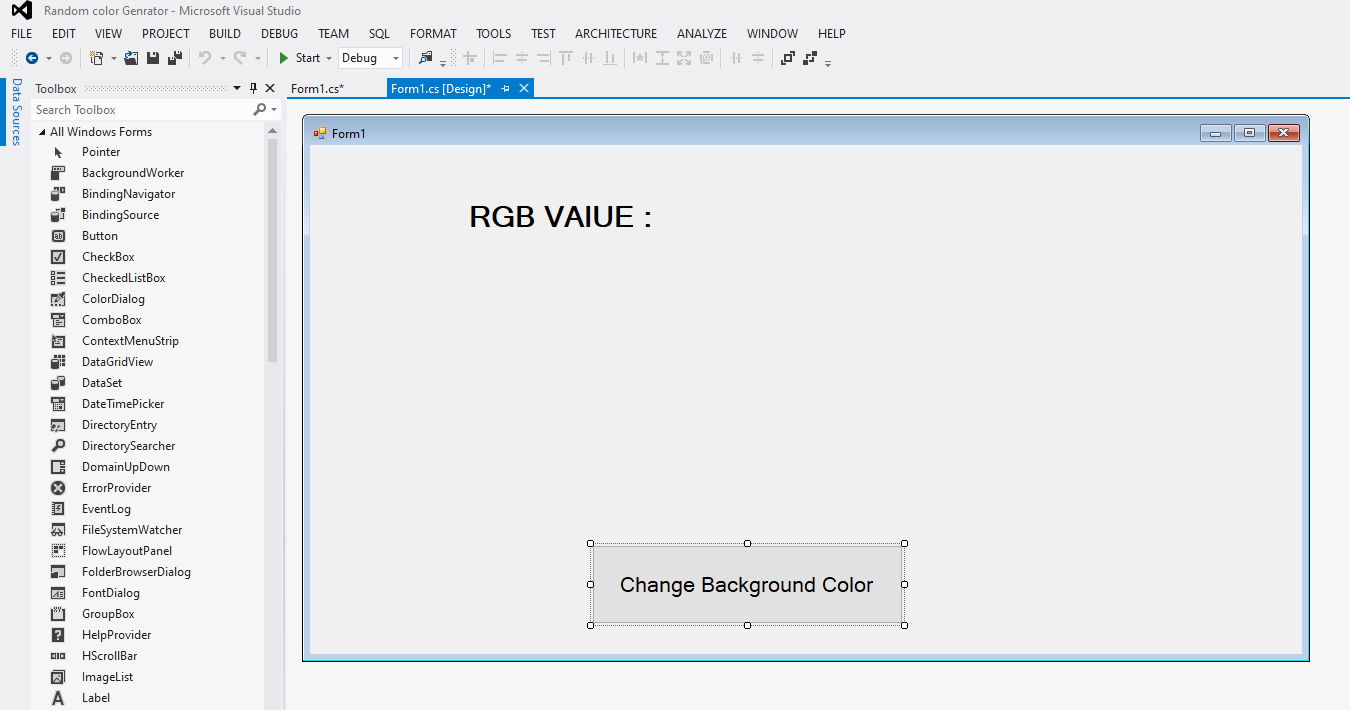
**Made by: Manthar Ali Sandano**

**Visual programming**



Any language that uses the graphics or blocks that are already defined with the code and you just need to use those blocks without worrying about the lines of code is known as a visual programming language.

| **Regular Languages** | **Visual Programming Language** |
| --- | --- |
| It is a programming language that only uses text. | It is a programming language that uses graphics or blocks in place of text. |
| This is not fast and efficient as every block has some code with it so it takes time and also it has graphics with it. | These are quite fast and efficient |
| This requires more memory as it has graphics so to store them more memory is used. | These require less memory as compared to VPL |
| Examples: Mblock, Blockly, Scratch Etc. | Examples: Javascript, C, C++, Java, Python Etc. |

**Why visual programming?**

* Visual elements are easy to understand, that is the development of something in a visual programming language requires less effort.
* There is no need for long-term learning.
* The probability of mistakes is reduced due to the absence of syntax.
* It helps to understand the basic concept of programming in order to be able to switch to more complex programming languages in the future.

