin = “Southampton”, string destination = “Halifax”, bool hazards = false);

Sail(“New York”, false);

This will throw an error because false will be assigned to destination since it is the second value and the second parameter. Default values are ignored in order assignment.

The above can be avoided by using named declaration. Parameters are called by name, and can be called in any order. To call parameters by name, simply type the parameter name, a : and the value, separating parameters by commas.

Sail(hazards:false, Destination: “New York”, Origin:”Southhampton”);

The above would return the following message even though the parameters are assigned out of order.

Unfortunately, the ship sank while sailing from Southhampton to New York because it encountered hazards on its journey.

Named Parameters and ordered paramaters can be combined, however ordered parameters must come first. This is useful when we know there are default parameters we don’t want to override. Named declaration can be used with the parameters out of order.

When combining declaration methods, ordered parameters can only be used up to the first parameter to be named. Otherwise, an error will be thrown saying that values are assigned to parameters when they were already covered in ordered declaration.

For example, consider a version of sail that takes the name of a ship before origin.

Sail(“Southhampton”, false, ship:”Olympic”, destination:”Halifax”);

Will return an error saying ship has already found in order declaration (Southhampton)

For this, pure order declaration would need to be used.

This problem can be avoided if every value has a default parameter.

Sail(string ship = “Titanic”, string origin = “Southampton”, string destination = “New York”, hazards = false){}

In this version arguments can be freely using named declaration to assign the parameters in order, and the method will always work.

### Overloading Methods

As mentioned above, the combination of a methods name and its parameter data types in order is referred to as its signature.

However, the same method can have different signatures. In this section, two different signatures for the sail method were given. One with three parameters, and one with four.

Sail(string origin, string destination, bool hazards){}

Sail(string ship, string origin, string destination, bool hazards{}

Both of these definitions can exist side by side in a single program and can be used as necessary.

It is common for methods to have different versions to deal with certain inputs. The process of creating multiple signatures for a method is called “Overloading” a method.

The version of a method run is decided at run-time based on the arguments passed when it is called.

Keep in mind, that a signature considers only the Method name, and parameter types, not the names. It is possible for the method to be overloaded multiple times provided that each version has distinct data type combination. For example defining a signature of Sail(string, string, bool) and a signature of Sail(string, string, string) are valid because each signature is unique in terms of data types. However signatures of Sail(string origin, string destination, bool hazards) and Sail(string origin, string destination, bool sunk) are not valid since the name and parameter data types are identical.

This can be circumvented by moving the order of the parameters. This will affect how the method is called, but not necessarily how it is processed.

Sail(string origin, string destination, bool hazards){

If(hazards == false){

Console.WriteLine(“The ship sailed from {origin} to {destination} safely, because it did not encounter hazards”)}

Else{

Console.WriteLine(“Unfortunately, the ship sank on its way from {origin} to {destination} because it encountered hazards.”

}

}

Sail(string origin, bool hazards, string destination){

If(hazards == false){

Console.WriteLine(“The ship sailed from {origin} to {destination} safely, because it did not encounter hazards”)}

Else{

Console.WriteLine(“Unfortunately, the ship sank on its way from {origin} to {destination} because it encountered hazards.”

}

}

Though the method signatures are different, the processing, and data is exactly the same.

### Method Outputs

There are times when a method needs to return some piece of information to the part of the code that called it.

The type of value the method returns must match its return type. The return type is specified in the method declaration.

Public static void Sail();

In the above example the return type is void. This is used for methods that aren’t required to return any type of variable.

If the method were supposed to return a specific value, the data type of the returned value would match the return type. In the below example, Sail returns the string describing the joining.

Public static string Sail (string origin, string destination, bool hazards){

String didWeMakeIt

If (hazards == false){

didWeMakeIt = Console.WriteLine(“The ship sailed from {origin} to {destination} safely, because it did not encounter hazards”);}

Else{

didWeMakeIt = Console.WriteLine(“Unfortunately, the ship sank on its way from {origin} to {destination} because it encountered hazards.”);

}

return didWeMakeIt;

}

When its necessary to return a value, the value passed back to the calling code block must be pre-faced by the keyword return. The returned value can then be further processed in the program.

With the use of return, only a single value can be sent back for each method.

