# C# Notes

## Naming Conventions

- For local variables: Camel Case (int rollNumber)
- For constants: Pascal Case (const int MaxZoom = 5)

## Primitive Types

- Integral Numbers
  - Byte
  - Short
  - Int
  - Long
- Real Numbers
  - Float
  - Double
  - Decimal
- Character – char
- Boolean – bool

> Note – double is used as a default data type when using real numbers in C#.

- Declare a float `float number = 1.2f`
- Declare a decimal `decimal number = 1.2m`

**Overflowing** – Overflow happens when we perform an operation with a data type and the result of this operation exceeds the size of a storage for this datatype.

```
byte number = 255;
number = number + 1 //this will output 0
```

## Type Conversion

1. **Implicit Type Conversion**

```
byte b = 1;
int i = b;

int i = 1;
float f = i
```

2. **Explicit Type Conversion**

```
int i = 1;
byte b = i //won't compile, so we use explicit type conversion

byte b = (byte)i;
```

3. **Non-Compatible Types**

```
string s = "1";
int i = (int)s; //won't compile

int i = convert.ToInt32(s);
or
int i = int.Parse(s);
```

## Non-Primitive Types

1. String
2. Array
3. Enum
4. Class

### Classes

Combines related variables (fields) and functions (methods).

**Declaring a class**

```
public class Person {
  public string Name;
  public void Introduce() {
    Console.WriteLine("Hi, my name is " + Name);
  }
}
```

**Creating Objects**

```
Person person = new Person();
//or
var person = new Person();
person.Name = "John";
person.Introduce(); //Hi, my name is John
```

## Arrays

An array is a data structure to store a collection of variables of the same type.

### Declaring an array

```
int[] numbers = new int[3];
numbers[0] = 1;
numbers[1] = 2;
numbers[3] = 3;

//or directly use object initialization syntax
int[] numbers = new int[3] {1, 2, 3};
```

## Strings

A sequence of characters.

### Creating strings

```
string name = "John";

//Using string concatenation
string name = firstName + " " + lastName;

//Using string format
string name = string.Format({0} {1}, firstName, lastName);

//Using string join
var numbers = new int[3] {1, 2, 3};
string list = string.Join(",", numbers); //1,2,3
```

### String Elements

```
string name = "John";
char firstChar = name[0]; //J

name[0] = 'm'; //cannot be done as strings are immutable
```

> Strings are Immutable. Once you create them, you cannot change them.

### Escape Characters

| Char | Description |
|---|---|
| \n | New Line |

| Char | Description |
|---|---|
| \t | Tab |
| \ | Backslash |
| ' | Single Quotation Mark |
| " | Double Quotation Mark |

**Verbatim Strings**

```
string path = "c:\\projects\\csharp\\folder";
//using verbatim strings
string path = @"c:\projects\csharp\folder";
```

### Enums

A set of name/value pairs (constants).

```
public enum ShippingMethod {
 RegularShipping = 1,
 RegisteredMail = 2,
 Express = 3
}

var method = ShippingMethod.Express;

//We can also specify the type for enum
public enum ShippingMethod : byte {

}
```

## Arrays

Represents a fixed number of variables of a particular type.

**Type of Arrays**

1. **Single Dimentional Arrays**

```
var numbers = new int[5];
var numbers = new int[5]{1, 2, 3, 4, 5};
```

2. **Multi Dimentional Array**

    - **Rectangular Array** - Number of columns for each rows are same.

```
var matrix = new int[3, 5]; //3 rows and 5 columns
var matrix = new int[3, 5]{
  {1, 2, 3, 4, 5},
  {6, 7, 8, 9, 10},
  {11, 12, 13, 14, 15}
};
```

- **Jagged Array** - Number of columns for each rows are different.

```
var array = new int[3][];

array[0] = new int[4];
array[1] = new int[5];
array[2] = new int[3];
```

## Array Properties and Methods

```
var numbers = new[] {3, 4, 5, 6, 7, 8, 9, 0};

//Length
Console.WriteLine(numbers.Length); // 8

//IndexOf()
var index = Array.IndexOf(numbers, 5)
Console.WriteLine(index); // 2

//Clear()
Array.Clear(numbers, 0, 2);

foreach(var n in numbers)
  Console.WriteLine(n); // {0, 0, 5, 6, 7, 8, 9, 0}

//Copy()
int[] another = new int[3];
Array.Copy(numbers, another, 3); // {3, 4, 5}

// Sort()
Array.Sort(numbers) // {0, 3, 4, 5, 6, 7, 8, 9}

//Reverse()
Array.Reverse(numbers) // {0, 9, 8, 7, 6, 5, 4, 3}
```

## Lists

List clas can be used to create a collection of different types like int, string, etc. It can be resized dynamically.