**.net framework architecture**

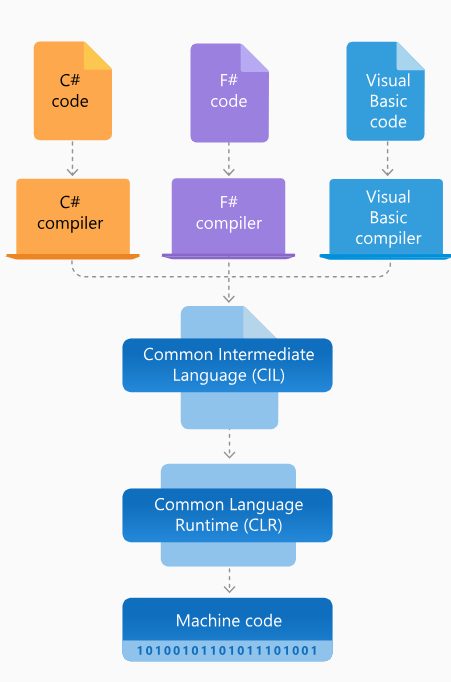
### **What is .NET Framework used for?**

.NET Framework is used to create and run software applications. .NET apps can run on many operating systems, using different implementations of .NET. .NET Framework is used for running .NET apps on Windows.

### **What are the main components/features of .NET Framework?**

The two major components of .NET Framework are the Common Language Runtime (CLR) and the .NET Framework Class Library.

The CLR (common language runtime) manages .NET code execution, loads libraries, and compiles code to machine code.

 The .NET Framework Class Library provides a large set of pre-built functions and classes that can be used to create a wide range of applications.

CIL (Common Intermediate Language): This is a middle step where code is converted into a standardized form that .NET understands.

CLR (Common Language Runtime): This is the system that takes the CIL code and translates it into machine code that your computer’s processor can execute. It also manages running the application, including things like memory and security.

Timer

System. Timer**class** in C# offers a way to run a block of code after a set interval of time – it can be run repeatedly.

For example, backing up a folder every 10 minutes or writing to a log file every second. The method that needs to be executed is placed inside the timer event.

This sort of module has various applications and is ideal for use in web applications to check the status of a connection. Another application might be a timer for a video game.

Do you want to add an embedded clock in your application? Or is there a scenario where a delay in task execution is necessary? In that case, creating a Timer comes to your rescue.

The Interval**property** sets the time interval, in milliseconds, after which the required code is to be run.

namespace Timer\_Object

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void Form1\_Load(object sender, EventArgs e)

{

WindowState = FormWindowState.Maximized;

BackColor = Color.Green;

Text = "timer program";

t1.Interval = 1000;

t1.Start();

}

int count = 1;

private void t1\_Tick(object sender, EventArgs e)

{

count++;

if (count > 10)

{

count = 1;

}

label1.Text = Convert.ToString(count);

}

}

## **C# - What is OOP?**

OOP stands for Object-Oriented Programming.

