Basic Csharp Programming

1. Object oriented programming
2. Introduction to Debugging, Unit Testing, coding standards and code analysis
3. Introduction to Value types and reference types and need for conversion
4. Access specifiers - public and private
5. Encapsulation
6. Static variables, methods, and constructors
7. Parameter types
8. Static Polymorphism
9. Relationships
10. Dynamic Polymorphism
11. Abstract
12. Interface

Advanced Csharp Programming

1. Exception Handling
2. Generics
3. Collections
4. CSharp language enhancements
5. Delegates
6. Delegates Enhancements
7. Extension Methods
8. LINQ - Language Integrated Query
9. File Handling
10. Asynchronous Programming
11. Memory Management

Extra

1. Input In C#
2. Coding Standards
3. Recursion
4. Regular Expression
5. Partial Classes and Methods
6. Type Conversion
7. Class Diagram
8. String Vs StringBuilder
9. SOLID Principles
10. Design Patterns
11. Pointer
12. IEnumerable & IEnumerator
13. IQueryable
14. Console.Write() vs Console.WriteLine()

----------------------------------------------------------------------------------------------------------------------------------------------------

1) Object oriented programming

--Object Oriented Programming(OOP) is a type of programming approach which enables the programmers to work with real life entities like Customer, Trainee, Employee, Company, Product, Food, Book, etc.

--C#, Simula, c#Script, Python, C++, Visual Basic .NET, Ruby, Scala, PHP etc. are some of the popular object-oriented programming languages.

--OOP helps a programmer in breaking down the code into smaller modules. These modules (classes) will have state(represented by attributes/variables) and functionality (represented by behavior/methods).

--These modules can then be used for representing the individual real life entities known as objects.

--E.g. - We can have a class named **Customer** to represent the state(variables) and functionality(methods) of **all** **customers**. Each individual customer can then be represented using an object of the Customer class.

--------------------------------------------------------------------------

**OOPS Example:**

class Customer {

//state(variables) //Instance variables

public String customerId;

public String customerName;

public long contactNumber;

public String address;

// Constructor to initialize variables

public Customer(String customerId, String customerName, long contactNumber, String address)

{

this.customerId = customerId;

this. customerName = customerName;

this. contactNumber = contactNumber;

this. address = address;

}

//functionality/behavior(methods)

public void displayCustomerDetails() {

System.out.println("Displaying customer details \n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Customer Id : " + customerId);

System.out.println("Customer Name : " + customerName);

System.out.println("Contact Number : " + contactNumber);

System.out.println("Address : " + address);

System.out.println();

}

}

class Tester {

public static void main(String[] args) {

// Object creation

Customer customer1 = new Customer();

// Assigning values to the instance variables

customer1.customerId = "C101";

customer1.customerName = "Stephen Abram";

customer1.contactNumber = 7856341287L;

customer1.address = "D089, St. Louis Street, Springfield, 62729";

// Displaying the customer details

customer1.displayCustomerDetails();

// Move the above statement immediately after the object creation

// statement and observe the output

}

}

----------------------------------------------------------------------------------------------------------------------------------------------------

**List of c# OOPs Concepts:**

The following are the concepts in OOPs-

1. Class
2. Object
3. **Abstraction:** (make class abstract, we **cannot create object of abstract class** but we can **inherit abstract class**)

Abstraction is a process of **hiding the implementation details** and showing **only functionality** to the user.

1. **Encapsulation:** (make variables private, Hides Variables i.e data members)

Encapsulation in c# is the process by which data (variables) and the code that acts upon them (methods) are integrated as a single unit. By encapsulating a class's variables, other classes cannot access them, and only the methods of the class can access them

**Encapsulation** is a process of **hiding the Data/variable**

1. Inheritance
2. Polymorphism
3. Association
4. Aggregation
5. Composition
6. Coupling
7. Cohesion

---In OOP, two classes can communicate with each other using their objects. An object may communicate with another object to use the functionalities provided by the other object. This is very helpful if we want to reuse the members of a class in another. Some of the types of relationships in c# are:

**Class**

A class is a user-defined blueprint from which objects are created. A class is used to represent the state(variables) and functionality(methods).

**Object**

* A object is an instance of a class.
* Each object has an identity, a behavior and a state.
* The state of an object is stored in fields (variables), while methods (functions) display the object's behavior.
* Objects are created at runtime from templates, which are also known as classes.

<ClassName> <ObjectVariableName>=new <ClassConstructorName>();

//new <ClassConstructorName>(); = object

//<ObjectVariableName> == object reference

**Methods:**

A method is a set of statements which represents the behavior of a class. It is used to implement a specific functionality. In c#, every method must be part of some class.

e.g.

accessSpecifier returnType methodName()

{

}

**Constructor:**

A constructor in c# is a special method that is used to initialize class variables at the time of object creation.

ctor + Tab + Tab

e.g.

accessSpecifier constructorName(parameters)

{

}

In C#, constructors can be divided into 5 types

**1) Default Constructor/Parameterless constructor:**

A constructor without any parameters is called a default constructor/Parameterless constructor.

**2) Parameterized Constructor:**

A constructor with at least one parameter is called a parameterized constructor.

**3) Copy Constructor:**

The constructor which creates an object by copying variables from another object is called a copy constructor. The purpose of a copy constructor is to initialize a new instance to the values of an existing instance.

public employee(employee emp)

{

name=emp.name;

age=emp.age;

}

employee emp1= new employee (emp2);

**4) Static Constructor:**

When a constructor is created using a static keyword, it will be invoked only once for all of the instances of the class and it is invoked during the creation of the first instance of the class or the first reference to a static member in the class. A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

**5) Private Constructor:**

When a constructor is created with a private specifier, it is not possible for other classes to derive from this class, neither is it possible to create an instance of this class.

----------------------------------------------------------------------------------------------------------------------------------------------------

2) Introduction to Debugging, Unit Testing, coding standards and code analysis

**Debugging**

--After you perform unit testing, you will be able to know whether there are bugs in the code or not. If bugs are present, the next step is to remove them.

--The process of identifying and removing bugs or errors from a computer program is known as debugging. Debugging can be done manually or by using different tools.

**Testing:**

**1) Unit Testing:**

Testing of single small units of code such as a method or a class is called unit testing.

Unit testing can be done manually or can be automated using tools.

**2) Manual Testing:**

Manual Testing is a type of software testing in which test cases are executed manually by a tester without using any automated tools

**3) Automated Testing:**

Automated Testing is the practice of automatically reviewing and validating a software product

**Coding standards**

Inorder to add the ruleset ,right -click on the Project ->Properties->Code Analysis

**Code Analysis**

Source code analyzing tools help in identifying violations in code.

----------------------------------------------------------------------------------------------------------------------------------------------------

3) Introduction to Value types and reference types and need for conversion

**Value Type & Reference Type:**

<https://www.tutorialsteacher.com/csharp/csharp-value-type-and-reference-type>

**Value Type:**

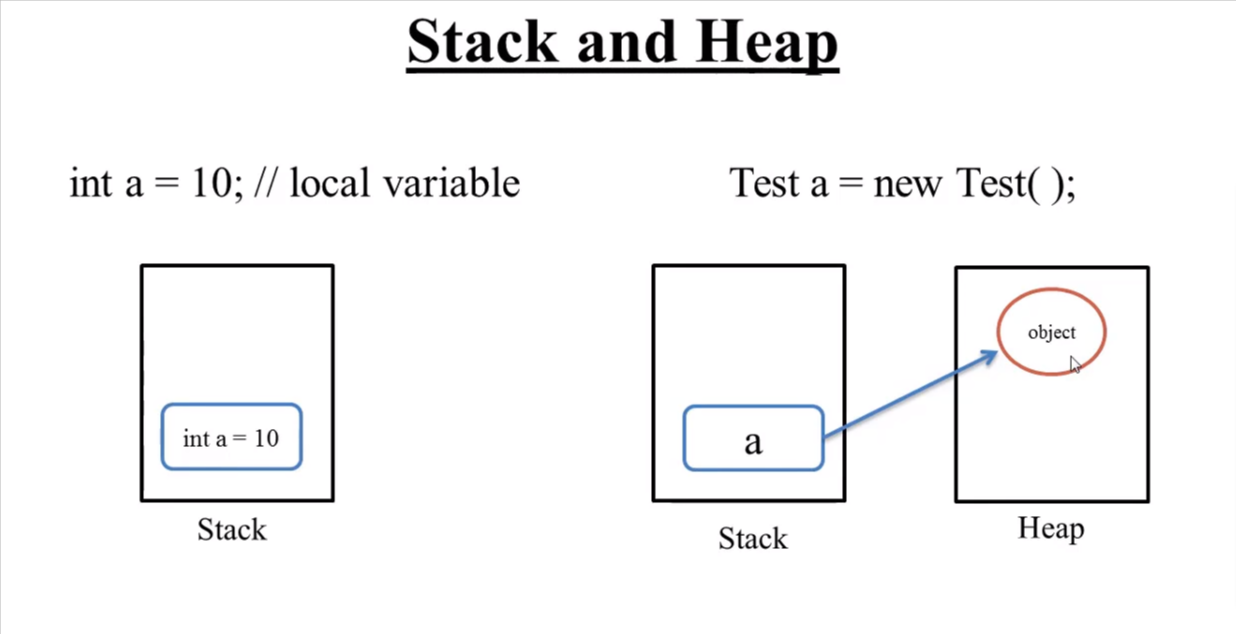
A Value Type holds the data within its own memory allocation.

Value Type variables are stored in the stack.

**Reference Type:**

Reference Type contains a pointer to another memory location that holds the real data.

Reference Type variables are stored in the heap.



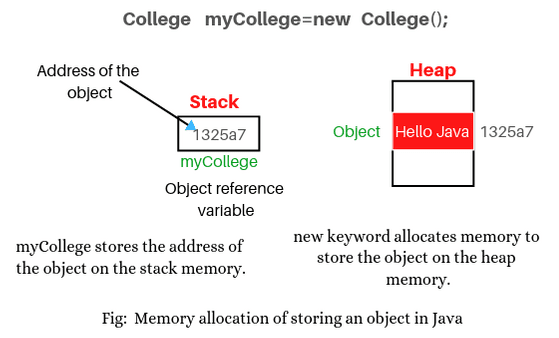
Here **a** points to the **object[ new Test(); ]** so here **a** is the reference variable which stores the address of

the **object[ new Test(); ]**

<https://www.scientecheasy.com/2020/06/how-to-create-object-in-java.html/>

College myCollege = new College();

System.out.println(myCollege.hashCode()); // It will display hash code stored in myCollege.