**Creating a List**

```
var numbers = new List<int>();
var numbers = new List<int>() {1, 2, 3, 4, 5};
```

**Lists Methods**

```
var numbers = new List<int>() {1, 2, 3};

numbers.Add(4); // {1, 2, 3, 4}
numbers.AddRange(new int[3] {5, 6, 7});

foreach(var number in numbers) {
  Console.WriteLine(number);
} // {1, 2, 3, 4, 5, 6, 7}

Console.WriteLine(numbers.Count); // 7
```

## Working With Strings

### Converting Strings to Numbers

```
string s = "1234";

int i = int.Parse(s);
int j = Convert.ToInt32(s);
```

### Converting Numbers to Strings

```
int i = 1234;

string s = i.ToString(i); // "1234"
string s = i.ToString("C"); // "$1,234.00", C is for Currency
string s = i.ToString("C0"); // "$1,234" , C0 removes digits after decimal. To use
1 decimal place, use "C1".
```

## Working With Files

**File** - provide static methods **FileInfo** - provide instance methods

> Both provide methods for creating, copying, deleting, moving and opening of files.

**Directory** - provide static methods **DirectoryInfo** - provide instance methods

**Path**

- GetDirectoryName()
- GetFileName()
- GetExtension()
- GetTempPath()

## Class

A building block of an application.

#### Anatomy of a Class

- Data (represented by fields)
- Behaviour (represented my methods/functions)

| **Post** |
|---|
| Title: string<br>Description: string<br>DateTime: DateTime |

| |
|---|
| Publish()<br>Like()<br>Comment(message) |

## Object

An instance of a class.

### Declaring Classes

```
public class Post {
  public string Title;

  public void Publish() {
    Console.WriteLine("Creating the post " + Title);
  }
}
```

### Creating And Using Objects

```
Post post = new Post();
//or
var post = new Post();

post.Title = "My first post";
post.Publish();
```

#### Class Members

- Instance: accessible from an object.

  ```
  var person = new Person();
  person.Introduce();
  ```

- Static: accessible from the class.

  ```
  Console.WriteLine("Hello");
  ```

## Constructor

A method that is called when an instance of a class is created. It has the same name as of the class name.

```
public class Customer {
  public Customer() {

  }
}
```

## Constructor Overloading

It is a technique of creating multiple constructors with a different set of parameters and the different numbers of parameters.

```
public class Customer {
  public Customer() {...}

  public Customer(string name) { ... }

  public Customer(int id , string name) { ... }
}
```

## Object Initializers

A syntax for quickly initializing an object without the need to call one of its constructors. This is needed to avoid creating multiple constructors.

Consider this simple Car class

```
public class Car {
  public string Name {get; set;}
  public string Color {get; set;}
}
```

Typically, we will create a class instance and then set its properties. But with object initializers, we can pass the public properties values during when we are creating the object without explicitly invoking the Author class constructor

```
Car car = new Car() {
  Name = "Audi",
  Color = "Red"
};
```

### Signature of a Method

- Name
- Number and types of parameter

```
public class Point {
  public void Move(int x, int y) {}
}
```

### Overloading Methods

Having a method with the same name but different signatures

```
public class Point {
  public void Move(int x, int y) {}
  public void Move(Point newLocation) {}
  public void Move(Point newLocation, int speed) {}
}
```

## Access Modifiers

A way to control access to a class and/or its members.

There are 5 different access modifiers in C#:

1. Private
2. Public
3. Protected
4. Internal
5. Protected Internal

> Private members are available only within the containing type.

> Public members are available anywhere. There is no restriction.

> Protected members are available, within the containing type and to the types that derive from the containing type.

> A member with internal access modifier is available anywhere within the containing assembly. It's a compile time error to access an internal member from outside the containing assembly.

> Protected Internal members can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly. It is a combination of protected and internal

> By default, class follows internal access modifier.

> We cannot declare a class as private, protected and protected internal.

> By default, class members are Private.

## Inheritance

Inheritance allows code reuse. Code reuse can reduce time and errors.

> We specify all the common fields, properties, methods in the base class, which allows reusability. In the derived class, we will only have fields, properties and methods that are specific to them.

