C# - Polymorphism

The word **polymorphism** means having many forms.

It can also be said as, single form behaving differently in different situations.

A single function or single operator has different characters at different places.

In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

Real World Example

President of United States Mr. Donald Trump is in the form of a President, when he is at White House.



But when he is at his company premises (Trump Organisation),then he is in the form of a businessman.



Types of Polymorphism

## Polymorphism can be static or dynamic.

## In static polymorphism, the response to a function is determined at the compile time.

## In dynamic polymorphism, it is decided at run-time.

Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are −

* Function overloading
* Operator overloading

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

The following example shows using function **print()** to print different data types −

[Live Demo](http://tpcg.io/rUApo0)

using System;

namespace PolymorphismApplication {

class Printdata {

void print(int i) {

Console.WriteLine("Printing int: {0}", i );

}

void print(double f) {

Console.WriteLine("Printing float: {0}" , f);

}

void print(string s) {

Console.WriteLine("Printing string: {0}", s);

}

static void Main(string[] args) {

Printdata p = new Printdata();

// Call print to print integer

p.print(5);

// Call print to print float

p.print(500.263);

// Call print to print string

p.print("Hello C++");

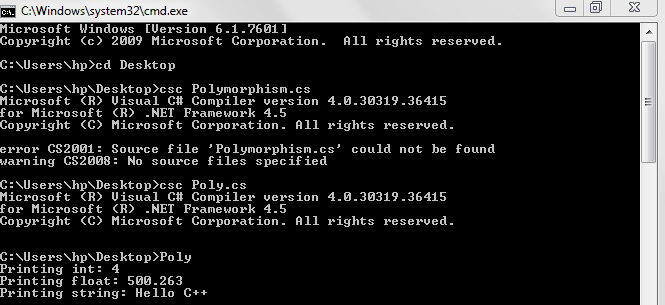
Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result –



Operator Overloading

You can redefine or overload most of the built-in operators available in C#.

Thus a programmer can use operators with user-defined types as well. Overloaded operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined. Similar to any other function, an overloaded operator has a return type and a parameter list.

For example, go through the following function −

public static Box operator+ (Box b, Box c) {

Box box = new Box();

box.length = b.length + c.length;

box.breadth = b.breadth + c.breadth;

box.height = b.height + c.height;

return box;

}

The above function implements the addition operator (+) for a user-defined class Box. It adds the attributes of two Box objects and returns the resultant Box object.

Implementing the Operator Overloading

The following program shows the complete implementation −

[Live Demo](http://tpcg.io/rv16gU)

using System;

namespace OperatorOvlApplication {

class Box {

private double length; // Length of a box

private double breadth; // Breadth of a box

private double height; // Height of a box

public double getVolume() {

return length \* breadth \* height;

}

public void setLength( double len ) {

length = len;

}

public void setBreadth( double bre ) {

breadth = bre;

}

public void setHeight( double hei ) {

height = hei;

}

// Overload + operator to add two Box objects.

public static Box operator+ (Box b, Box c) {

Box box = new Box();

box.length = b.length + c.length;

box.breadth = b.breadth + c.breadth;

box.height = b.height + c.height;

return box;

}

}

class Tester {

static void Main(string[] args) {

Box Box1 = new Box(); // Declare Box1 of type Box

Box Box2 = new Box(); // Declare Box2 of type Box

Box Box3 = new Box(); // Declare Box3 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

Console.WriteLine("Volume of Box1 : {0}", volume);

// volume of box 2

volume = Box2.getVolume();

Console.WriteLine("Volume of Box2 : {0}", volume);

// Add two object as follows:

Box3 = Box1 + Box2;

// volume of box 3

volume = Box3.getVolume();

Console.WriteLine("Volume of Box3 : {0}", volume);

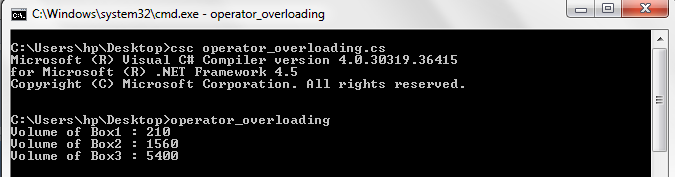
Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result –



Overloadable and Non-Overloadable Operators

The following table describes the overload ability of the operators in C# −

|  |  |
| --- | --- |
| **Sr.No.** | **Operators & Description** |
| 1 | **+, -, !, ~, ++, --**  These unary operators take one operand and can be overloaded. |
| 2 | **+, -, \*, /, %**  These binary operators take one operand and can be overloaded. |
| 3 | **==, !=, <, >, <=, >=**  The comparison operators can be overloaded. |
| 4 | **&&, ||**  The conditional logical operators cannot be overloaded directly. |
| 5 | **+=, -=, \*=, /=, %=**  The assignment operators cannot be overloaded. |
| 6 | **=, ., ?:, ->, new, is, sizeof, typeof**  These operators cannot be overloaded. |