**C # DATA TYPE**

|

---------------------------------------------------------------------------------------------------------------------

| |

**Value Type(stack) Reference Type(heap)**

| |

------------------------------------------------ ------------------------------------------------------------

| | | |

**User defined Type** **Built in Type** **User defined Type Built in Type**

Int short struct object class

long char enum string Interface

float bool dynamic array

double delegate

byte

decimal

**Call By Value & Call By Reference:**

**Call By Values:**

While calling a function, we pass values of variables to it. Such functions are known as “Call By Values”.

**Call By Reference:**

While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as “Call By References.

**1) Struct**

Struct type **directly stores the data of the struct** ,not the address of the memory location.

As they are value types and does not require heap allocation. This can make a larger difference in memory allocation compared to classes.

Ex.

struct Rectangle {

public int length;

public int width;

public int Area() {

return length \* width;

}

}

class Program {

static void Main(string[] args) {

Rectangle rect = new Rectangle();

rect.length = 5;

rect.width = 10;

Console.WriteLine("Area of the rectangle is: " + rect.Area());

}

}

**2) Enum**

An enum is a special "class" that **represents a group of constants** (unchangeable/read-only variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the enum items with a comma

Ex.

enum DaysOfWeek {

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

class Program {

static void Main(string[] args) {

DaysOfWeek today = DaysOfWeek.Monday;

switch (today) {

case DaysOfWeek.Monday:

Console.WriteLine("Today is Monday.");

break;

case DaysOfWeek.Tuesday:

Console.WriteLine("Today is Tuesday.");

break;

case DaysOfWeek.Wednesday:

Console.WriteLine("Today is Wednesday.");

break;

case DaysOfWeek.Thursday:

Console.WriteLine("Today is Thursday.");

break;

case DaysOfWeek.Friday:

Console.WriteLine("Today is Friday.");

break;

case DaysOfWeek.Saturday:

Console.WriteLine("Today is Saturday.");

break;

case DaysOfWeek.Sunday:

Console.WriteLine("Today is Sunday.");

break;

}

}

}

**Boxing and Unboxing**

Boxing and unboxing in C# allows developers to convert .NET data types from **value type to reference type** and vice versa.

Converting a value type to a reference type is called **boxing.**

Converting a reference type to a value type is called **unboxing.**

e.g.

class Test

{

static void Main()

{

int i = 1; //Value Type To

object o = i; // boxing //Reference Type To

int j = (int)o; // unboxing //Value Type

}

}

----------------------------------------------------------------------------------------------------------------------------------------------------

4) Access specifiers

They are used to specify access levels to control the visibility of a class and its members.

There are 4 such access modifiers in c#:

**1) Public:** The code is accessible for all classes

**2) Private:** The code is only accessible within the same class

**3) Protected:** In C#, the protected access specifier is used to specify that a member (field or method) of a class is accessible within the class and its derived classes, but not outside of the class hierarchy.

**4) Internal:** The code is only accessible within its own assembly, but not from another assembly.

----------------------------------------------------------------------------------------------------------------------------------------------------

5) Encapsulation

(make variables private, Hides Variables i.e data members)

Encapsulation in c# is the process by which data (variables) and the code that acts upon them (methods) are integrated as a single unit. By encapsulating a class's variables, other classes cannot access them, and only the methods of the class can access them.

**Properties**

A property is like a **combination of a variable and a method**, and it has two methods: a get and a set method

Private variables can only be accessed within the same class (an outside class has no access to it). However, sometimes we need to access them - and it can be done with properties.

**Syntax:**

accessSpecifier dataType propertyName{get;set;};

e.g.

class Person

{

private string name; // field

public string Name // property

{

get { return name; } // get method

set { name = value; } // set method

}

}

The get method returns the value of the variable name . The set method assigns a value to the name variable.

------------------------------------------------------------------------------------------------

Now we can use the Name property to access and update the private field of the Person class:

class Person

{

private string name; // field

public string Name // property

{

get { return name; }

set { name = value; }

}

}

class Program

{

static void Main(string[] args)

{

Person myObj = new Person();

myObj.Name = "Liam";

Console.WriteLine(myObj.Name);

}

}

------------------------------------------------------------------------------------------------------

**Automatic Properties (Short Hand):**

C# also provides a way to use short-hand / automatic properties, where you do not have to define the field for the property, and you only have to write get; and set; inside the property.

The following example will produce the same result as the example above. The only difference is that there is less code:

class Person

{

public string Name // property

{ get; set; }

}

class Program

{

static void Main(string[] args)

{

Person myObj = new Person();

myObj.Name = "Liam";

Console.WriteLine(myObj.Name);

}

}

------------------------------------------------------------------------------------------------

**Way 1:**

private int categoryId; //field

public void SetCategoryId(int categoryId)

{

this.categoryId = categoryId;

}

public int GetCategoryId()

{

return this.categoryId;

}

===========================================

**Way 2:**

private int categoryId;

public int CategoryId

{

get { return categoryId; }

set {categoryId = value; }

}

===========================================

**Way 3:**

private int categoryId;

public int CategoryId { get ; set; }

===========================================

**Way 4:**

public int CategoryId { get ; set; }

**Indexer**

An indexer is **a special type of property** that **allows a class or a structure to be accessed like an array** for its internal collection.

C# allows us to define custom indexers, generic indexers, and also overload indexers.

An indexer can be defined the same way as property with **this** keyword and square brackets [].

<Access Specifier> <Return Type> this[<parameter type> index]

{

get{

// return the value from the specified index of an internal collection

}

set{

// set values at the specified index in an internal collection

}

}

**Ex.1**

class ToyBox

{

private string[] cars = new string[3] {"red car", "blue car", "green car"};

public string this[int index]

{

get { return cars[index]; }

set { cars[index] = value; }

}

}

**Ex.2**

public class MyClass

{

private string[] \_data = new string[5];

// Define the indexer

public string this[int index]

{

get { return \_data[index]; }

set { \_data[index] = value; }

}

}

class Program

{

static void Main(string[] args)

{

MyClass myObj = new MyClass();

myObj[0] = "Hello";

myObj[1] = "world";

Console.WriteLine(myObj[0] + " " + myObj[1]);

}

}

In this example, we define a class MyClass with an indexer that allows us to access the elements of \_data array from outside the class. The indexer takes an integer index and returns the string at that index.

In the Main method, we create an instance of MyClass and use the indexer to set the first two elements of \_data. We then use the indexer to retrieve those values and print them to the console.

The output of this code will be:

Hello world

----------------------------------------------------------------------------------------------------------------------------------------------------

6) Static variables, methods and constructors

In simple words, the static keyword in C# is used to create something that is shared among all instances of a class, rather than being specific to individual instances.

the static keyword allows us to create shared elements among instances, access members without creating objects, optimize memory usage, define utility methods, and declare constants that are common to the entire class.

**1) Static Class:**

--A static class is declared with the help of static keyword. A static class can only contain static data members, static methods, and a static constructor. It is not allowed to create objects of the static class. Static classes are sealed, means one cannot inherit a static class from another class.

**2) Static Variable:**

--A static variable is declared with the help of static keyword. When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level. Static variables are accessed with the name of the class, they do not require any object for access.

E.g. private static float deliveryCharge; //Declaration of static variable

private static float deliveryCharge = 1.5f;

**3) Static Constructor:**

--A static constructor is declared with the help of static keyword. Static Constructor has to be invoked only once in the class and it has been invoked during the creation of the first reference to a static member in the class. A static constructor is initialized static fields or data of the class and to be executed only once.

**Points To Remember:**

* It can’t be called directly.
* When it is executing, then the user has no control.
* It does not take access modifiers or any parameters.
* It is called automatically to initialize the class before the first instance created.

**4) Static Method:**

--Static methods are methods prefixed with the static keyword. These methods can be accessed without an object of the class. Similar to static variables, they are accessed using the class name. A static method can be invoked without creating an instance of a class.

--A static method can only access static variables and cannot access instance variables.

E.g.

public static float getDeliveryCharge() {

return deliveryCharge;

}

public static void setDeliveryCharge(float deliveryCharge) {

Customer.deliveryCharge = deliveryCharge;

}

**Limitation of using static keyword:**

--static keyword cannot be used by indexers, finalizers, or types other than classes.

--A static member is not referenced through an instance.

--In C#, it is not allowed to use this to reference static methods or property accessors.

--In C#, if static keyword is used with the class, then the static class always contain static members.

----------------------------------------------------------------------------------------------------------------------------------------------------

7) Parameter types

**1) Named Parameters:**

we can specify the value of a parameter by parameter name regardless of its ordering in method.

**Ex.**

private void SomeObject(double width, double height, double length, double weight, double volume, double density, bool isMetal, bool isConductor)

{

//do something

}

static void Main(string[] args)

{

Program p = new Program();

p.SomeObject(10.0, 20.0, **isConductor: true, isMetal: true, length: 50.0, weight: 100.0, density: 50.0, volume: 200.0**);

}

**2) Ref Parameters:**

C# includes ref and out are keywords, which help us to pass the value type variables to another function by the reference.

e.g.

class Program

{

static void Main(string[] args)

{

int myNum = 10;

ProcessNumber**(ref myNum);** //use ref to pass the parameter by reference

Console.WriteLine(myNum);

Console.ReadLine();

}

public static void ProcessNumber**(ref int num)**

{

num = 100;

}

}

//output is 100 if we use 'ref' keyword in method parameter

//output is 10 if we do not use 'ref' keyword in method parameter

**3) Out Parameters:**

The out keyword can be used with variables and method parameters. The out paramters are always passed by reference for both, the value type and the reference type data types.

e.g.

class Program

{

static void Main(string[] args)

{

int a; // declare variable without initialization

OutParamExample**(out a)**;// calling method with out keyword

Console.Write(a);// accessing out parameter value

}

public static void OutParamExample**(out int x**)

{

x = 100;

}

}

**4) Default or Optional Parameters:**

As the name suggests optional parameters are not compulsory parameters, they are optional.

e.g.

static public void scholar(string fname, string lname, **int age = 20, string branch = “Computer Science”)**

In the above example, int age = 20 and string branch = “Computer Science” is the optional parameter and has its default value.