object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

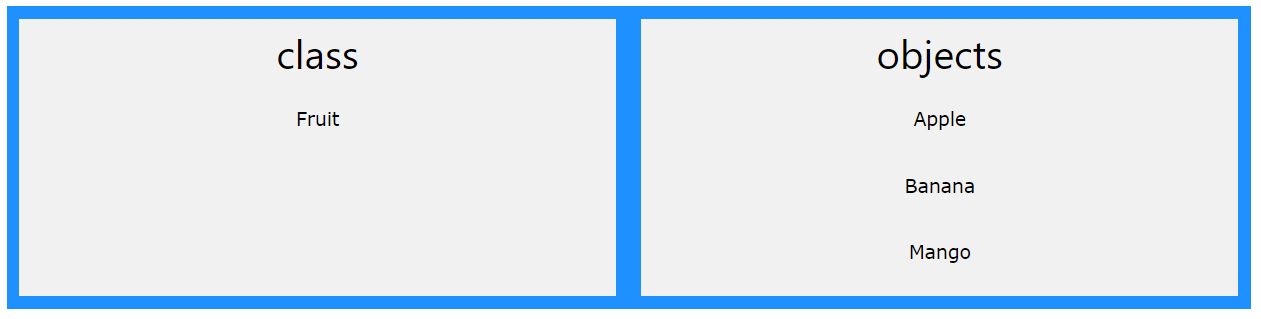
* OOP is faster and easier to execute
* OOP provides a clear structure for the programs
* OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
* OOP makes it possible to create full reusable applications with less code and shorter development time

**Tip:** The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

## **C# - What are Classes and Objects?**

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:



Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

 object is an entity that has state and behavior.

Object is an instance of a class. All the members of the class can be accessed through object.

1. Student s1 = **new** Student();//creating an object of Student

### **C# Class**

In C#, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

1. public **class** Student
2. {
3. **int** id;//field or data member
4. String name;//field or data member
5. }

## **C# Object and Class Example**

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

**using** System;

**public** **class** Student

{

**int** id;//data member (also instance variable)

String name;//data member(also instance variable)

**public** **static** **void** Main(**string**[] args)

{

Student s1 = **new** Student();//creating an object of Student

s1.id = 101;

s1.name = "Sonoo Jaiswal";

Console.WriteLine(s1.id);

Console.WriteLine(s1.name);

}

}

1. *101*
2. *Sonoo Jaiswal*

### **C# Class Example 3: Initialize and Display data through method**

Let's see another example of C# class where we are initializing and displaying object through method.

**using** System;

**public** **class** Student

{

**public** **int** id;

**public** String name;

**public** **void** insert(**int** i, String n)

{

id = i;

name = n;

}

**public** **void** display()

{

Console.WriteLine(id + " " + name);

}

}

**class** TestStudent{

**public** **static** **void** Main(**string**[] args)

{

Student s1 = **new** Student();

Student s2 = **new** Student();

s1.insert(101, "Ajeet");

s2.insert(102, "Tom");

s1.display();

s2.display();

}

}

Output:

1. *101 Ajeet*
2. *102 Tom*

### **C# Class Example 4: Store and Display Employee Information**

**using** System;

**public** **class** Employee

{

**public** **int** id;

**public** String name;

**public** **float** salary;

**public** **void** insert(**int** i, String n,**float** s)

{

id = i;

name = n;

salary = s;

}

**public** **void** display()

{

Console.WriteLine(id + " " + name+" "+salary);

}

}

**class** TestEmployee{

**public** **static** **void** Main(**string**[] args)

{

Employee e1 = **new** Employee();

Employee e2 = **new** Employee();

e1.insert(101, "Sonoo",890000f);

e2.insert(102, "Mahesh", 490000f);

e1.display();

e2.display();

}

}

Output:

*101 Sonoo 890000*

*102 Mahesh 490000*

Background Worker

Three events

DoWork

ProgressChanged

RunWorkerCompleted (completed task)

We often use background threads when a time-consuming process needs to be executed in the background without affecting the responsiveness of the user interface. This is where a BackgroundWorker component comes into play. If you are not familiar with the multi-threaded application, visit our Multithreading Programming using the C# section, where you can find many articles and tutorials

private void backgroundWorker1\_DoWork(object sender, DoWorkEventArgs e)

{

for (int i = 1; i <= 100; i++)

{

Thread.Sleep(50);

backgroundWorker1.ReportProgress(1);

//report prgress will call progress changed

}

}

private void backgroundWorker1\_ProgressChanged(object sender, ProgressChangedEventArgs e)

{

progressBar1.Value += 1;

}

private void backgroundWorker1\_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)

{

MessageBox.Show("Task Completed");

}

private void button1\_Click(object sender, EventArgs e)