Example

In the light of the above discussions, let us extend the preceding example, and overload few more operators −

[Live Demo](http://tpcg.io/IwiEcC)

using System;

namespace OperatorOvlApplication {

class Box {

private double length; // Length of a box

private double breadth; // Breadth of a box

private double height; // Height of a box

public double getVolume() {

return length \* breadth \* height;

}

public void setLength( double len ) {

length = len;

}

public void setBreadth( double bre ) {

breadth = bre;

}

public void setHeight( double hei ) {

height = hei;

}

// Overload + operator to add two Box objects.

public static Box operator+ (Box b, Box c) {

Box box = new Box();

box.length = b.length + c.length;

box.breadth = b.breadth + c.breadth;

box.height = b.height + c.height;

return box;

}

public static bool operator == (Box lhs, Box rhs) {

bool status = false;

if (lhs.length == rhs.length && lhs.height == rhs.height

&& lhs.breadth == rhs.breadth) {

status = true;

}

return status;

}

public static bool operator !=(Box lhs, Box rhs) {

bool status = false;

if (lhs.length != rhs.length || lhs.height != rhs.height ||

lhs.breadth != rhs.breadth) {

status = true;

}

return status;

}

public static bool operator <(Box lhs, Box rhs) {

bool status = false;

if (lhs.length < rhs.length && lhs.height < rhs.height

&& lhs.breadth < rhs.breadth) {

status = true;

}

return status;

}

public static bool operator >(Box lhs, Box rhs) {

bool status = false;

if (lhs.length > rhs.length && lhs.height >

rhs.height && lhs.breadth > rhs.breadth) {

status = true;

}

return status;

}

public static bool operator <=(Box lhs, Box rhs) {

bool status = false;

if (lhs.length <= rhs.length && lhs.height

<= rhs.height && lhs.breadth <= rhs.breadth) {

status = true;

}

return status;

}

public static bool operator >=(Box lhs, Box rhs) {

bool status = false;

if (lhs.length >= rhs.length && lhs.height

>= rhs.height && lhs.breadth >= rhs.breadth) {

status = true;

}

return status;

}

public override string ToString() {

return String.Format("({0}, {1}, {2})", length, breadth, height);

}

}

class Tester {

static void Main(string[] args) {

Box Box1 = new Box(); // Declare Box1 of type Box

Box Box2 = new Box(); // Declare Box2 of type Box

Box Box3 = new Box(); // Declare Box3 of type Box

Box Box4 = new Box();

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

//displaying the Boxes using the overloaded ToString():

Console.WriteLine("Box 1: {0}", Box1.ToString());

Console.WriteLine("Box 2: {0}", Box2.ToString());

// volume of box 1

volume = Box1.getVolume();

Console.WriteLine("Volume of Box1 : {0}", volume);

// volume of box 2

volume = Box2.getVolume();

Console.WriteLine("Volume of Box2 : {0}", volume);

// Add two object as follows:

Box3 = Box1 + Box2;

Console.WriteLine("Box 3: {0}", Box3.ToString());

// volume of box 3

volume = Box3.getVolume();

Console.WriteLine("Volume of Box3 : {0}", volume);

//comparing the boxes

if (Box1 > Box2)

Console.WriteLine("Box1 is greater than Box2");

else

Console.WriteLine("Box1 is not greater than Box2");

if (Box1 < Box2)

Console.WriteLine("Box1 is less than Box2");

else

Console.WriteLine("Box1 is not less than Box2");

if (Box1 >= Box2)

Console.WriteLine("Box1 is greater or equal to Box2");

else

Console.WriteLine("Box1 is not greater or equal to Box2");

if (Box1 <= Box2)

Console.WriteLine("Box1 is less or equal to Box2");

else

Console.WriteLine("Box1 is not less or equal to Box2");

if (Box1 != Box2)

Console.WriteLine("Box1 is not equal to Box2");

else

Console.WriteLine("Box1 is not greater or equal to Box2");

Box4 = Box3;

if (Box3 == Box4)

Console.WriteLine("Box3 is equal to Box4");

else

Console.WriteLine("Box3 is not equal to Box4");

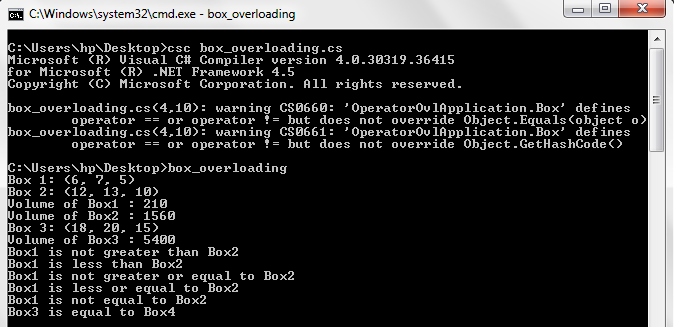
Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result −



Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Here are the rules about abstract classes −

* You cannot create an instance of an abstract class
* You cannot declare an abstract method outside an abstract class
* When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed.