Note:

When you do not pass the value of these parameters, then the optional parameters use their default value. And when you pass the parameters for optional parameters, then they will take the passed value not their default value.

**5) Dynamic Parameters:**

In C# 4.0, a new type of parameters is introduced that is known as a dynamic parameter. Here the parameters pass dynamically means the compiler does not check the type of the dynamic type variable at compile-time, instead of this, the compiler gets the type at the run time. The dynamic type variable is created using a dynamic keyword.

**6) Value Parameters:**

It is a normal value parameter in a method or you can say the passing of value types by value. So when the variables are passed as value type they contain the data or value, not any reference. If you will make any changes in the value type parameter then it will not reflect the original value stored as an argument.

**7) Params:**

It is useful when the programmer doesn’t have any prior knowledge about the number of parameters to be used. By using params you are allowed to pass any variable number of arguments. Only one params keyword is allowed and no additional Params will be allowed in function declaration after a params keyword. The length of params will be zero if no arguments will be passed.

public void ExampleMethod(**params int[] numbers**)

{

foreach (int num in numbers)

{

Console.WriteLine(num);

}

}

ExampleMethod(1, 2, 3); // prints 1, 2, 3

ExampleMethod(4, 5, 6, 7); // prints 4, 5, 6, 7

ExampleMethod(); // prints nothing

----------------------------------------------------------------------------------------------------------------------------------------------------

8) Static Polymorphism

Polymorphism is the **ability of an object to take different forms**, i.e. a single action that can be performed in different ways. So, polymorphism means many forms.

Polymorphism is of two types:

**1) Static polymorphism:**

Polymorphism that gets **Executed during compile time** is known as **static polymorphism or compile time polymorphism**. This polymorphism is achieved using overloading of the methods in the **same class**, called as Method overloading.

**Method overloading:**

Method overloading allows the programmer to have multiple methods with **the same name in the same class, but differing in their signature.**

**Constructor overloading:**

Constructor overloading allows the programmer to have multiple Constructor with the **same name in the same class, but differing in their signature.**

**Signature can differ by-**

* the number of parameters
* the data type of parameters
* the order of the parameters

**Note:** We cannot overload methods by their return type, i.e., two or more methods are not overloaded if they differ only in their return type.

public class Calculator {

public int Add(int a, int b) {

return a + b;

}

public double Add(double a, double b) {

return a + b;

}

public int Add(int a, int b, int c) {

return a + b + c;

}

public double Add(double a, double b, double c) {

return a + b + c;

}

}

// Usage

Calculator calculator = new Calculator();

int sum1 = calculator.Add(2, 3);

Console.WriteLine("Sum 1: " + sum1); // Output: Sum 1: 5

double sum2 = calculator.Add(2.5, 3.5);

Console.WriteLine("Sum 2: " + sum2); // Output: Sum 2: 6

int sum3 = calculator.Add(2, 3, 4);

Console.WriteLine("Sum 3: " + sum3); // Output: Sum 3: 9

double sum4 = calculator.Add(2.5, 3.5, 4.5);

Console.WriteLine("Sum 4: " + sum4); // Output: Sum 4: 10.5

----------------------------------------------------------------------------------------------------------------------------------------------------

9) Relationships

**Association[Uses-a]**

|

--------------------------------------------------------------------------------------------------------------------------

| |

**Aggregation[Has-a] Composition [Has-a]**

(Weak Association) (Strong Association)

**Association (uses-a relationship)**

Association, also known as **uses-a relationship** exists between two classes when one object makes use of another object for its functionality. Here, both the objects can exist independently.

So, do you know what a group of things that have something in common is called? For example, a group of animals that can fly like birds, butterflies, and bees. We call this group an association.

In programming, we can also create associations between different things. In C#, we can create what we call an association between two or more objects.

Let's say we have a class called "Person" and another class called "Car". We can create an association between these two classes by making it possible for a person to own a car. So, in the Person class, we can create a property called "Car" which will hold a reference to a Car object.

Here's an example of how we can create this association:

class Person {

public string Name { get; set; }

public int Age { get; set; }

public Car Car { get; set; }

}

class Car {

public string Make { get; set; }

public string Model { get; set; }

}

So, now we have created an association between a Person and a Car. We can create an instance of the Person class and assign a Car object to the Car property like this:

Person john = new Person();

john.Name = "John";

john.Age = 25;

Car johnsCar = new Car();

johnsCar.Make = "Toyota";

johnsCar.Model = "Camry";

john.Car = johnsCar;

Now, John owns a Toyota Camry, and we can access the car's properties through the Car property of the Person object:

Console.WriteLine(john.Car.Make); // Output: Toyota

Console.WriteLine(john.Car.Model); // Output: Camry

I hope that helps you understand associations in C#!

**Aggregation (Has-a relationship)**

The first relationship that you will see is Aggregation which is also known as **Has-a relationship**. This kind of relationship exists between two classes when a reference variable of one class is a member variable in another class.

In C#, we can also create what we call an aggregation between objects. Aggregation is similar to association, but it has a different meaning.

Aggregation is when we create a "has-a" relationship between two objects, where one object "has" another object as a part of it. For example, let's say we have a class called "School" and another class called "Classroom". A School can have multiple classrooms, and each classroom belongs to one school.

Here's an example of how we can create this aggregation:

class School {

public string Name { get; set; }

public List<Classroom> Classrooms { get; set; }

}

class Classroom {

public string RoomNumber { get; set; }

public int Capacity { get; set; }

}

In the School class, we have a List property called Classrooms, which will hold a list of Classroom objects. This represents the aggregation relationship between a School and its classrooms.

We can create an instance of the School class and add classrooms to it like this:

School mySchool = new School();

mySchool.Name = "ABC School";

Classroom classroom1 = new Classroom();

classroom1.RoomNumber = "101";

classroom1.Capacity = 30;

Classroom classroom2 = new Classroom();

classroom2.RoomNumber = "102";

classroom2.Capacity = 35;

mySchool.Classrooms = new List<Classroom>() { classroom1, classroom2 };

Now, our School object has two classrooms, and we can access them through the Classrooms property:

Console.WriteLine(mySchool.Classrooms[0].RoomNumber); // Output: 101

Console.WriteLine(mySchool.Classrooms[1].Capacity); // Output: 35

I hope that helps you understand aggregation in C#!

**Composition:(Has-a relationship)**

The Composition is a way to design or implement the "has-a" relationship. Composition and Inheritance both are design techniques. The Inheritance is used to implement the "is-a" relationship. The "has-a" relationship is used to ensure the code reusability in our program. In Composition, we use an instance variable that refers to another object.

In C#, we can create what we call a composition relationship between objects. Composition is another way of creating a "has-a" relationship between two objects, but it's a bit different from aggregation.

Composition is when one object "owns" another object, and the owned object cannot exist without the owner object. For example, let's say we have a class called "Car" and another class called "Engine". A Car "has" an Engine as a part of it, and the Engine cannot exist without the Car.

Here's an example of how we can create this composition:

class Car {

public string Make { get; set; }

public string Model { get; set; }

public Engine Engine { get; set; }

public Car() {

Engine = new Engine();

}

}

class Engine {

public int Horsepower { get; set; }

}

In the Car class, we have a property called Engine, which holds an Engine object. But notice that in the Car constructor, we also create a new Engine object and assign it to the Engine property. This is the key difference between composition and aggregation - in composition, the owned object (Engine) is created inside the owner object (Car), and the owner object takes responsibility for the lifetime of the owned object.

We can create an instance of the Car class like this:

Car myCar = new Car();

myCar.Make = "Toyota";

myCar.Model = "Camry";

myCar.Engine.Horsepower = 200;

Now, our Car object has an Engine object inside it, and we can access the Engine's properties through the Engine property of the Car object:

Console.WriteLine(myCar.Engine.Horsepower); // Output: 200

I hope that helps you understand composition in C#!

**Inheritance (Is-a Relationship)**

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in C# by which **one class is allowed to inherit the features(fields and methods) of another class.**

The symbol used for inheritance is ':'

class Customer {

//Parent/Super/Base class

}

class RegularCustomer : Customer { // RegularCustomer is a Customer

//Child/Sub/Derived class

}

class Guest : Customer { // Guest is a Customer

//Child/Sub/Derived class

}

**Constructor chaining**

In the parameterized constructor of base class Customer, observe the 'this' keyword. 'this' is used to call the parameter less constructor of the same class.

This approach where a constructor calls another constructor in the same or base class is called Constructor chaining.

public class Person {

private string name;

private int age;

public Person() : this("Unknown") {

}

public Person(string name) : this(name, 0) {

}

public Person(string name, int age) {

this.name = name;

this.age = age;

}

public string GetName() {

return name;

}

public int GetAge() {

return age;

}

}

// Usage

Person person1 = new Person();

Console.WriteLine("Name: " + person1.GetName() + ", Age: " + person1.GetAge()); // Output: Name: Unknown, Age: 0

Person person2 = new Person("John");

Console.WriteLine("Name: " + person2.GetName() + ", Age: " + person2.GetAge()); // Output: Name: John, Age: 0

Person person3 = new Person("Jane", 25);

Console.WriteLine("Name: " + person3.GetName() + ", Age: " + person3.GetAge()); // Output: Name: Jane, Age: 25

------------------------------------------------------------------------------------------------------------------------------------

10) Dynamic Polymorphism

**Dynamic polymorphism**

Polymorphism that gets **Executed during run time** is known as **dynamic polymorphism or run time polymorphism.**

This type of polymorphism is achieved using overriding the parent method in the child class, called as Method Overriding.

**Method Overriding:**

-Method Overriding allows the programmer to have **multiple methods with the same name signature but different implementation.**

--When we override a method in the child class, it should have the same signature as that of the parent class.

--The method should not have a weaker access modifier.

--Private methods are not overridden.

In C# we can use 3 types of keywords for Method Overriding:

**1) virtual keyword:** This modifier or keyword use **within base class method**. It is used to modify a method in base class for overridden that particular method in the derived class.