### Out keyword

Some situations require that more than one value be returned to the calling code block. When declaring a method, a parameter can be prefaced by the out keyword. This denotes a variable that will receive a value from the method and be assign it to itself. When specifying an out parameter, the parameter must be assigned a value in the method before the method closes. If a value is not assigned to the parameter, an error will be returned.

This version of Sail returns a message indicating the specifics of the voyage, as well as Boolean value as an out parameter that states if the ship is floating or sunk.

Sail(string origin, string destination, bool hazards, out bool sunk){

String didWeMakeIt;

If(hazards == false){

didWeMakeIt = “The ship sailed from {origin} to {destination} safely, because it did not encounter hazards”;

Sunk = false;

}

Else{

didWeMakeIt = “Unfortunately, the ship sank on its way from {origin} to {destination} because it encountered hazards.”;

Sunk = true;

}

Return didWeMakeIt;

}

}

### Calling a Method with Out

There are special steps that must be taken in order to successfully use a method with an out parameter.

Before calling such a method, a variable must be declared matching the type of the out parameter of the method.

This variable is then fed as an argument to the method call, prefaced by the keyword out.

If the value of this variable is checked after the method is called, the value will match the returned parameter of the method. In the below example, we take a ship name, call the Sail method and print If the ship sank or not.

String ship = “Titanic”;

Bool sinking;

Sail(“Southampton”, “New York”, true, out sinking);

Console.WriteLine($”Did {ship} sink? {sinking});

Returns: Did Titanic sink? True

### Method Shorthand

Typically, method structure requires many distinct parts including the definition, keywords, and various forms of punctuation. If certain conditions are met, different styles of short-hand can be used to make method calling issue. Two particular forms are called Expression-Bodied Definitions and Lambda Definitions.

### Expression-Bodied Definitions

Expression Body Definitions allow a single-lined method to be defined without the need for curly braces or return statements. However, the return type, access modifier, and parameters are still required and written per normal.

The definition of an Expression-Bodied definition is the same as a normal definition.

Public static void sink (int length)

After the method definition, the curly braces and returned statements are replaced with “=>”.

Public static void sink (int length) => Console.WriteLine($A boat of {length} feet will need at least {length \* 2} feet of water to fully sink.}.”);

Expression-Bodied Definitions are limited to only methods with one expression (line of code).

### Lambda Definitions

Lambda Definitions are used when a method is passed as parameter to another method (these are often used with arrays). Instead of writing a method and passing a call, the method can be defined similar to an expression-bodied definition.

The major difference between the two is their purpose and their anonymity. As described previously, Expression-Bodied definitions are used only with single line methods and are a short-hand way of defining methods that still need to be called traditionally.

Lambda Definitions are used when passing methods as parameters to other methods, defined in their call, can handle multi-line statements, and allow methods to remain nameless (which would prevent it from being referenced in other areas of the code).

To use a Lambda-based definition, define the parameters traditionally, and then the expression as would be done with an expression bodied definition.

Consider below an array of oceanliners, and an example that returns if any of the ships are lost. The method will take two inputs, an array of ships, and the sink method defined above. If sink is true for any items in the array, the method returns true.

String ships [] = {‘Olympic’, ‘Titanic’, ‘Mauretania’, ‘Lusitania’}

Bool sinkingShips(string [] ships, (string ship) => ship == “Titanic” || ship == “Lusitania”)

This will apply the second parameter method to all items in ships.

This definition can also be expanded to multi-line methods

Bool sinkingShips(string [] ships, (string ship) =>

{If ship == “Titanic” || ship == “Lusitania”{

Sunk = true;}

Else{ sunk = false; }

Return sunk; );

Lambda expressions can be further shortened if certain conditions are met.

If the data type can be inferred from the logic, then the data-type definition can be skipped.

String SinkingShips(string ship, (ship), => ship.ToUpper());

In the event that there is only a single parameter to the anonymous method, the () can also be skipped.

Strink SinkingShips(string ship, ship => ship.ToUpper();

## Useful Built In Classes

C# has several useful classes that offer built in functionality.

### Random

Random is used to generate a random integer and return it. This can be useful in calculations when needing to generate a fluctuating value or to simulate certain events (like a dice roll).

As random is a class, an object of that class must be created before the methods can be used. A new random object is defined by using the Random keyword, assigning a name to the object, using the = sign and then the new keyword before calling a Random constructer.