{

backgroundWorker1.WorkerReportsProgress = true;

backgroundWorker1.RunWorkerAsync();

//by default it will call do worker

}

Lab

Program changer

int r, g, b;

Random ran = new Random();

r = ran.Next(0, 255);

g = ran.Next(0, 255);

b = ran.Next(0, 255);

BackColor = Color.FromArgb(r,g,b);

colorval.Text = Convert.ToString( r +" , "+ g+ " , " + b);

**Backcolor=Color.color\_name;**

it is used to change background color of form

**label1.ForeColor = Color.White;**

it is used to change text color

**WindowState =FormWindowState.Maximized**

It is used to maximize the form window.

**Label.Text=”string”;**

It is used to set the text or get the text

**Listbox.items.add();**

It is used to add items in the boxlist

**Listbox.items.remove();**

It is used to remove items from the boxlist

**Listbox.items.clear();**

It is used to clear the items from the boxlist

**label1.Font = new Font("Arial", 12, FontStyle.Bold);**

it is used to change the fonts

**label1.TextAlign = ContentAlignment.MiddleCenter;**

it is used to align the text on center

**textBox1.Multiline = true;**

it is used to enable multiple line in textbox

**textBox1.ReadOnly = true;**

it is used to make textbox read-only

**checkBox1.Checked = true;**

it is used to make checked the checkbox

**int selectedIndex = listBox1.SelectedIndex;**

it is used to select the index.

**pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;**

it is used to set size mode for picture

**pictureBox1.Image = Image.FromFile("F:/pics for web/1.jpg");**

it is used to import the image into form

**textBox1.Enabled = true;**

it is used to enable

**textBox1.Enabled = false;**

it is used to disable

**form obj=new form()**

**obj.close();**

it is used to close the form

**obj.hide();**

it is used to hide the form

**Timer program**

WindowState = FormWindowState.Maximized;

BackColor = Color.Green;

Text = "timer program";

t1.Interval = 1000;

t1.Start();

StartPosition = FormStartPosition.CenterParent;

**StartPosition = FormStartPosition.CenterParent**;

It is used to set center position for form at starting;

**colorDialog1.ShowDialog();**

**label1.ForeColor = colorDialog1.Color;**

It is used to show color dialog box and set color to label;

**FontDialog obj = new FontDialog();**

**obj.ShowDialog();**

**label1.Font = obj.Font;**

**fontDialog1.ShowDialog();**

**label1.Font = fontDialog1.Font;**

It is used to show font dialog box and set font to label;

**Using** **System.Threading.**

**Thread.Sleep(100);**

It is used to delay execution.

**MessageBox.Show("");**

Is used to show message box

GUI components

* Label
* Textbox
* Button
* Checkbox
* Radio Button
* List Box
* Picture Box
* Panel
* Group Box
* Menu Strip
* Font dialogue box
* Color dialogue box
* File dialogue box
* Background Worker

Stop watch in JavaScript

<https://medium.com/@harshitgambhir88/how-to-create-a-stopwatch-in-under-50-lines-of-javascript-code-963e42f9276f>

STOP WATCH

3600 seconds in one hour

60 minutes in one hour

60 seconds in one hour

Hours: (Time/3600)

Minutes: ((time%3600)/60)

Seconds: (Time/60)

Inheritence: Inheritance in C# (and other .NET languages) is a way to create a new class based on an existing class. It allows the new class to **inherit** properties and methods from the existing class, so you don’t have to rewrite the same code.\

TYPES OF INHERITANCE:

**Single Inheritance**

In single inheritance, a class (child) inherits from one base (parent) class.

// Base class

public class Animal

{

public void Eat()

{

Console.WriteLine("Eating...");

}

}

// Derived class (Single Inheritance)

public class Dog : Animal

{

public void Bark()

{

Console.WriteLine("Barking...");

}

}

class Program

{

static void Main(string[] args)

{

Dog myDog = new Dog();

myDog.Eat(); // Inherited from Animal

myDog.Bark(); // Specific to Dog

}

}

**Multilevel Inheritance**

In multilevel inheritance, a class is derived from a class which is already derived from another class.