**2) override:** This modifier or keyword **use with derived class method**. It is used to modify a virtual or abstract method into derived class which presents in base class.

**3)base Keyword:** This is used to **access members of the base class from derived class**. It basically used to access **constructors and methods or functions of the base class**. The base keyword cannot use within a static method. Base keyword specifies which constructor of the base class should be invoked while creating the instances of the derived class.

**Comparison of Method Overloading and Method Overriding:**

-Method overloading in which class can have multiple methods with the **same name but different signature**.

-Method overriding in which subclass or child class can have a method with the **same name and signature** as that of the parent class.

public class Animal {

public virtual void MakeSound() {

Console.WriteLine("The animal makes a sound");

}

}

public class Cat : Animal {

public override void MakeSound() {

Console.WriteLine("Meow!");

}

}

public class Dog : Animal {

public override void MakeSound() {

Console.WriteLine("Woof!");

}

}

// Usage

Animal animal = new Animal();

Cat cat = new Cat();

Dog dog = new Dog();

animal.MakeSound(); // Output: The animal makes a sound

cat.MakeSound(); // Output: Meow!

dog.MakeSound(); // Output: Woof!

-------------------------------------------------------------------------------------------------------------------------------------

11) Abstract

Abstraction in C# is the process to hide the internal details and show only the functionality.

In simple terms, an abstract class is a blueprint for other classes to inherit from, providing common functionality, while an interface is a contract that a class can fulfill by implementing the required members. Abstract classes focus on defining a common structure for related classes, whereas interfaces allow unrelated classes to share a common set of functionality.

**1) Abstract Class:**

public abstract class Customer {}

---An abstract class is an incomplete class or special class we can't be instantiated. The purpose of an abstract class is to provide a blueprint for derived classes and set some rules what the derived classes must implement when they inherit an abstract class.

---We can use an abstract class as a base class and all derived classes must implement abstract definitions. An abstract method must be implemented in all non-abstract classes using the override keyword. After overriding the abstract method is in the non-Abstract class. We can derive this class in another class and again we can override the same abstract method with it.

---C# Abstract Class Features:

An abstract class can inherit from a class and one or more interfaces.

An abstract class can implement code with non-Abstract methods.

An Abstract class can have modifiers for methods, properties etc.

An Abstract class can have constants and fields.

An abstract class can implement a property.

An abstract class can have constructors or destructors.

An abstract class cannot be inherited from by structures.

An abstract class cannot support multiple inheritance.

Parent Reference

|

Customer regularCustomer=new RegularCustomer();

|

Child Object

We know a parent class reference can refer to a child class object. So, if a reference belongs to an abstract class, you can be sure that the object it refers to will always be of its child type.

**2) Abstract Method:**

public abstract double payBill(double totalPrice);

--Sometimes, we require just method declaration in super-classes.This can be achieve by specifying the abstract type modifier. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the super-class. Thus, a subclass must override them to provide method definition. To declare an abstract method, use this general form:

AccessSpecifier abstract returntype method-name(parameter-list);

--As you can see, no method body is present. Any concrete class(i.e. class without abstract keyword) that extends an abstract class must overrides all the abstract methods of the class.

Important rules for abstract methods:

--Any class that contains one or more abstract methods must also be declared abstract

The following are various illegal combinations of other modifiers for methods with respect to abstract modifier :

-final

-abstract native

-abstract synchronized

-abstract static

-abstract private

-abstract strict

**3)abstract and final:**

"Final" is used to prevent a class or method from being inherited or overridden by derived classes. This means that the class or method cannot be modified or extended by any derived class. A final class or method is marked with the "sealed" keyword.

public sealed class Dog : Animal {

public override void MakeSound() {

Console.WriteLine("Woof!");

}

}

----------------------------------------------------------------------------------------------------------------------------------------------------

12) Interface

* An interface defines a contract. Any class or struct that implements that contract must provide an implementation of the members defined in the interface.
* Another way to achieve abstraction in C#, is with interfaces.
* An interface is a completely "abstract class", which can only contain abstract methods and properties (with empty bodies)

e.g.

// interface

interface Animal

{

void animalSound(); // interface method (does not have a body)

void run(); // interface method (does not have a body)

}

-----------------------------------------------------------------------------------------------

--It is considered good practice to start with the letter "I" at the beginning of an interface, as it makes it easier for yourself and others to remember that it is an interface and not a class.

--By default, members of an interface are abstract and public.

--Note: Interfaces can contain properties and methods, but not fields.

-----------------------------------------------------------------------------------------------

--To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class. To implement an interface, use the : symbol (just like with inheritance). The body of the interface method is provided by the "implement" class. Note that you do not have to use the override keyword when implementing an interface:

Example

// Interface

interface IAnimal

{

void animalSound(); // interface method (does not have a body)

}

// Pig "implements" the IAnimal interface

class Pig : IAnimal

{

public void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

}

}

-----------------------------------------------------------------------------------------------

**Notes on Interfaces:**

--Like abstract classes, interfaces cannot be used to create objects (in the example above, it is not possible to create an "IAnimal" object in the Program class)

--Interface methods do not have a body - the body is provided by the "implement" class

--On implementation of an interface, you must override all of its methods

--Interfaces can contain properties and methods, but not fields/variables

--Interface members are by default abstract and public

--An interface cannot contain a constructor (as it cannot be used to create objects)

**Why And When To Use Interfaces?**

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can implement multiple interfaces. Note: To implement multiple interfaces, separate them with a comma (see example below).

-----------------------------------------------------------------------------------------------

**Multiple Interfaces:**

To implement multiple interfaces, separate them with a comma:

Example

interface IFirstInterface

{

void myMethod(); // interface method

}

interface ISecondInterface

{

void myOtherMethod(); // interface method

}

// Implement multiple interfaces

class DemoClass : IFirstInterface, ISecondInterface

{

public void myMethod()

{

Console.WriteLine("Some text..");

}

public void myOtherMethod()

{

Console.WriteLine("Some other text...");

}

}

class Program

{

static void Main(string[] args)

{

DemoClass myObj = new DemoClass();

myObj.myMethod();

myObj.myOtherMethod();

}

}

----------------------------------------------------------------------------------------------------------------------------------------------------

13) Exception Handling

The **runtime errors are called as exceptions** and the **process of handling the exceptions** is known as **Exception Handling.**

-The try block contains a set of statements where an exception can occur.

-A catch block is where you handle the exceptions, i.e., the catch block specifies what is to be done when an exception occurs.

-- A finally block always executes whether the try block terminates normally or terminates due to an exception.

----------------------------------------------------------------------------

try

{

// statements that may cause an exception

}

catch (exceptionType e) ‏

{

// error handling code

}

----------------------------------------------------------------------------

//main try block

try{

statement 1;

statement 2;

//nested try-catch block

try{

statement 3;

}

catch(Exception ex){

//exception message

}

}

catch(Exception ex){

//exception message

}

----------------------------------------------------------------------------

try{

//Statements that may cause an exception

}

catch (ExceptionType e) ‏{

//Error handling code

}

finally {

//Statements that must be executed

}

**Types of Exceptions:**

There are two different types of exceptions in c#:

**1)Checked Exception:**

E.g. - SQLException, IOException, etc. are Checked Exceptions

**2)Unchecked Exception:**

E.g. - ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. are Unchecked Exceptions

**throw:**

In C#, "throw" is a keyword used to explicitly throw an exception. When an exception is thrown, it interrupts the normal flow of the program and transfers control to the nearest catch block that can handle the exception.

throw new <exceptionclass>("error message");

**2) User-defined Exception:**

class NegativeMarksException : Exception

{

public NegativeMarksException(String message)

{

super(message);// The message is being passed to the parent class

// constructor as the detailed message for the exception

}

}

--An Exception is an unwanted event that interrupts the normal flow of the program.

--In c#, exception is basically of two types – Checked and Unchecked Exception.

--Exception can be handled with try, catch and finally block.

--A single try block can have any number of catch blocks.

--A single try block must be followed by at least one catch block or finally block.

--A generic catch block can handle all the exceptions.

--If no exception occurs in try block, then the catch blocks are completely ignored.

--Generic catch block must be the last catch block of try-catch.

--Finally block is optional.

--A finally block gets executed irrespective of whether an exception occurs or not.

--A finally block must be associated with a try block, you cannot use finally without a try block.

--An exception in the finally block behaves exactly like any other exception.

--Exception can be raised explicitly also by using throw keyword.

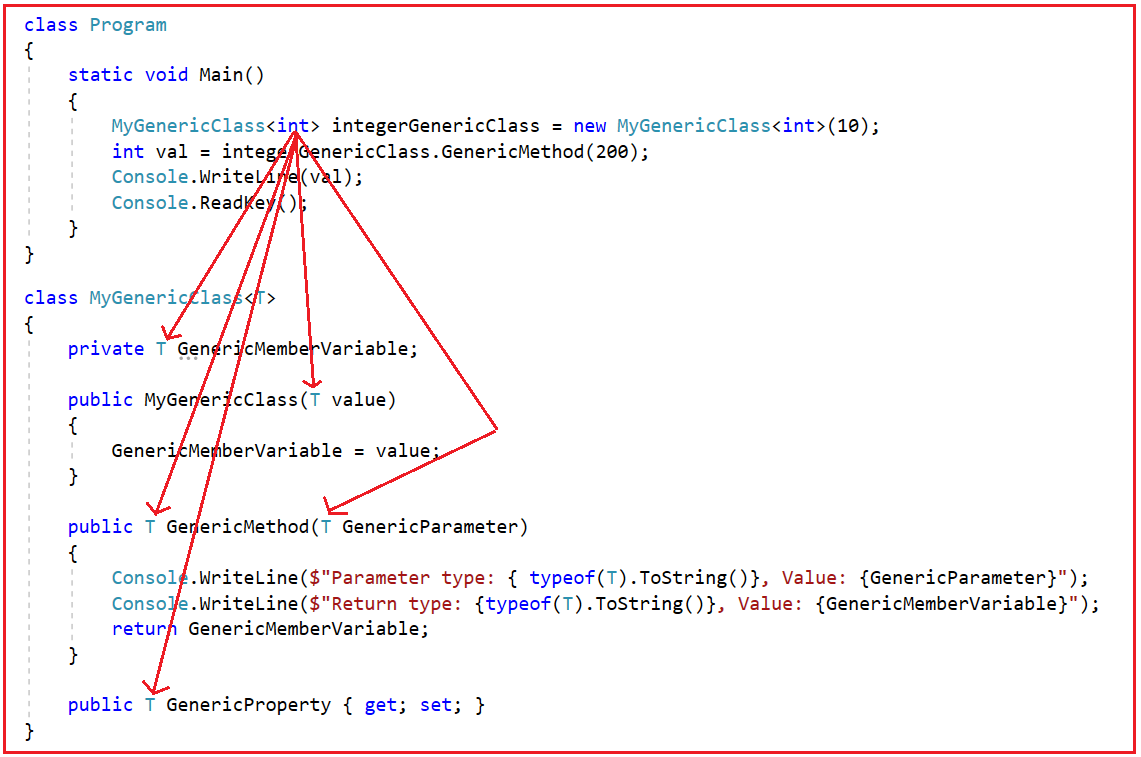
--If you are propagating the exception to the method call using throw, then method must be declared with throws keyword.