Random dice = new Random();

Random methods can be called by invoking them on the object, dot notation, and then the method name followed by ().

ObjectName.MethodName();

### Next Method

One particularly useful method of the random class is the next function. This method can be used to generate a random number between two integers. It is called by passing two integers and will return a random number between those values.

Dice.Next(1, 5) can possibly return 1, 2, 3, or 4.

## Classes

Each type of object (int, string, array, etc.) is an instance of a class. A class can be considered a data type definition which contains all the information needed to define and use said data type. Classes are made up of various “member” items. These include fields, properties, methods, and constructors.

### Fields

Fields are the most basic aspect of a class. If a class represents an object, then fields represent their characteristics and features. For example, if a ship is considered an object, then it might posses fields such as length, maxSpeed, service date, and lifeboats.

Fields are defined similarly to data types. Fields have a data type, a name, and a value. These can have default values assigned, or be accessed and set outside of the class by other programs.

It is conventional, but not required, to name fields similar to variables in all lowercase.

An example field is defined below:

Class Ship {

string name = “Titanic”;

int lifeboats = 20;

int maxSpeed = 23;

string type = “liner”; }

### Properties

Properties function as an extension of fields. Where fields hold the actual value pertinent to the class, properties function as a bodyguard or gatekeeper to fields so that they cannot be accessed directly. By convention, properties are named the same as their associated but with a capital letter (i.e. the property for lifeboats would be named Lifeboats).

The major distinction between properties and fields is that fields contain a value and properties contain methods that either return or change that value. Properties hold no distinct values themselves.

#### Property Methods

Each property will define two methods, get and set. The get method is intended to return the associated value, and is called whenever the value of the property is accessed.

The set method is used to set the value of the associated field. This is called automatically whenever the value of the property is updated.

A property is defined by the data type, followed by the name and {}.

Within the {}, get and set are defined each with their own {}. Set methods use the term value to represent the parameter passed to it to be assigned to the associated property. A basic “Lifeboat” property is created below.

#### Public int Lifeboat{

Get{return lifeboat;}

Set{lifeboats = value;}

}

#### Automatic Assignment

The above behavior is so commonly used in C# that it can be considered “the default” value of get and set methods.   
  
C# has included a shorthand method to this behavior called automatic assignment, allowing the property to be set up in just a few words.

Property int Lifeboat{get; set;}

This property will function the same as above. Whenever the property is called, it will access and return the value of lifeboat, and when assigned, will update lifeboat to be the passed value.

#### Customizing Property Methods

The definition described above is the most common, but by no means the only way to define properties. As properties are methods, they can be customized just as much.

For example, they can be written in a way to assign an altered value, or to return a message to the user.

Another common customization is to add a from of validation to the set method in order to make sure the values are acceptable.

For example, there may be a certain minimum or maximum value that must be observed. In the below methods, get will print the value to the Console, and set will check the weight of the vessel, ensure the new value of lifeboat matches the minimum legal requirement, and return a warning if not.

Public int Lifeboats{

Get{Console.WriteLine(lifeboats);}

Set{if(weight >= 10000 and value < 16)

{Console.WriteLine($“All ships weighing 10,000 or more tons must have at least 16 lifeboats by law. Please place at least 16 lifeboats on this ship. As this ship weighs{weight}, we recommend placing at least{(weight / 10000) \* 16} lifeboats on this ship.”);

}

Else{lifeboats = value;}

}

### Static Keyword

Whereas methods so far manipulate the properties of an object or an object itself, but with no recourse for modifying information about the class itself (number of instances created, general information of what the class represents, etc.)

The static keyword can be used to indicate a method or property associated with the class itself (such as finding out how many oceanliners are in a fleet).

By assigning the static keyword to something it is marked as associated with a class itself and not any specific instance (hence why static methods and properties cannot be referenced from a specific instance resulting in an error).

### Static Methods

Static methods can only access other static members (properties and fields). If a static method attempts to access a non-static member, it will not work properly.

### Static Constructors

Static Constructors are called when object is made from the type or before a static member is accessed. A static constructor is preceded by the static keyword, and does not accept an access modifier. Static constructors are often used to assign, access, or update values for static properties.

### Static Classes

Static classes are created when their main purpose is as a tool such as a utility or library. Classes such as Math and Console exist only to return and process information, but not to store data.