// Base class

public class Animal

{

public void Eat()

{

Console.WriteLine("Eating...");

}

}

// Derived class

public class Mammal : Animal

{

public void Walk()

{

Console.WriteLine("Walking...");

}

}

// Further derived class (Multilevel Inheritance)

public class Dog : Mammal

{

public void Bark()

{

Console.WriteLine("Barking...");

}

}

class Program

{

static void Main(string[] args)

{

Dog myDog = new Dog();

myDog.Eat(); // Inherited from Animal

myDog.Walk(); // Inherited from Mammal

myDog.Bark(); // Specific to Dog

}

}

**Hierarchical Inheritance**

In hierarchical inheritance, multiple derived classes inherit from the same base class.

// Base class

public class Animal

{

public void Eat()

{

Console.WriteLine("Eating...");

}

}

// Derived class 1

public class Dog : Animal

{

public void Bark()

{

Console.WriteLine("Barking...");

}

}

// Derived class 2

public class Cat : Animal

{

public void Meow()

{

Console.WriteLine("Meowing...");

}

}

class Program

{

static void Main(string[] args)

{

Dog myDog = new Dog();

myDog.Eat(); // Inherited from Animal

myDog.Bark(); // Specific to Dog

Cat myCat = new Cat();

myCat.Eat(); // Inherited from Animal

myCat.Meow(); // Specific to Cat

}

}**Multiple Inheritance (Not Directly Supported in C#)**

C# does not support multiple inheritance directly (where a class inherits from more than one base class) but allows it through **interfaces**.

**Example using Interfaces (to simulate multiple inheritance):**

// Interface 1

public interface IAnimal

{

void Eat();

}

// Interface 2

public interface IPet

{

void Play();

}

// Class implementing multiple interfaces

public class Dog : IAnimal, IPet

{

public void Eat()

{

Console.WriteLine("Eating...");

}

public void Play()

{

Console.WriteLine("Playing...");

}

}

class Program

{

static void Main(string[] args)

{

Dog myDog = new Dog();

myDog.Eat(); // Implemented from IAnimal

myDog.Play(); // Implemented from IPet

}

}**Hybrid Inheritance**

Hybrid inheritance is a combination of two or more types of inheritance. In C#, since multiple inheritance is not directly supported, hybrid inheritance is often achieved using a mix of class inheritance and interfaces.

// Base class

public class Animal

{

public void Eat()

{

Console.WriteLine("Eating...");

}

}

// Interface

public interface IPet

{

void Play();

}

// Derived class (Single Inheritance + Interface Implementation)

public class Dog : Animal, IPet

{

public void Bark()

{

Console.WriteLine("Barking...");

}

public void Play()

{

Console.WriteLine("Playing...");

}

}

class Program

{

static void Main(string[] args)

{

Dog myDog = new Dog();

myDog.Eat(); // Inherited from Animal

myDog.Bark(); // Specific to Dog

myDog.Play(); // Implemented from IPet

}

}

**Abstraction** is the concept of hiding the complex details of how something works and showing only the necessary functionality to the user. In other words, it means focusing on **what** an object can do rather than **how** it does it.

Share



**You said:**

what is inheeritance in simple words in .neg language

**ChatGPT said:**

ChatGPT

Inheritance in C# (and other .NET languages) is a way to create a new class based on an existing class. It allows the new class to **inherit** properties and methods from the existing class, so you don’t have to rewrite the same code.

For example, if you have a class called Animal with properties like Name and methods like Eat(), you could create a new class Dog that inherits from Animal. The Dog class will automatically have the Name property and Eat() method, but you can also add extra features like a Bark() method specifically for dogs.

It’s like getting a blueprint for free and then adding your own custom features!