--Programmer can define their own exception, i.e., create User-defined exception.

--User-defined exception must extend Exception class.

----------------------------------------------------------------------------------------------------------------------------------------------------

14) Generics



* C# includes specialized classes that store series of values or objects are called collections.
* There are two types of collections available in C#:

**1) generic collections**

**2) non-generic collections**

* **System.Collections.Generic** namespace includes **generic collection** types
* The **System.Collections** namespace contains the **non-generic collection** types

**Collections**

**1) Generic Collections:**

--C# includes the following generic collection classes in the **System.Collections.Generic** namespace.

**Generic Class**

public class MyGenericClass<T>

{

private T myField;

public MyGenericClass(T initialValue)

{

myField = initialValue;

}

public void MyGenericMethod(T argument)

{

Console.WriteLine("Argument: {0}, Field: {1}", argument, myField);

}

}

**Here's an example of how you can create an instance of the class and call the generic method:**

MyGenericClass<string> myGenericObject = new MyGenericClass<string>("initial value"); myGenericObject.MyGenericMethod("argument value");

**The output of the program would be:**

Argument: argument value, Field: initial value

**1) List<T>**

The **List<T>** class is a generic collection that represents a list of elements. Here's an example of how to use a **List<string>** to hold a list of names:

List<string> names = new List<string>();

names.Add("Alice");

names.Add("Bob");

names.Add("Charlie");

foreach (string name in names)

{

Console.WriteLine(name);

}

In this example, we create a **List<string>** called **names**, and add three names to it using the **Add** method. We then use a **foreach** loop to iterate through the **names** list and write each name to the console.

**2) Dictionary<TKey,TValue>**

The **Dictionary<TKey, TValue>** class is a generic collection that represents a collection of key-value pairs. Here's an example of how to use a **Dictionary<string, int>** to hold a collection of words and their lengths:

Dictionary<string, int> wordLengths = new Dictionary<string, int>();

wordLengths.Add("hello", 5);

wordLengths.Add("world", 5);

wordLengths.Add("foo", 3);

wordLengths.Add("bar", 3);

foreach (KeyValuePair<string, int> pair in wordLengths)

{

Console.WriteLine("{0} has length {1}", pair.Key, pair.Value);

}

In this example, we create a **Dictionary<string, int>** called **wordLengths**, and add four key-value pairs to it using the **Add** method. We then use a **foreach** loop to iterate through the **wordLengths** dictionary and write each key-value pair to the console.

**3) SortedList<TKey,TValue>**

The **SortedList<TKey, TValue>** class is a generic collection that represents a sorted list of key-value pairs. The list is **sorted by the keys**, and you can add and remove key-value pairs, access them by key, and iterate through the list in key order. Here's an example of how to use a **SortedList<string, int>** to hold a sorted list of words and their lengths:

SortedList<string, int> wordLengths = new SortedList<string, int>();

wordLengths.Add("hello", 5);

wordLengths.Add("world", 5);

wordLengths.Add("foo", 3);

wordLengths.Add("bar", 3);

foreach (KeyValuePair<string, int> pair in wordLengths)

{

Console.WriteLine("{0} has length {1}", pair.Key, pair.Value);

}

In this example, we create a **SortedList<string, int>** called **wordLengths**, and add four key-value pairs to it using the **Add** method. The list is automatically sorted by the keys, so the output of the **foreach** loop will be:

bar has length 3

foo has length 3

hello has length 5

world has length 5

You can also access individual values by key using the **this** indexer, like this:

int lengthOfHello = wordLengths["hello"];

In this example, we access the value associated with the key "hello", which is 5.

In conclusion, the **SortedList<TKey, TValue>** class is a useful tool for working with sorted data in C#. It can help you keep your data organized and easily accessible, and can save you time and effort when dealing with large amounts of data.

**4) Queue<T>**

The **Queue<T>** class is a generic collection that represents a queue of elements. Here's an example of how to use a **Queue<int>** to hold a queue of numbers:

Queue<int> numbers = new Queue<int>();

numbers.Enqueue(1);

numbers.Enqueue(2);

numbers.Enqueue(3);

while (numbers.Count > 0)

{

int number = numbers.Dequeue();

Console.WriteLine(number);

}

In this example, we create a **Queue<int>** called **numbers**, and add three numbers to it using the **Enqueue** method. We then use a **while** loop to dequeue each number from the **numbers** queue and write it to the console.

**5) Stack<T>**

The **Stack<T>** class is a generic collection that represents a last-in, first-out (LIFO) stack of elements. You can push elements onto the top of the stack, pop elements off the top of the stack, and peek at the element at the top of the stack without removing it. Here's an example of how to use a **Stack<string>** to hold a stack of strings:

Stack<string> stack = new Stack<string>();

stack.Push("Alice");

stack.Push("Bob");

stack.Push("Charlie");

while (stack.Count > 0)

{

string name = stack.Pop();

Console.WriteLine(name);

}

In this example, we create a **Stack<string>** called **stack**, and push three names onto it using the **Push** method. We then use a **while** loop to pop each name off the top of the stack and write it to the console.

**6) Hashset<T>**

The **HashSet<T>** class is a generic collection that represents a set of unique elements. You can add elements to the set, remove elements from the set, and check if an element is in the set. Here's an example of how to use a **HashSet<int>** to hold a set of integers:

HashSet<int> set = new HashSet<int>();

set.Add(1);

set.Add(2);

set.Add(3);

if (set.Contains(2))

{

set.Remove(2);

}

foreach (int number in set)

{

Console.WriteLine(number);

}

In this example, we create a **HashSet<int>** called **set**, and add three numbers to it using the **Add** method. We then check if the set contains the number 2 using the **Contains** method, and remove it from the set using the **Remove** method. We then use a **foreach** loop to iterate through the remaining elements in the set and write them to the console.

**2) Non-generic Collections:**

----C# includes the following non-generic collection classes in the **System.Collections** namespace.

**1) ArrayList**

It is a collection that can hold objects of any type. It is dynamically resizable and can add, remove, and access items by index.

ArrayList list = new ArrayList();

list.Add("apple");

list.Add(123);

list.Add(DateTime.Now);

**2) SortedList**

SortedList is a collection in C# that maintains a sorted order of key-value pairs. It is implemented using an array and binary search algorithm. The keys in the SortedList must be unique, and the collection automatically maintains the keys in sorted order. SortedList provides fast access to items using the key and is useful in scenarios where you need to maintain a sorted collection of items.

SortedList sortedList = new SortedList();

sortedList.Add("apple", 1);

sortedList.Add("banana", 2);

sortedList.Add("orange", 3);

foreach (var key in sortedList.Keys)

{

Console.WriteLine("{0}: {1}", key, sortedList[key]);

}

In the above example, we create a SortedList of string keys and integer values. We add three items to the SortedList, and then use a foreach loop to print out the keys and values in sorted order.

**3) Stack**

It is a collection that stores items in a last-in, first-out (LIFO) order. It is useful for tasks such as undo/redo operations.

Stack stack = new Stack();

stack.Push("one");

stack.Push("two");

stack.Push("three");

**4) Queue**

It is a collection that stores items in a first-in, first-out (FIFO) order. It is useful for tasks such as processing items in the order they were received.

Queue queue = new Queue();

queue.Enqueue("one");

queue.Enqueue("two");

queue.Enqueue("three");

**5) Hashtable**

It is a collection that stores key/value pairs. The keys are unique and are used to access the corresponding values.

Hashtable ht = new Hashtable();

ht.Add("key1", "value1");

ht.Add(2, "value2");

ht.Add("today", DateTime.Now);

**6) BitArray**

BitArray is a collection in C# that stores a sequence of Boolean values as a compact array of bits. It is useful when you need to store a large number of Boolean values efficiently. BitArray provides bitwise operations such as AND, OR, XOR, and NOT, making it easy to manipulate the Boolean values.

BitArray bitArray = new BitArray(8);

bitArray[0] = true;

bitArray[1] = false;

bitArray[2] = true;

Console.WriteLine("bitArray[0]: {0}", bitArray[0]);

Console.WriteLine("bitArray[1]: {0}", bitArray[1]);

Console.WriteLine("bitArray[2]: {0}", bitArray[2]);

In the above example, we create a BitArray of size 8 and set three Boolean values in the array. We then print out the values of the first three elements in the BitArray.

----------------------------------------------------------------------------------------------------------------------------------------------------

16) CSharp language enhancements

**Implicitly typed variables and arrays**

**Part I: Implicitly typed variables**

**var** emailId = "albert@gmail.com";

**Part II: Implicitly typed array**

We're using the **var** keyword to declare the array without specifying its type explicitly. The compiler will infer the type of the array based on the type of its elements

var contacts = new[] { “” };

contacts[0] = “albert@gmail.com”;

contacts[1] = “aria@gmail.com”;

contacts[2] = “andrew@gmail.com”;

or

var myArray = new [] { 1, 2, 3 };

**Dynamic Variables**

You can initialize or assign dynamic variables with any type of values.

