**Polymorphism:**

Polymorphism is an object-oriented programming concept that refers to the ability of a variable, function or object to take on multiple forms. A language that features polymorphism allows developers to program in the general rather than program in the specific.

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

* At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this polymorphism occurs, the object's declared type is no longer identical to its run-time type.
* Base classes may define and implement [virtual](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual) *methods*, and derived classes can [override](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override) them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. In your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Virtual methods enable you to work with groups of related objects in a uniform way. For example, suppose you have a drawing application that enables a user to create various kinds of shapes on a drawing surface. You don't know at compile time which specific types of shapes the user will create. However, the application has to keep track of all the various types of shapes that are created, and it has to update them in response to user mouse actions. You can use polymorphism to solve this problem in two basic steps:

1. Create a class hierarchy in which each specific shape class derives from a common base class.
2. Use a virtual method to invoke the appropriate method on any derived class through a single call to the base class method.

First, create a base class called Shape, and derived classes such as Rectangle, Circle, and Triangle. Give the Shape class a virtual method called Draw, and override it in each derived class to draw the particular shape that the class represents. Create a List<Shape> object and add a Circle, Triangle, and Rectangle to it.

C#Copy

public class Shape

{

// A few example members

public int X { get; private set; }

public int Y { get; private set; }

public int Height { get; set; }

public int Width { get; set; }

// Virtual method

public virtual void Draw()

{

Console.WriteLine("Performing base class drawing tasks");

}

}

public class Circle : Shape

{

public override void Draw()

{

// Code to draw a circle...

Console.WriteLine("Drawing a circle");

base.Draw();

}

}

public class Rectangle : Shape

{

public override void Draw()

{

// Code to draw a rectangle...

Console.WriteLine("Drawing a rectangle");

base.Draw();

}

}

public class Triangle : Shape

{

public override void Draw()

{

// Code to draw a triangle...

Console.WriteLine("Drawing a triangle");

base.Draw();

}

}

To update the drawing surface, use a [foreach](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements" \l "the-foreach-statement) loop to iterate through the list and call the Draw method on each Shape object in the list. Even though each object in the list has a declared type of Shape, it's the run-time type (the overridden version of the method in each derived class) that will be invoked.

C#Copy

// Polymorphism at work #1: a Rectangle, Triangle and Circle

// can all be used whereever a Shape is expected. No cast is

// required because an implicit conversion exists from a derived

// class to its base class.

var shapes = new List<Shape>

{

new Rectangle(),

new Triangle(),

new Circle()

};

// Polymorphism at work #2: the virtual method Draw is

// invoked on each of the derived classes, not the base class.

foreach (var shape in shapes)

{

shape.Draw();

}

/\* Output:

Drawing a rectangle

Performing base class drawing tasks

Drawing a triangle

Performing base class drawing tasks

Drawing a circle

Performing base class drawing tasks

\*/

In C#, every type is polymorphic because all types, including user-defined types, inherit from [Object](https://docs.microsoft.com/en-us/dotnet/api/system.object).

Another Example in JAVA:

//package name

package com. methods;

//class name

public class Example2

{

            //product method without parameters

            public static void product()

            {

            int length = 30;

            int breadth = 10;

            int area = length\*breadth;

            System.out.println(area);

              }

              //product method with parameters

              public static void product(int num1, int num2)

              {

            int result;

            result = num1\*num2;

            System.out.println(“The Result: “+result);

            }

            //main() method

            public static void main(String[] args)

            {

            //calling product() method without parameters

            product();

            System.out.println(“===============”);

            //calling product() method with parameters

            product(20, 30);//here 20, 30 are arguments

          }

}