> C# does not support multiple class inheritance.

```
public class ParentClass {
  public string name;
  public void PrintName() {
    Console.WriteLine(name);
  }
}

public class BaseClass : ParentClass {
  public string DOB;
}
```

### Method Hiding

In method hiding, we can hide the implementation of the methods of a base class from the derived class using the new keyword. In other words, we can redefine the method of the base class in the derived class by using the new keyword.

```
public class ParentClass {
  public void PrintName() {
    Console.WriteLine("Print name from parent");
  }
}

public class BaseClass : ParentClass {
  public void new PrintName() {
    Console.WriteLine("Print name from base");
    }
}
```

## Polymorphism

It allows us to invoke derived class methods through a base class reference during runtime.

In the base class the method is declared virtual, and in the derived class we override the same method. The virtual keyword indicates the method can be overridden in any derived class.

### Method Overriding

Creating a method in the derived class with the same signature as a method in the base class is called a Method Overriding. It is one of the ways to achieve Runtime Polymorphism (Dynamic Polymorphism).

```
class base_class
{
    public virtual void gfg();
}

class derived_class : base_class
{
    public override void gfg();
}

class Main_Method
{
 static void Main()
 {
    derived d_class = new derived_class();
    d.gfg();

    base_class b = new derived_class();
    b.gfg();
 }
}
```

### Types of Polymorphism

1. **Static or Compile Time Polymorphism** - In this, which method is to be called is decided at compile-time only. Method Overloading is an example of this.

2. **Dynamic or Runtime Polymorphism** - It is also known as method overridding. In this mechanism by which a call to an overridden function is resolved a runtime if a base class contains a method that is overridden

### Difference between Method Hiding and Method Overridding

**Method Overridding** - A base class reference variable pointing to the child class object, will invoke the overridden method in the child class.

**Method Hiding** - A base class reference variable pointing to the child class object, will invoke the hidden method in the base class.

## Encapsulation

The variables or data of a class are hidden from any other class and can be accessed only through any member function of own class in which they are declared.

> Encapsulation can be achived by declaring all the variables in a class as private and using C# properties in the class to set and get the values of variables.

```
public class Customer {
  private int _id;
  protected int key;

  public int ID {
    get { return _id ;}
    set { _id = value ;}
  }
}

public class CorporateCustomer : Customer {
  public void PrintID() {
    CorporateCustomer CC = new CorporateCustomer();
    CC.key = 101; // key is accessible as CorporateCustomer is derived class of
Customer
  }
}

public class MainClass {
  private static void Main() {
    Customer c1 = new Customer();
    Console.WriteLine(c1._id); // Error, as _id is not accessible outside the
Customer class due to its protection
    Console.WriteLine(c1.key); // Error, as key is not accessible due to
protection
    Console.WriteLine(c1.ID); //This is accessible as ID is public;
  }
}
```

**Properties** - A class member that encapsulates a getter/setter for accessing a field.

```
public class Person {
  private DateTime _birthdate;
  public DateTime BirthDate {
    get { return _birthdate }
    set { _birthdate = value }
    }
}
```

**Auto-Implemented Properties**

```
public class Person {
  public DateTime BirthDate { get; set; }
}
```

## List Collection

A list class can be used to create a collection of any type. For example, we can create a list of integers, string and even complex types.

The object stored in the list can be accessed by index.

> Unlike arrays, list can grow in size automatically.

```
class UnderstandingList
    {
        public static void Main(string[] args)
        {
            Customr customer1 = new Customr()
            {
                ID = 101,
                Name = "John",
                Salary = 4000
            };

            Customr customer2 = new Customr()
            {
                ID = 102,
                Name = "Jane",
                Salary = 5000
            };

            Customr customer3 = new Customr()
            {
                ID = 103,
                Name = "Joe",
                Salary = 3000
            };

            List<Customr> customers = new List<Customr>(2);

            customers.Add(customer1);
            customers.Add(customer2);
            customers.Add(customer3);
            Customr c = customers[0];

            Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", c.ID, c.Name, 
c.Salary);
            //ID = 101, Name = John, Salary = 4000
        }
    }

    public class Customr
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public int Salary { get; set; }
    }
```

**Iterating over List using foreach**

```
foreach(Customr c in customers){
  Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", c.ID, c.Name, c.Salary);
}
```

**Iterating over List using for loop**

```
for(int i=0; i<customers.Count; i++){
  Customr c = customers[i];
  Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", c.ID, c.Name, c.Salary);
}
```

**Insert data in List at a particular index**

```
customers.Insert(0, customer3);
// this will insert the customer3 at zero index and push down other items in the 
List
```