Ex. dynamic data = 100;

**Object and Collection Initializers**

**Part I: Object initializer**

In C#, an object initializer is a syntactic shortcut that allows you to create and initialize an object in a single statement. Instead of creating the object first and then setting its properties or fields one by one, you can use an object initializer to create the object and set its properties or fields at the same time.

#region From C# 3.0 and above

Product productOne = new Product

{

ProductId = "P1001",

ProductName = "Tennis Ball",

Price = 500,

QuantityAvailable = 25,

CategoryId = 1

};

#endregion

**Part II: Anonymous types**

Anonymous types in C# help you create a description of an object without needing to give it a name. This is helpful when you want to describe an object that you don't need to keep track of or reuse in your code.

Here's an example. Let's say we want to describe a book we have seen, but we don't have the actual book to show to our friends. We can create an anonymous type to describe it like this:

Ex.

var book = new

{

Title = "Harry Potter",

Author = "J.K. Rowling",

Pages = 400

};

----------------------------------------------------------------------------------------------------------------------------------------------------

17) Delegates

* Delegates are used to **pass a function as a parameter**. C# handles the callback functions or event handler using delegates.
* The delegate is a **reference type data type** that defines the **method signature**.
* You can define variables of delegate, just like other data type, that can refer to any method with the same signature as the delegate.

--There are three steps involved while working with delegates:

1) Declare a delegate

2) Set a target method

3) Invoke a delegate

-------------------------------------------------------------------------------------

**1) Declare a delegate:**

A delegate can be declared using the delegate keyword followed by a function signature, as shown below.

**Syntax:**

**[access modifier] delegate [return type] [delegate name]([parameters]);**

public **delegate** void MyDelegate(string msg);

-------------------------------------------------------------------------------------

**2) Set a target method:**

Example: Set Delegate Target

**public delegate void MyDelegate(string msg);** // declare a delegate

// set target method

**MyDelegate del = new MyDelegate(MethodA);**

// or

MyDelegate del = MethodA;

// or set lambda expression

MyDelegate del = (string msg) => Console.WriteLine(msg);

// target method

**static void MethodA(string message)**

**{**

**Console.WriteLine(message);**

**}**

-------------------------------------------------------------------------------------

**3) Invoke a delegate:**

del.Invoke("Hello World!");

// or

**del("Hello World!");**

-------------------------------------------------------------------------------------

The following is a full example of a delegate.

**Example: Delegate**

public delegate void MyDelegate(string msg); //declaring a delegate

class Program

{

static void Main(string[] args)

{

MyDelegate del = ClassA.MethodA;

del("Hello World");

del = ClassB.MethodB;

del("Hello World");

del = (string msg) => Console.WriteLine("Called lambda expression: " + msg);

del("Hello World");

}

}

class ClassA

{

static void MethodA(string message)

{

Console.WriteLine("Called ClassA.MethodA() with parameter: " + message);

}

}

class ClassB

{

static void MethodB(string message)

{

Console.WriteLine("Called ClassB.MethodB() with parameter: " + message);

}

}

----------------------------------------------------------------------------------------------------------------------------------------------------

18) Delegates Enhancements

**Lambda expressions**

Lambda expressions in C# are used like anonymous functions, with the difference that in Lambda expressions you don’t need to specify the type of the value that you input thus making it more flexible to use.

The ‘=>’ is the lambda operator which is used in all lambda expressions. The Lambda expression is divided into two parts, the left side is the input and the right is the expression.

The Lambda Expressions can be of two types:

**Expression Lambda:** Consists of the input and the expression.

Syntax:

input => expression;

**Statement Lambda:** Consists of the input and a set of statements to be executed.

Syntax:

input => { statements };

There are 4 types of delegates:

1) Func Delegate

2) Action Delegate

3) Predicate Delegate

4) Anonymous Method

**Built-in Delegates**

Func, Action and Predicate are define in C# 3.0 and these are generic inbuilt delegates. The above steps are not required, if you use these delegates.

**1) Func Delegate**

The **Func** delegate is used for methods that return a value. It takes up to 16 parameters, with the last parameter being the return type.

Great question! The reason we have three **int** types in **Func<int, int, int>** is because **Func** is a generic delegate type that takes type parameters to specify its signature.

The first two **int** types specify the types of the two input parameters that the **Func** delegate expects. In this case, both input parameters are of type **int**.

The last **int** type specifies the return type of the delegate. In this case, the **Func** delegate returns an **int**.

// Declare a Func delegate that takes two int parameters and returns an int

Func<int, int, int> myFunc = (x, y) => x + y;

// Call the delegate

int result = myFunc(10, 20); // result is 30

**2) Action Delegate**

The **Action** delegate is used for methods that return nothing, or **void**. It takes up to 16 parameters.

// Declare an Action delegate that takes two int parameters

Action<int, int> myAction = (x, y) => Console.WriteLine(x + y);

// Call the delegate

myAction(10, 20); // prints 30

**3) Predicate Delegate**

The **Predicate** delegate is used for methods that return a Boolean value. It takes one parameter.

// Declare a Predicate delegate that takes an int parameter and returns a Boolean value

Predicate<int> myPredicate = x => x > 5;

// Call the delegate

bool result = myPredicate(10); // result is true

**Anonymous Method Delegate**

An anonymous method is a type of delegate in C# that allows you to define a method inline without having to explicitly declare a separate method. It's a way to create a delegate on the fly, without having to create a separate method to handle the delegate.

Anonymous methods are declared using the **delegate** keyword followed by the input parameters and body of the method. Here's an example:

// Declare an anonymous method that takes two int parameters and returns their sum

var myDelegate = delegate (int x, int y) { return x + y; };

// Call the delegate

int result = myDelegate(10, 20); // result is 30

----------------------------------------------------------------------------------------------------------------------------------------------------

19) Extension Methods

Extension methods, as the name suggests, are **additional methods**. Extension methods allow you to inject additional methods without modifying, deriving or recompiling the original class, struct or interface. Extension methods can be added to your own custom class, .NET framework classes, or third party classes or interfaces.

Let's say you have a simple **int** variable in C# and you want to add a method to it that can double its value. You could create a static utility class with a method to do this, like this:

public static class MathUtils

{

public static int Double(int number)

{

return number \* 2;

}

}

Then you could call this method like this:

int myNumber = 5;

int doubledNumber = MathUtils.Double(myNumber); // doubledNumber is now 10

However, this approach requires you to call the **Double** method on the **MathUtils** class, which may not be the most readable or intuitive way to write your code. Instead, you can use an extension method to add a **Double** method directly to the **int** type. Here's how you can do this:

public static class IntExtensions

{

public static int Double(this int number)

{

return number \* 2;

}

}

Notice the **this** keyword before the **int** parameter in the method signature. This tells C# that this method is an extension method for the **int** type.

Now you can call this method directly on an **int** variable:

int myNumber = 5;

int doubledNumber = myNumber.Double(); // doubledNumber is now 10

This syntax is more intuitive and makes it clearer that you're doubling an **int** variable.

Extension methods can be very useful for adding functionality to types that you don't control or for creating utility methods that make your code more readable and maintainable. However, it's important to use them judiciously and not abuse them to clutter your codebase with unnecessary extension methods.

----------------------------------------------------------------------------------------------------------------------------------------------------

20) LINQ - Language Integrated Query

LINQ in C# is used to work with data access from sources such as objects, data sets, SQL Server, and XML. LINQ stands for **Language Integrated Query.**

There are three ways to implement a query using LINQ:

**1) Query syntax:**

***from <variable> in <collection>***

***< where, joining, grouping, operators, etc.> <lambda expression or condition>***

***<select or groupBy operator> <format the results>***

**e.g.**

**//Data Source**

List<int> integerList = new List<int>()

{

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

};

**//LINQ Query using Query Syntax**

var QuerySyntax = from obj in integerList

where obj> 5

select obj;

**//Execution**

foreach (var item in QuerySyntax)

{

Console.Write(item + " ");

}

-------------------------------------------------------------------------------------------

**2) Method syntax:**

***DataSource.ConditionMethod( ).SelectionMethod( )***

**e.g.**

**//Data Source**

List<int> integerList = new List<int>()

{

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

};

**//LINQ Query using Method Syntax**

var MethodSyntax = integerList.Where(obj => obj> 5).ToList();

**//Execution**

foreach (var item in MethodSyntax)

{

Console.Write(item + " ");

}

**There are no semantic differences (in terms of performance, execution) between Query Syntax and Method Syntax.**

-------------------------------------------------------------------------------------------

**3) Mixed Syntax:**

***( from object in DataSource***

***where condition***

***select object ).Method( )***

**e.g.**

**//Data Source**

List<int> integerList = new List<int>()

{

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

};

**//LINQ Query using Mixed Syntax**

var MixedSyntax = (from obj in integerList

where obj> 5

select obj).Sum();

**//Execution**

Console.Write("Sum Is : " + MixedSyntax);

-------------------------------------------------------------------------------------------

**Execution Types:**

**1) Deferred Execution:**

With deferred execution, only a reference to the query will be stored in the variable.

e.g.

var allProductNames = from product in productList

select product.ProductName;

-------------------------------------------------------------------------------------------

**2) Immediate Execution:**

The second way of query execution is called as the immediate execution. With immediate execution, the query executes and gets the result based on the number of objects that is currently present in the list at any point of runtime. Immediate execution of the query can be done by using methods such as ToArray and ToList.

e.g.

var allProductNamesTwo = (from product in productList

select product.ProductName).ToList();

Instead of the implicitly typed variable, you can also use a generic List with String as its parameter, as you have used the ToList() that returns a List.

List<string> allProductNamesTwo = (from product in productList

select product.ProductName).ToList();

-------------------------------------------------------------------------------------------

Object Collection [LINQ to Objects]

ADO.Net Dataset [LINQ to DataSet]

XML Document [LINQ to XML]

Developer----->Uses-------->To Query----------->Entity Framework [LINQ to Entities]

LINQ SQL Database [LINQ to SQL]

Other data sources [By implementing IQueryable]

-------------------------------------------------------------------------------------------

----------------------->

Objects Returns { LINQ Query } Execute Query Data Source

<--------------- Retrieve Result

<------------------------

----------------------------------------------------------------------------------------------------------------------------------------------------

21) File Handling

**Stream:**

C# includes following standard IO (Input/Output) classes to read/write from different sources like files, memory, network, isolated storage, etc.