A static class cannot be instantiated like other classes (such as OceanLiner).

### Static Errors

Common static errors include errors CS0515, CS)120, CS0176

CS0515 “Static constructor cannot have an access modifier”. This occurs when a constructor is assigned both an access member and a static keyword. Simply remove the access modifier or the static keyword depending on the choice and the error should be resoloved.   
  
CS0120 “An object reference is required to access non-static field, method, or property”

This most often occurs when a non-static member, property, or method, is accessed by a static member or class, or accessed by a non-object. For example, calling Titanic.name will return no error. Calling OceanLiner.name will cause an error since this can only be accessed by a specific member. Simply make sure both the accessor and the item being accessed are both non-static.

CS0176: “Member cannot be accessed by with an instance reference; qualify it with a type name instead”. This occurs when a non-static item access a static item. Simply remove the reference or the static item.

## Interfaces

Interfaces are similar to classes in that they contain properties and methods and can define the outline of the object. However, unlike a class, an interface cannot have fields or constructors.   
  
The purpose of an interface is to define an outline of behavior and members required by any class using or implementing the interface. An interface serves to define a “blueprint” or minimum standard of what needs to be implemented by any class using the interface.

For example, each OceanLiner will have certain features and actions which each ocean liner will have. These may include a weight, length, maxspeed, class, and actions such as set sail, and launch life boats.

### Interface Construction

Interfaces are denoted by the interface keyword when creating them (similar to how a class is created).

Within the interface, properties and methods are defined, but are created empty. It is up to the class using the interface to implement each and every aspect of the interface as customized for the class.

An example interface for an OceanLiner interface is below.

Public interface OceanLiner{

Public int Weight{get; set:}

Public int Length{get; set;}

Public int MaxSpeed{get; set:}

Public string Class{get; set;}

Public void LaunchLifeboats(int lifeboats){}

Public void SetSail(string origin, string destination){}

### Interface Implementation

Any class can implement an interface. This is done via the colon-notation after the class declaration and placing the interface name after it. Once a class has implemented an interface it must include a definition for each item in the interface (though it does not necessarily have to use it).

The example below is an example of a ship that implents the OceanLiner interface.

Public class ship : OceanLiner

{

Public int Weight{get; set:}

Public int Length{get; set:}

Public int MaxSpeed{get; set;}

Public string Class{get; set;}

Public int lifeboats;

Public OceanLiner (int weight, int length, int MaxSpeed, string class){

This.Weight = weight;

This.Length = length;

This.MaxSpeed = maxSpeed;

This.Class = class;

Int lifeboats = ((weight / 10000) \* 16);

}

Public void LaunchLifeboats(int lifeboats){

For(int I = 1; I <+lifeboats; i++){

Console.WriteLine($”Lifeboat{i} launched successfully!”);

}

}

Public void SetSail(sting origin, string destination){

Console.WriteLine($”The ship set sail from {origin} to {destination} and arrived safely!”);

}

}

Classes implementing an interface can have additional functionality beyond that of the interface, but all members of the interface must be included and defined in the class.

### Interface Limitations

Interfaces merely define was must be included in a class, but it will not and cannot defined how it is used. Also, it is not possible to exclude things via an interface.

## Inheritance

It can be cumbersome to write duplicate code if we have a number of related objects that will utilize similar functions, fields, etc. One solution to this dilemma is to use inheritance. Inheritance allows code from one class to be passed on to and used by another without the need to re-write it in every single object.

### Inheritance Implementation

Once a class is created, any other class can inherit from it by including the classes’ name after a : in the class declaration.

Class Oceanliner : Ship.

Once an inheritance relationship is established, the class implementing inheritance is known as the sub class (or derived class), and the class it’s inheriting from is known as the super class or (the base class). A sub class generally has access to all fields, properties, methods, etc. of the base class and can access them as if they were their own. Assuming there’s a length property in both ship and ocean liner, then both statements below are valid.

Ship Titanic = new Ship();

Console.WriteLine(Titanic.Length);

OceanLiner Titanic = new OceanLiner();

Console.WriteLine(Titanic.Length);

## Access Modifiers and Inheritance

Every member in a class can be prefaced by an access modifier. Currently, there are three access modifiers in C#

Public: Anything with this modifier can be freely accessed by any code calling it.   
Private: Anything with this modifier can only be accessed the class itself. Even subclasses cannot access private members of a base class.