4o

**You said:**

give an example in. net lNGUgw

**ChatGPT said:**

ChatGPT

Here’s an example of inheritance in C# (.NET language):

csharp

Copy code

// Base class

public class Animal

{

public string Name { get; set; }

public void Eat()

{

Console.WriteLine($"{Name} is eating.");

}

}

// Derived class

public class Dog : Animal

{

public void Bark()

{

Console.WriteLine($"{Name} is barking.");

}

}

class Program

{

static void Main(string[] args)

{

// Create an instance of Dog

Dog myDog = new Dog();

myDog.Name = "Buddy";

// Call methods from both the base class and the derived class

myDog.Eat(); // Inherited from Animal class

myDog.Bark(); // Specific to Dog class

}

}

**Explanation:**

* The Animal class is the **base class**. It has a property Name and a method Eat().
* The Dog class is the **derived class**. It inherits the Name property and Eat() method from the Animal class, and it also has its own method Bark().
* In the Main() method, we create a Dog object and can call both the inherited method (Eat()) and the method specific to Dog (Bark()).

This is how inheritance works in C#.

4o

**You said:**

can we say base class is a parent and derived is child

**ChatGPT said:**

ChatGPT

Yes, exactly! In object-oriented programming, including in .NET languages like C#, the **base class** is often referred to as the **parent class**, and the **derived class** is called the **child class**.

* **Parent (Base) class**: The class from which properties and methods are inherited.
* **Child (Derived) class**: The class that inherits from the parent class and can add its own features.

So in the example above:

* Animal is the **parent class** (or base class).
* Dog is the **child class** (or derived class).

This relationship is also called a **"is-a" relationship**—for example, a Dog **is an** Animal.

4o

**You said:**

what are types of inheritence give example on each

**ChatGPT said:**

ChatGPT

In C# and other .NET languages, there are several types of inheritance. Let’s go through each type with a brief explanation and examples.

**1. Single Inheritance**

In single inheritance, a class (child) inherits from one base (parent) class.

**Example:**

csharp

Copy code

// Base class

public class Animal

{

public void Eat()

{

Console.WriteLine("Eating...");

}

}

// Derived class (Single Inheritance)

public class Dog : Animal

{

public void Bark()

{

Console.WriteLine("Barking...");

}

}

class Program

{

static void Main(string[] args)

{

Dog myDog = new Dog();

myDog.Eat(); // Inherited from Animal

myDog.Bark(); // Specific to Dog

}

}

Here, Dog inherits from Animal. This is **single inheritance** since there’s only one parent class (Animal).

**2. Multilevel Inheritance**

In multilevel inheritance, a class is derived from a class which is already derived from another class.

**Example:**

csharp

Copy code

// Base class

public class Animal

{

public void Eat()

{

Console.WriteLine("Eating...");

}

}

// Derived class

public class Mammal : Animal

{

public void Walk()

{

Console.WriteLine("Walking...");

}

}

// Further derived class (Multilevel Inheritance)

public class Dog : Mammal

{

public void Bark()

{

Console.WriteLine("Barking...");

}

}

class Program

{

static void Main(string[] args)

{

Dog myDog = new Dog();

myDog.Eat(); // Inherited from Animal

myDog.Walk(); // Inherited from Mammal

myDog.Bark(); // Specific to Dog

}

}

Here, Dog is derived from Mammal, which is derived from Animal, forming a **multilevel inheritance** chain.

**3. Hierarchical Inheritance**

In hierarchical inheritance, multiple derived classes inherit from the same base class.

**Example:**

csharp

Copy code

// Base class

public class Animal

{

public void Eat()

{

Console.WriteLine("Eating...");

}

}

// Derived class 1

public class Dog : Animal

{

public void Bark()

{

Console.WriteLine("Barking...");

}

}

// Derived class 2

public class Cat : Animal

{

public void Meow()

{

Console.WriteLine("Meowing...");

}

}

class Program

{

static void Main(string[] args)

{

Dog myDog = new Dog();

myDog.Eat(); // Inherited from Animal

myDog.Bark(); // Specific to Dog

Cat myCat = new Cat();

myCat.Eat(); // Inherited from Animal

myCat.Meow(); // Specific to Cat

}

}

In this example, both Dog and Cat inherit from the Animal class, forming a **hierarchical inheritance** structure.