**Get postion of an element in the list**

```
customers.IndexOf(customer3);
// return 0

customers.IndexOf(customer3, 1);
// return 3. '1' in this case id the starting point

customers.IndexOf(customer3, 1, 2)
// Here it will start from 1 index and check 2 elements from the index 1
```

#### List Methods

1. **Contains()** - returns True if an item exists in the list, else returns false.

```
customers.Contains(customer3); // returns True
```

2. **Exists()** - Checks if an item exists in the list based on a condition.

```
customers.Exists(cust => cust.Name.StartsWith("J")); // returns True
```

3. **Find()** - returns the first matching item from the List based on a condition.

```
Customr f = customers.Find(cust => cust.Salary >= 4000);
Console.WriteLine("ID = {0}, Name = {1}", f.ID, f.Name);
// ID = 101, Name = John
```

4. **FindLast()** - returns the last matching item from the list based on a condition.

```
Customr l = customers.FindLast(cust => cust.Salary >= 4000);
Console.WriteLine("ID = {0}, Name = {1}", l.ID, l.Name);
// ID = 102, Name = Jane
```

5. **FindAll()** - returns all the matching item from the list based on a condition.

```
List<Customr> all = customers.FindAll(cust => cust.Salary >= 4000);
foreach(Customr c in customers){
  Console.WriteLine("ID = {0}, Name = {1}", c.ID, c.Name);
}
// ID = 101, Name = John
// ID = 102, Name = Jane
```

6. **FindIndex()** - returns the first matching index from the list based on a condition.

```
customers.FindIndex(cust => cust.Salary >= 4000); // return 1
```

7. **FindLastIndex()** - returns the last matching index from the list based on a condition.

```
customers.FindLastIndex(cust => cust.Salary >= 4000); // return 2
```

**Convert an array to a List** - Using ToList() function

```
Customer[] customers = new Customer[3];
customers[0] = customer1;
customers[1] = customer2;
customers[2] = customer3;

List<Customer> customerList = customers.ToList();
```

**Convert a List to an array** - Using ToArray() function

```
List<Customer> customers = new List<Customer>();
customers.Add(customer1);
customers.Add(customer2);
customers.Add(customer3);

Customer[] customerArray = customers.ToArray();
```

8. **AddRange()** - Add another list of items to the end of the existing List.
9. **GetRange()** - Get list of items from the List for the specified range. It expects 2 parameters, start index and the number of elements to be returned.
10. **InsertRange()** - Insert another list of items to the list at specified index.
11. **Remove()** - Removes the first matching item from the list.
12. **RemoveAt()** - Removes the item at the specified index of the list.
13. **RemoveAll()** - Removes all the items that matches the specified condition.
14. **RemoveRange()** - Remove a range of elements from the list. It expects 2 parameters, start index and the number of elements to be removed.
15. **Clear()** - Removes all the items from the list without specifying any condition.

**Sorting a List of Simple Type**

```
List<int> numbers = new List<int> { 0, 3, 2, 6, 4, 5, 8, 7, 9 };
numbers.Sort();

List<string> names = new List<string> { "Andrew", "Sam", "Paul", "Michael",
"Kelly" };
names.Sort();
```

**Reverse a List of Simple Type**

```
List<int> numbers = new List<int> { 0, 3, 2, 6, 4, 5, 8, 7, 9 };
numbers.Reverse();

List<string> names = new List<string> { "Andrew", "Sam", "Paul", "Michael",
"Kelly" };
names.Reverse();
```

> When we implement the sort funnction on a complex type, user-defined classes like Customer, we will get a runtime invalid operation exception. We have to implement IComparable Interface to sort the list of complex type.

> Sort function works on simple type because IComparable Interface is already implemented for these types.

**Sorting a List of Complex Types**

To sort list of complex types, it has to implement IComparable Interface and provide implementation for CompareTo() method. CompareTo() returns an integer.

```
public class Customer : IComparable<Customer>
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Salary { get; set; }
    public int CompareTo(Customer other)
    {
        //if (this.Salary > other.Salary)
        //    return 1;
        //else if (this.Salary < other.Salary)
        //    return -1;
        //else
        //    return 0;

        return this.Salary.CompareTo(other.Salary);
    }
}
```

We can also provide our own sort functionality by implementing the IComparer Interface.

For example, if we want to sort the customers by name

1. Implement IComparer Interface

```
public class SortByName : IComparer<Customer>
{
    public int Compare(Customer A, Customer B)
    {
        return A.Name.CompareTo(B.Name);
    }
}
```

2. Pass an instance of that class that implements IComparer Interface, as an argument to the Sort() method.

```
SortByName sortByName = new SortByName();
customers.Sort(sortByName);
```