Protected: Anything with this modifier can be accessed only by the class itself and any sub-classes. This is the best method to use if something needs to be accessible outside of the class, but not to anything else.

Trying to access things with the private or protected modifier outside of the right classes will result in an error.

### Base Keyword

It is possible to access the members of a base class directly using the base keyword preceding a dot and the member name.

Ship Titanic = new Ship();

base.length

Typically, this isn’t used too much as inherited items can be accessed directly by the class.

However, one exception to this rule is in the base constructor. Base class constructors can be used by appending the colon after the subclass constructor and passing any relevant parameters to the base constructor.

Calling the base constructor will perform any actions in that constructor allowing for less work to be done in the subclass constructor.

Public OceanLiner(string name):base(name);

Even when the base keyword isn’t called, subclass constructors call a parameterless base constructor (be careful if a parameterless constructor is defined in the baseclass).

### Method Overrides

Typically when inheriting from a base class, the associated methods are stuck as defined in the base class, and trying to reimplement the method will result in an error about the inherited member being hidden.   
  
This can be circumvented by use of the virtual and override keyword. If a method is implemented using the virtual keyword, it can be redefined in the subclass by adding the word override after the access modifier. The below example sees a loadLifeboats method as defined in ocean liner and redefined in Ship

Public virtual void LoadLifeboats(){

Lifeboats = (weight/1000) \* 16;

}

Public override void LoadLifeboats(lifeboats){

this.lifeboats = lifeboats;

}

This can be another time when its useful to sue the base keyword.   
  
If for some reason, a method is overridden, but the base functionality is needed, the original can be called using base.LaunchLifeboats

### Abstract Classes and Members

While inheritance allows members to be accessed and utilized by an inherited class, it does not require them to be used or implemented. An interface would accomplish this, but that doesn’t provide any functionality.   
  
Using abstract classes and denoting some members as abstract allows the functionality of inheritance and interfaces.   
  
To utilize abstraction, the keyword abstract must be added in two places: the class definition, and the definition of any members that are to be denoted as abstract.   
  
To denote abstraction, put the keyword abstract after the access modifier of the member.

Abstract class OceanLiner{

Public abstract loadLifeboats();

}

The key to defining an abstract method is that a declaration, but not definition or {} are used.   
  
Any classes inheriting from an abstract class, will be forced to implement any incomplete definitions in their class definition.

## References

Every data type in C# is either a value type, or a reference type. The difference between them is in what data they store and how.   
  
Whenever a value is created it is stored in memory. This includes both variable values and class objects.

A value type, (including primitive types such as int, double, bool, etc) stores the actual value. Changing this value will update the actual characteristics of the item itself.

Reference types simply reference the value they’re assigned and are stored separately. Reference types include any classes, custom objects, etc created by a user.

If a reference is used to update a value, it will update the base object, and any other references to that object will reflect these changes.

For example, if we create an OceanLiner object and then create a new Ocean Liner object from the same one, all updates will apply to both.   
  
OceanLiner Titanic = new OceanLiner(50000, 882);

Titanc.LoadLifeBoats();

Console.WriteLine(Titanic.LifeBoats); = 80 (lifeboats are equal to weight divided by 10000 16)

OceanLiner Britannic = Titanic

Britannic.Weight = 60000;

Britannic.LoadLifeBoats(); //should be 96.

Console.WriteLine(Titanic.LifeBoats) = 96.

## Referential vs. Value Comparisons

When comparing two different objects (or traits, methods, etc), there are two types of comparison done based on the type.

If the value being compared is of a reference type (for custom obects, and non-primitive types, etc) it compares the memory of address of the two objects to see if they are the same. It does not look at the actual content of the values even if they are exactly the same.

OceanLiner Titanic = new OceanLiner(50000, 882, “White Star Line”);

OceanLiner Britannic = Titanic;

Console.WriteLine(Titanic == Britannic) returns true;

This is because Britannic is created from Titanic, and thus points to the same object (and memory address for that object) as illustrated in previous examples.