**4. Multiple Inheritance (Not Directly Supported in C#)**

C# does not support multiple inheritance directly (where a class inherits from more than one base class) but allows it through **interfaces**.

**Example using Interfaces (to simulate multiple inheritance):**

csharp

Copy code

// Interface 1

public interface IAnimal

{

void Eat();

}

// Interface 2

public interface IPet

{

void Play();

}

// Class implementing multiple interfaces

public class Dog : IAnimal, IPet

{

public void Eat()

{

Console.WriteLine("Eating...");

}

public void Play()

{

Console.WriteLine("Playing...");

}

}

class Program

{

static void Main(string[] args)

{

Dog myDog = new Dog();

myDog.Eat(); // Implemented from IAnimal

myDog.Play(); // Implemented from IPet

}

}

In this case, Dog class is implementing two interfaces: IAnimal and IPet, which gives the behavior similar to **multiple inheritance**.

**5. Hybrid Inheritance**

Hybrid inheritance is a combination of two or more types of inheritance. In C#, since multiple inheritance is not directly supported, hybrid inheritance is often achieved using a mix of class inheritance and interfaces.

**Example of Hybrid Inheritance:**

csharp

Copy code

// Base class

public class Animal

{

public void Eat()

{

Console.WriteLine("Eating...");

}

}

// Interface

public interface IPet

{

void Play();

}

// Derived class (Single Inheritance + Interface Implementation)

public class Dog : Animal, IPet

{

public void Bark()

{

Console.WriteLine("Barking...");

}

public void Play()

{

Console.WriteLine("Playing...");

}

}

class Program

{

static void Main(string[] args)

{

Dog myDog = new Dog();

myDog.Eat(); // Inherited from Animal

myDog.Bark(); // Specific to Dog

myDog.Play(); // Implemented from IPet

}

}

Here, Dog inherits from Animal (single inheritance) and implements the IPet interface, demonstrating **hybrid inheritance**.

In summary, the types of inheritance are:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**
4. **Multiple Inheritance** (through interfaces)
5. **Hybrid Inheritance** (combination of above types)

Each one allows classes to inherit properties and behaviors in a structured way, making code more reusable and organized.

4o

**You said:**

what is polymarphism and its typpes and give example

**ChatGPT said:**

ChatGPT

**Polymorphism** in C# (and other object-oriented languages) is a concept that allows objects to be treated as instances of their parent class, but behave differently depending on the actual derived class that is implementing them. This makes the code more flexible and extensible.

There are two main types of polymorphism:

**1. Compile-time Polymorphism (Method Overloading)**

Compile-time polymorphism is also known as **method overloading** or **operator overloading**. This happens when multiple methods in the same class have the same name but different parameters (different signatures).

**Example:**

csharp

Copy code

public class Calculator

{

// Method with two integer parameters

public int Add(int a, int b)

{

return a + b;

}

// Overloaded method with three integer parameters

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

{

return a + b + c;

}

}

class Program

{

static void Main(string[] args)

{

Calculator calc = new Calculator();

Console.WriteLine(calc.Add(5, 10)); // Calls the method with two parameters

Console.WriteLine(calc.Add(5, 10, 15)); // Calls the overloaded method with three parameters

}

}

In this example, the Add method is overloaded with different numbers of parameters, demonstrating **compile-time polymorphism**.

**2. Run-time Polymorphism (Method Overriding)**

Run-time polymorphism is achieved through **method overriding**, where a method in a derived class has the same signature as a method in the base class, but the derived class provides its own implementation.

This is also known as **dynamic polymorphism** or **virtual methods**.