Method Overriding

Method overriding is a feature that allows you to invoke functions (that have the same signatures) that belong to different classes in the same hierarchy of inheritance using the base class reference.

C# makes use of two keywords: virtual and overrides to accomplish Method overriding.

class BC

{

public virtual void Display()

{

System.Console.WriteLine("BC::Display");

}

}

class DC : BC

{

public override void Display()

{

System.Console.WriteLine("DC::Display");

}

}

class Demo

{

public static void Main()

{

BC b;

b = new BC();

b.Display();

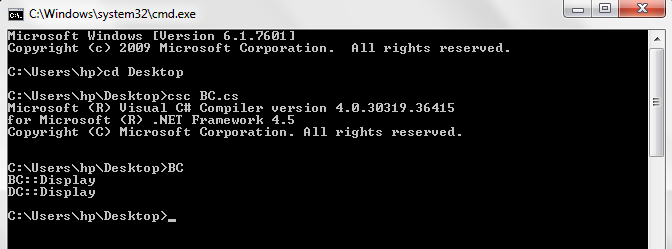
b = new DC();

b.Display();

}

}

#### Output



The following program demonstrates an abstract class −

[Live Demo](http://tpcg.io/oFyFed)

using System;

namespace PolymorphismApplication {

abstract class Shape {

public abstract int area();

}

class Rectangle: Shape {

private int length;

private int width;

public Rectangle( int a = 0, int b = 0) {

length = a;

width = b;

}

public override int area () {

Console.WriteLine("Rectangle class area :");

return (width \* length);

}

}

class RectangleTester {

static void Main(string[] args) {

Rectangle r = new Rectangle(10, 7);

double a = r.area();

Console.WriteLine("Area: {0}",a);

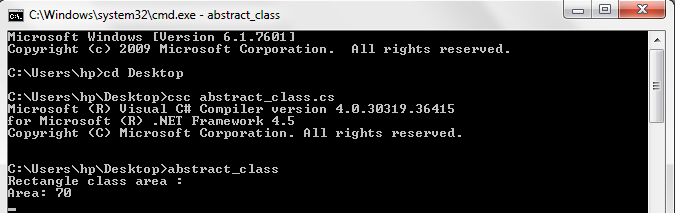
Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result –



When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

The following program demonstrates this −

[Live Demo](http://tpcg.io/3KXVP8)

using System;

namespace PolymorphismApplication {

class Shape {

protected int width, height;

public Shape( int a = 0, int b = 0) {

width = a;

height = b;

}

public virtual int area() {

Console.WriteLine("Parent class area :");

return 0;

}

}

class Rectangle: Shape {

public Rectangle( int a = 0, int b = 0): base(a, b) {

}

public override int area () {

Console.WriteLine("Rectangle class area :");

return (width \* height);

}

}

class Triangle: Shape {

public Triangle(int a = 0, int b = 0): base(a, b) {

}

public override int area() {

Console.WriteLine("Triangle class area :");

return (width \* height / 2);

}

}

class Caller {

public void CallArea(Shape sh) {

int a;

a = sh.area();

Console.WriteLine("Area: {0}", a);

}

}

class Tester {

static void Main(string[] args) {

Caller c = new Caller();

Rectangle r = new Rectangle(10, 7);

Triangle t = new Triangle(10, 5);

c.CallArea(r);

c.CallArea(t);

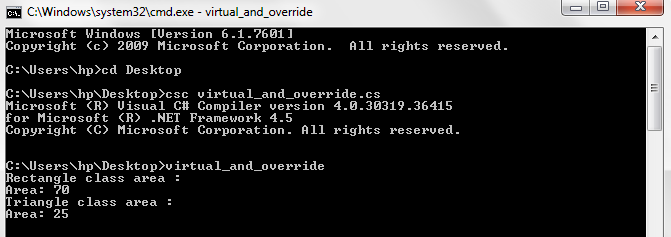
Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result –



Operator Overriding

Operator overriding is not supported in c#. The term Overriding is used when a method is inherited by a subclass and the subclass overrides it with its own implementation. Operators are all static in C# and cannot be overridden.

Overloading means that another method with the same name but a different signature (arguments) is defined. That's what you can do with operators.

This is extremely important to know when writing operators in C#. The operator is bound at compile time. The effective type which is passed to the operator at runtime is not important at all.

Eg. you write some comparison operator

public static bool operator==(MyClass c1, MyClass c2)

{

//...

}

And have the following code:

object myObj1 = new MyClass();

object myObj2 = new MyClass();

if (myObj1 == myObj2)

//...

object's operator will be called, not your own, because the arguments are references of type object.

References :-

1.tutorialspoint.com

2.csharpcorner.com

3.javatpoint.com