OceanLiner Titanic = new OceanLiner(50000, 882, “White Star Line”);

OceanLiner Britannic = new OceanLiner(50000, 882, “White Star Line”);

Console.WriteLine(Titanic == Britannic); returns false;

This is because while the values are identical at this point in time, Titanic and Britannic were created as distinct objects using the same parameters in the constructor. Since the objects are distinct, the references refer to distinct addresses causing the referential comparison to fail.

However, if we use the second example to compare primitive type fields, we can see how value comparisons work.

Console.WriteLine(Britannic.Weight == Titanic.Weight); Returns true.

Since Weight is an int type, and int is a primitive type, the actual values (what’s stored in memory) is compared and these are indeed the same.

## Type Referencing, Upcasting and Downcasting

Objects can be referenced as types that don’t match the type they were originally created as. However, this comes with two exceptions. First, the new reference must be of a type or interface somewhere along the object’s inheritance hierarchy. Second, the behavior available to this new reference is limited only to the behavior defined in the type of the reference.

Referencing a class as one of its superclasses is called Upcasting, and referencing it as one of its derived classes is called downcasting.

While a class can only inherit from one class at a time, it automatically inherits from any class, that its super class inherits from.

If the inheritance chain is MaritimeVehicle -> Boat -> Ship -> OceanLiner, the OceanLine class inherits from Martitime Vehicle, Boat, and Ship classes.

### Upcasting

Upcasting refers to creating a reference to a class of a type it is derived from.

In the example below, a reference to Titanic is made of the type ship. In this case, line is a unique member of the OceanLiner class.

OceanLiner Titanic = new OceanLiner(50000, 882, “White Star Line”);

Ship OlympicClass2 = Titanic;

We now have two references referring to the OceanLiner Object Titanic. While they reference the same object, the two references will not behave the same.

For example, both references can access the height and weight properties of Titanic.

Console.WriteLine(Titanic.Weight); 50000

Console.WrtieLine(Olympic2.Weight); 50000.

This is because both classes have a weight property (Weight is inherited from Ship).

However, if this is tried with the Line property, the result will be different.

Console.WriteLine(Titanic.Line); White Star Line.

Console.WriteLine(Olympic2.Line); Error!

The second line will not work properly because the Line property is implemented in OceanLiner and Ship has no knowledge of or any way to access that functionality.

There are typically no issues with upcasting so long as only functionality available to the base class is used.

### Downcasting

Creating a reference to an object as a derived class is called downcasting.

The below example shows downcasting.

Ship Olympic2 = new Ship(50000, 882);

OceanLiner Titanic = Olympic2;

This will most likely cause an invalidCastException. Unlike upcasting (which uses implicit casting), downcasting requires an explicit conversion from the base class to the derived class.

Explicit casting is done by putting the type to be converted to in parathenses before the object to be converted.

OceaLinerTitanic = (OceanLiner)Olympic2;

This can still lead to some errors, especially if there is new functionality introduced in the derived class.

### Interface Casting

Like superclasses and derived classes, references can also be made of type interface, so long as the interface is implemented by the class.

For example, if OceanLiner implements the Sailing Interface, the below is possible.

OceanLiner Titanic = new OceanLiner(50000, 882, “White Star Line”);

Sailing sunkenship = Titanic;

Again, functionalities of this reference is limited to the functionalities of the interface.

### Empty and Null References

Not every refence has to be to an object. Perhaps the reference needs to be filled later, or it needs to be emptied and potentially reused.

If a reference needs to be empty, it can set to null.

If a reference needs to have no value, it does not need to be assigned a value, but no operations can be performed.

## Object Class

The Object class is the base class of every class in C#.

Object contains methods such as Equals, ToString, and GetType().

Every class can make use of these and other methods.

The ToString method allows any object to be converted to a string and printed to the Console.   
  
Console.WriteLine utilizes the objects innate to String method. In fact, if ToString is overridden, then it will use that functionality instead.

### String Class

Strings are considered reference types but do value comparisons, meaning that they act as hybrid between value and reference types.

Strings are immutable meaning that they can’t be changed once created (but values can be updated)’

Each reference to a string is considered a new object and no other references are accepted.   
  
String s1 = “Titanic”;

String s2 = s1;

String s1 += “Sank”

Console.WriteLine(s2); Titanic

Console.WriteLine(s1); Titanic Sank

If we compare the strings, they are checked by value, not by memory reference.

Strings can be set to be null, or empty using String s1 = “”;