**Example:**

csharp

Copy code

// Base class

public class Animal

{

// Virtual method that can be overridden in derived classes

public virtual void Speak()

{

Console.WriteLine("Animal is making a sound.");

}

}

// Derived class 1

public class Dog : Animal

{

// Override the base class method

public override void Speak()

{

Console.WriteLine("Dog is barking.");

}

}

// Derived class 2

public class Cat : Animal

{

// Override the base class method

public override void Speak()

{

Console.WriteLine("Cat is meowing.");

}

}

class Program

{

static void Main(string[] args)

{

// Parent reference pointing to child objects (polymorphism)

Animal myAnimal = new Dog();

myAnimal.Speak(); // Output: Dog is barking

myAnimal = new Cat();

myAnimal.Speak(); // Output: Cat is meowing

}

}

In this example, the Speak() method is overridden in both Dog and Cat classes. When myAnimal points to a Dog object, it calls the Dog’s version of Speak(), and when it points to a Cat object, it calls the Cat’s version of Speak(). This is **run-time polymorphism**.

**Summary of Polymorphism Types:**

1. **Compile-time Polymorphism** (Method Overloading):
   * Same method name with different signatures (parameters) in the same class.
   * Resolved at compile time.
   * Example: Overloading methods in a class (like the Add method).
2. **Run-time Polymorphism** (Method Overriding):
   * A method in the base class is overridden by a method in the derived class.
   * Resolved at runtime.
   * Example: Using virtual methods and overriding them (like the Speak() method).

**Key Differences**:

* **Compile-time Polymorphism**: Resolved during compilation. You define multiple methods with the same name but different parameters (method overloading).
* **Run-time Polymorphism**: Resolved during execution. The method that gets called is determined by the actual object type (method overriding).

Polymorphism makes programs more flexible and allows objects of different types to be processed uniformly.

4o

**You said:**

what does man by ploy,mraephisim

**ChatGPT said:**

ChatGPT

**Polymorphism** comes from the Greek words "poly" (meaning **many**) and "morph" (meaning **forms**). So, polymorphism literally means "many forms."

In the context of object-oriented programming (OOP), polymorphism refers to the ability of a single action or method to behave differently based on the object it is acting upon. It allows one interface or method to be used for different types of objects, and each object can respond in its own way.

For example:

* In method **overloading** (compile-time polymorphism), the same method name can perform different tasks based on the parameters passed.
* In method **overriding** (run-time polymorphism), different classes can provide their own implementation of the same method, even if they share a common parent class.

In simple words, polymorphism allows objects to "take many forms" and behave differently depending on their specific type, even if they share the same method names.

4o

**You said:**

what is abstraction? in c# and deefine i in simple words and also abstract and interface class with example

**ChatGPT said:**

ChatGPT

**Abstraction in C#**

**Abstraction** is the concept of hiding the complex details of how something works and showing only the necessary functionality to the user. In other words, it means focusing on **what** an object can do rather than **how** it does it.

In C#, abstraction can be achieved using **abstract classes** and **interfaces**. These are used to provide a blueprint for other classes without giving the full implementation. The actual implementation is done in the derived classes.

**Abstract Class in C#**

An **abstract class** is a class that cannot be instantiated (you cannot create objects of it directly). It can contain both abstract methods (methods without any implementation) and normal methods (with implementation). Derived classes must provide the implementation for the abstract methods.

// Abstract class

public abstract class Animal

{

// Abstract method (no implementation)

public abstract void MakeSound();

// Regular method (with implementation)

public void Sleep()

{

Console.WriteLine("Sleeping...");

}

}

// Derived class from the abstract class

public class Dog : Animal

{

// Providing implementation of abstract method

public override void MakeSound()

{

Console.WriteLine("Dog is barking.");

}

}

class Program

{

static void Main(string[] args)

{

// Dog class can be instantiated

Dog myDog = new Dog();

myDog.MakeSound(); // Output: Dog is barking.

myDog.Sleep(); // Output: Sleeping...

}

}

**Key points about abstract class:**

* Cannot be instantiated.
* Can contain both abstract methods (without body) and normal methods (with body).
* Derived classes **must** implement the abstract methods.