**#Stream:** System.IO.Stream is an abstract class that provides standard methods to transfer bytes (read, write, etc.) to the source. It is like a wrapper class to transfer bytes. Classes that need to read/write bytes from a particular source must implement the Stream class.

The following classes inherit Stream class to provide the functionality to Read/Write bytes from a particular source:

**1) FileStream:** reads or writes bytes from/to a physical file, whether it is a .txt, .exe, .jpg, or any other file. FileStream is derived from the Stream class.

**2) MemoryStream:** MemoryStream reads or writes bytes that are stored in memory.

**3) BufferedStream:** BufferedStream reads or writes bytes from other Streams to improve certain I/O operations' performance.

**4) NetworkStream:** NetworkStream reads or writes bytes from a network socket.

**5) PipeStream:** PipeStream reads or writes bytes from different processes.

**6) CryptoStream:** CryptoStream is for linking data streams to cryptographic transformations.

**Stream Readers and Writers:**

**1) StreamReader:** StreamReader is a helper class for reading characters from a Stream by converting bytes into characters using an encoded value. It can be used to read strings (characters) from different Streams like FileStream, MemoryStream, etc.

**2) StreamWriter:** StreamWriter is a helper class for writing a string to a Stream by converting characters into bytes. It can be used to write strings to different Streams such as FileStream, MemoryStream, etc.

**3) BinaryReader:** BinaryReader is a helper class for reading primitive datatype from bytes.

**4) BinaryWriter:** BinaryWriter writes primitive types in binary.

----------------------------------------------------------------------------------------------------------------------------------------------------

22) Asynchronous Programming

--Synchronous represents a **set of activities that starts happening together at the same time.**

--A synchronous call waits for the method to complete before continuing with program flow.

**Asynchronous programming:**

--Asynchronous programming in C# is an efficient approach towards activities blocked or access is delayed. If an activity is blocked like this in a synchronous process, then the complete application waits and it takes more time. The application stops responding. Using the asynchronous approach, the applications continue with other tasks as well.

--The async and await keywords in C# are used in async programming. Using them, you can work with .NET Framework resources, .NET Core, etc. Asynchronous methods defined using the async keyword are called async methods.

The **async** keyword tells C# that a method can run asynchronously, and the **await** keyword tells C# to wait for a task to complete before continuing with the rest of the method.

using System;

using System.Threading.Tasks;

async Task BakeCake()

{

Console.WriteLine("Putting cake in the oven...");

await Task.Delay(5000); // wait for 5 seconds

Console.WriteLine("Cake is done baking!");

}

async Task FrostCake()

{

Console.WriteLine("Starting to frost the cake...");

await Task.Delay(1000); // wait for 1 second

Console.WriteLine("Cake is frosted!");

}

async Task MakeCake()

{

await BakeCake(); // start baking the cake asynchronously

await FrostCake(); // start frosting the cake asynchronously

Console.WriteLine("Cake is ready!"); // this line won't execute until the cake is frosted

}

In this example, we have three methods: **BakeCake()**, **FrostCake()**, and **MakeCake()**. We use the **async** keyword to mark these methods as asynchronous.

In **MakeCake()**, we call **BakeCake()** and **FrostCake()** using the **await** keyword. This means that the program will start baking the cake asynchronously, and then start frosting the cake asynchronously as well.

The **Console.WriteLine("Cake is ready!")** line won't execute until both **BakeCake()** and **FrostCake()** are finished running, which means that the cake is both baked and frosted.

----------------------------------------------------------------------------------------------------------------------------------------------------

23) Memory Management

C# employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by a garbage collector.

----------------------------------------------------------------------------------------------------------------------------------------------------

24) Input In C#

**1) Console.Read();**

Console.Read(); is used to **read only single character** from the standard output device.

Console.WriteLine("Enter some text...");

var str = Console.Read();

Console.WriteLine("You entered '{0}'", str);

**2) Console.ReadLine();**

Console.ReadLine(); is used **to read a line or string from** the standard output device.

Console.WriteLine("Enter some text...");

var str = Console.ReadLine();

Console.WriteLine("You entered '{0}'", str);

**console.readkey();**

ReadKey() Method makes the **program wait for a key press** and it prevents the screen until a key is pressed.

-----------------------------------------------------------------------------------------------------------------------------------------------------

25) Coding Standards

Comments are those statements which are not executed by the compiler. Comments can be used to provide information about a variables or statement.

1) //This is a single line comment

2) /\*This is a

​multi-line comment\*/

-----------------------------------------------------------------------------------------------------------------------------------------------------

26) Recursion

In programs that we have seen the **methods that can call other methods**.

It is also possible for a **method to call itself**. This type of method is known as a **recursive method**. A recursive method can call itself directly or indirectly.

-----------------------------------------------------------------------------------------------------------------------------------------------------

27) Regular Expression

A regular expression (or regex in short) is a **sequence of characters** that forms a **search pattern**.

It is composed of **various symbols and characters** and is mainly used in pattern matching for searching and editing. It helps in providing a pattern against which a string can be matched.

Literal Meta Characters

| |

E.g. - Customer|?

|

Quantifiers

https://lex.infosysapps.com/web/en/viewer/web-module/lex\_auth\_012875664402972672443?collectionId=lex\_auth\_012880464547618816347&collectionType=Course&pathId=lex\_auth\_012876945639448576289

----------------------------------------------------------------------------------------------------------------------------------------------------

28) Partial Classes and Methods

In C#, you can **split the implementation of a class, a struct, a method, or an interface in multiple .cs files using the partial keyword.** The compiler will combine all the implementation from multiple .cs files when the program is compiled.

Consider the following EmployeeProps.cs and EmployeeMethods.cs files that contain the Employee class.

---------------------------------------------------------------------------

**EmployeeProps.cs :**

public partial class Employee

{

public int EmpId { get; set; }

public string Name { get; set; }

}

---------------------------------------------------------------------------

**EmployeeMethods.cs :**

public partial class Employee

{

//constructor

public Employee(int id, string name){

this.EmpId = id;

this.Name = name;

}

public void DisplayEmpInfo() {

Console.WriteLine(this.EmpId + " " this.Name);

}

}

---------------------------------------------------------------------------

**Example: Combined Class :**

public class Employee

{

public int EmpId { get; set; }

public string Name { get; set; }

public Employee(int id, string name){

this.EmpId = id;

this.Name = name;

}

public void DisplayEmpInfo(){

Console.WriteLine(this.EmpId + " " this.Name );

}

}

-----------------------------------------------------------------------------------------------------------------------------------------------------

29) Type Conversion

**1)Implicit Type Conversion:**

Implicit Type Conversion is used when you want to assign a value of **smaller range data type** to a **larger range data type.**

Implicit Type Conversion is also known as **Widening conversion.**

E.g.:

int discountPercentage = 10;

double newDiscountPercentage = discountPercentage;

**2)Explicit Type Conversion:**

Explicit Conversion is used when you want to assign a **value of larger range data type** to a **smaller range data type.**

This conversion is not done by the compiler implicitly as there can be **loss of data** in some cases.

This is also known as **Narrowing conversion.**

E.g.:

double totalPrice = 200;

int newPrice = (int)totalPrice;

-----------------------------------------------------------------------------------------------------------------

**string to int:**

In C#, you can convert a string representation of a number to an integer using the following ways:

1. Parse() method
2. Convert class
3. TryParse() method - Recommended

**1) Parse Method**

The Parse() methods are available for all the **primitive datatypes(Int, Double, float etc.)**. It is the easiest way to convert from **string** to **integer**.

1. Int16.Parse()
2. Int32.Parse() or **Int.Parse()**
3. Int64.Parse()

**2) Convert Class**

The Convert class includes different methods which **convert base data type to another base data type**.

1. Convert.ToInt16() //16-bit integer e.g. short
2. Convert.ToInt32() or **Convert.ToInt()** //32-bit integers e.g. int
3. Convert.ToInt64() //64-bit integer e.g. long

**Convert.ToInt32() allows null value, it doesn't throw any errors and returns 0.**

**Int.Parse() does not allow null value, and it throws an ArgumentNullException error.**

**3) TryParse Method**

The TryParse() methods are available for **all the primitive types to convert string to the calling data type**. It is the recommended way to convert **string** to an **integer**.

1. Int16.TryParse()
2. Int32.TryParse() or **int.TryParse()**
3. Int64.TryParse()

Note:

1. If you've got a string, and you expect it to always be an integer you'd use **Int32.Parse().**
2. If you're collecting input from a user, you'd generally use **Int32.TryParse()**, since it allows you more fine-grained control over the situation when the user enters invalid input.
3. **Convert.ToInt32()** takes an object as its argument. In short, **Convert.ToInt32()** returns **0** if **null** to prevent **int.Parse()** from raising an **ArgumentNullException**

**Other Datatype Conversions**

**1) Convert Class:**

**Convert.ToBoolean()** converts a type to a Boolean value

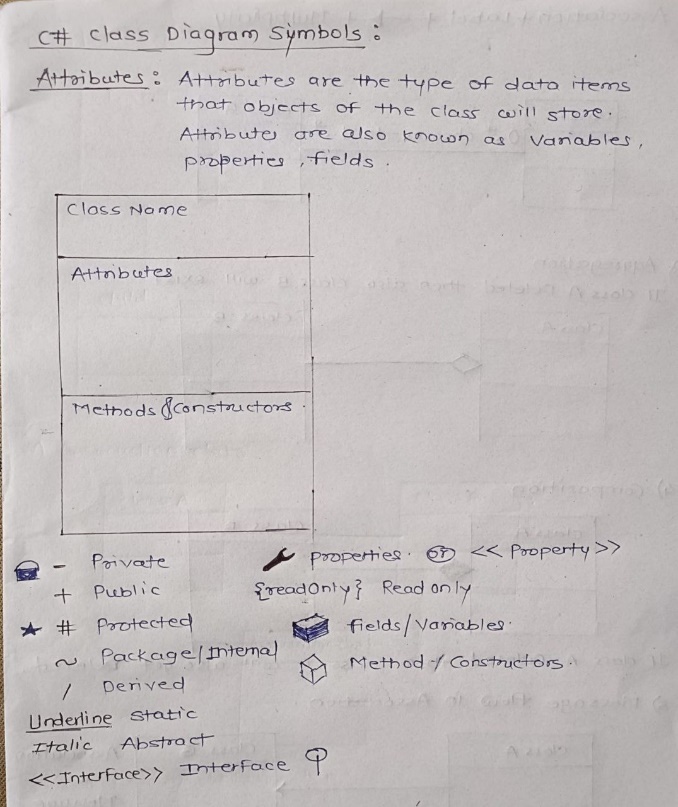
**Convert.ToChar()** converts a type to a char type

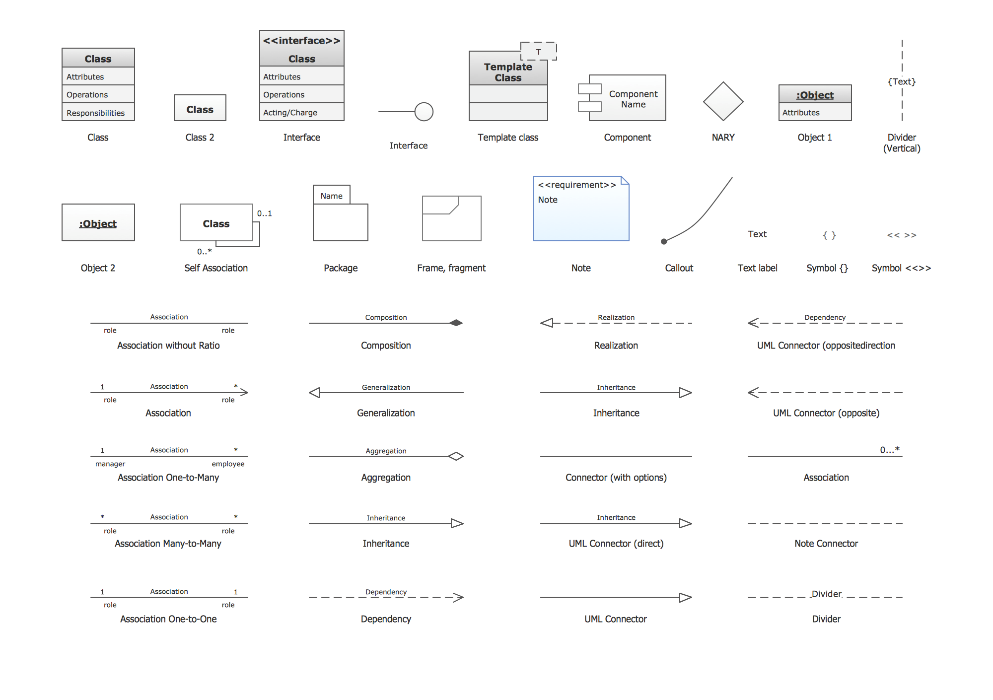
**Convert.ToDouble()** converts a type to a double type

**Convert.ToString()** converts a type to a string

----------------------------------------------------------------------------------------------------------------------------------------------------

30) Class Diagram





----------------------------------------------------------------------------------------------------------------------------------------------------

31) String Vs StringBuilder

**StringBuilder** is used to represent **a mutable string of characters**.

Mutable means the string which can be changed.

So **String** objects are **immutable** but **StringBuilder** is the **mutable** string type.

It will **not create a new modified instance** of the current string object **but do the modifications in the existing string object.**

Here is an example that demonstrates the difference between using a string and a StringBuilder:

string str = "";

for(int i = 0; i < 10000; i++)

{

str += i.ToString();

}

In this example, a new string object is created on each iteration of the loop, which can be slow for large numbers of iterations.

StringBuilder sb = new StringBuilder();

for(int i = 0; i < 10000; i++)

{

sb.Append(i.ToString());

}

string str = sb.ToString();

In this example, a StringBuilder object is used to build up the string one piece at a time without creating a new string object on each iteration. The resulting string is then obtained by calling the ToString() method of the StringBuilder object.

----------------------------------------------------------------------------------------------------------------------------------------------------

32) SOLID Principles

SOLID design principles in C# are basic design principles.

**SOLID stands for:**

**S:** Single Responsibility Principle (SRP)

**O:** Open closed Principle (OCP)

**L:** Liskov substitution Principle (LSP)

**I:** Interface Segregation Principle (ISP)

**D:** Dependency Inversion Principle (DIP)

----------------------------------------------------------------------------------------------------------------------------------------------------

33) Design Patterns

Design patterns provide **general solutions or a flexible way to solve common design** problems.

----------------------------------------------------------------------------------------------------------------------------------------------------

34) Pointer

The pointer in C# language is a **variable which stores the address of another variable**. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

The pointer in c# language can be declared using \* (asterisk symbol).

It is also known as indirection pointer used to dereference a pointer.

e.g

int n = 10;

int\* p=&n;

Variable p of type pointer is pointing to the address of the variable n of type integer.

The unary **address-of** operator ( & ) returns the **address** of (that is, a pointer to) its **operand**.

----------------------------------------------------------------------------------------------------------------------------------------------------

35) IEnumerable & IEnumerator

**IEnumerable:**

* The IEnumerable interface **declares only one method called GetEnumerator** which **returns another type of interface called the IEnumerator interface** for that particular collection.
* This allows **readonly access** to a collection then a collection that implements IEnumerable can be used with a for-each statement.

**Key Points:**

* IEnumerable interface contains the System.Collections.Generic namespace.
* IEnumerable interface is a generic interface which allows looping over generic or non-generic lists.
* IEnumerable interface also works with linq query expression.
* IEnumerable interface Returns an enumerator that iterates through the collection.

e.g.

public class Customer : IEnumerable

{

public IEnumerator GetEnumerator()

{

throw new NotImplementedException();

}

}

In the above example, you have seen that **after implementing the IEnumerable Interface** there is method called **GetEnumerator** along with **interface IEnumerator** which helps to **get current elements from the collection**.

**IEnumerator:**

IEnumerator is the **base interface for all non-generic enumerators** which are used to read the data in the collection.

It has the following two methods:

1. MoveNext()
2. Reset()

**Enumerable** is like a collection of things, like a box of toys. You can do different things with the toys in the box, like play with them, sort them, count them, and so on. In C#, Enumerable is a type of object that lets you work with a collection of data, like a list of numbers or a set of names.

**Enumerator** is like a helper that helps you go through each toy in the box one by one. In C#, Enumerator is a type of object that helps you go through each item in an Enumerable object.

Here's an example to make it clearer. Let's say you have a list of numbers, like {1, 2, 3, 4, 5}. This list is an Enumerable object. To go through each number in the list, you can use an Enumerator object to move from one number to the next. It's like you're pointing at each number in the list one by one, like this:

1, 2, 3, 4, 5

e.g.

using System;

using System.Linq; // We need to include this namespace to use Enumerable

class Program

{

static void Main()

{

string[] basket = {"apple", "banana", "orange", "apple", "grape"};

// Create an Enumerable object from the basket array

var fruits = basket.AsEnumerable();

// Create an Enumerator object to go through each item in the fruits Enumerable

var enumerator = fruits.GetEnumerator();

int appleCount = 0;

while (enumerator.MoveNext())

{

// Check if the current item is an apple

if (enumerator.Current == "apple")

{

appleCount++;

}

}

Console.WriteLine($"There are {appleCount} apples in the basket.");

}

}

Output: There are 2 apples in the basket.

**IEnumerable vs IEnumerator interface:**

* IEnumerable and IEnumerator are both interfaces.
* IEnumerable has just one method called GetEnumerator. This method returns another type which is an interface that interface is IEnumerator.
* If we want to implement enumerator logic in any collection class, it needs to implement IEnumerable interface (either generic or non-generic).
* IEnumerable has just one method whereas IEnumerator has two methods (MoveNext and Reset) and a property Current.

----------------------------------------------------------------------------------------------------------------------------------------------------

36) IQueryable

The IQueryable interface **inherits the IEnumerable** interface so that if it represents a query, the results of that query can be enumerated.

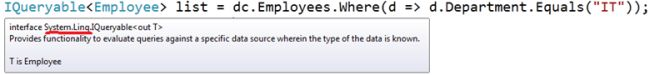
The major difference between IQueryable and IEnumerable is that **IQueryable executes query with filters whereas IEnumerable executes the query first and then it filters the data based on conditions.**

**IEnumerable Vs IQueryable:**





**Vs**





IQueryable is similar to Enumerable, but it's used for more complex data sources, like a database. It allows you to query the data in the source and get only the data that you need, instead of loading everything at once.

For example, let's say you have a database table with thousands of records. If you use Enumerable to load all the records at once, it could take a long time and use up a lot of memory. But with IQueryable, you can write a query to get only the records you need, like all the records where the name starts with "J", and load only those records. This can be faster and more efficient.

e.g.

using System;

using System.Linq; // We need to include this namespace to use IQueryable

class Program

{

static void Main()

{

// Assume we have a database with a "Students" table

// We create an IQueryable object to query the data in the "Students" table

IQueryable<Student> students = dbContext.Students;

// Use LINQ to filter the students who are in the 5th grade

var fifthGraders = students.Where(s => s.Grade == 5);

// Print out the names of the fifth graders

foreach (var student in fifthGraders)

{

Console.WriteLine(student.Name);

}

}

}

// Assume we have a Student class with a Name and Grade property

class Student

{

public string Name { get; set; }

public int Grade { get; set; }

}

----------------------------------------------------------------------------------------------------------------------------------------------------

37) Console.Write() vs Console.WriteLine()

Console.Write() and Console.WriteLine() are both methods in C# that allow you to output text to the console.

Console.Write() method writes the text to the console without adding a newline character at the end. This means that any subsequent text that is written using Console.Write() will be written immediately after the previous text, without any space or line break. Here's an example:

Program:

Console.Write("Hello");

Console.Write("World");

Output:

HelloWorld

Console.WriteLine() method, on the other hand, writes the text to the console and adds a newline character at the end. This means that any subsequent text that is written using Console.WriteLine() will be written on a new line. Here's an example:

Program:

Console.WriteLine("Hello");

Console.WriteLine("World");

Output:

Hello

World

As you can see, the Console.WriteLine() method is useful when you want to output text on multiple lines, whereas the Console.Write() method is useful when you want to output text on the same line without any separation.