Polymorphism in C

Polymorphism in C # can be static or dynamic. In particular, a static polymorph can be called a static polymorph, a response to a function defined at compile time and a dynamic polymorph can be called dynamic polymorphism, defined at runtime .

The word **polymorphism** means that there are many forms. In object-oriented programming, polymorphism is often expressed as "one Interface, multiple functions".

Polymorphism in **C #** can be **static** or **dynamic** . In particular, a **static polymorph** can be called a static polymorph, a response to a function defined at compile time and a **dynamic polymorph** can be called **dynamic** polymorphism, defined at runtime .

Static polymorphism in C

The technique of linking a function to an object during compilation is called Early Binding. It is also called Static Binding. C # provides two techniques for deploying static polymorphisms, namely:

1. Load overlap (Function overloading)
2. Load operator stack (Operator overloading)

We have a separate discussion about operator overloading in C #.

Load overlap in C

You can have multiple definitions for the same function name in the same scope. These definitions of the function must be different: such as type and / or number of parameters in the parameter list. In C #, you cannot overload function declarations but only in return types.

The following example illustrates how to use the **print ()** function to print different data types in C #:

```
 using System ; namespace VdNapChong { class InDuLieu { void print ( int  i ) { C
?
 nguyên: {0}" ,  i  ); } void print ( double  f ) { Console . WriteLine ( "In s
? th?
p phân: {0}" ,  f ); } void print ( string  s ) { Console . WriteLine ( "In chu
?
i: {0}" ,  s ); } static void Main ( string []  args ) { InDuLieu  p  = new InDu
?i hàm in s? nguyên  p . print ( 9 ); // G?i hàm in s? th?
p phân  p . print ( 501.263 ); // G?i hàm in chu?i  p . print ( "H?c C# th?
t vui!" ); Console . ReadKey (); } } }
```

If you have read the previous C # articles, you will know, otherwise use the **Console.ReadKey** () command; then the program will run and finish (too fast to see the results in time). This command allows us to see the results more clearly.

Compiling and running the above C # program will produce the following results:

```
Print integer: 9
Print decimal number: 501.263
Chain printing: Learning C # is fun!
```

Dynamic polymorphism in C

C # allows you to create abstract classes that are used to provide a local class implementation of an Interface. Implementation is completed when a derived class inherits from it. Abstract classes contain abstract methods implemented by derived classes. This derived class has more specialized functions.

Here are some rules about abstract classes in C #:

1. You cannot create an Instance of an abstract class.
2. You cannot declare an abstract method outside an abstract class.
3. When a sealed class is sealed, it cannot be derived, abstract classes cannot be declared sealed.

The following example illustrates an abstract class in C #: create 3 classes named **Shape, HinhChuNhat and TimDienTich respectively** :

```
 using System ; namespace VdLopAbstract {
 // L?p Shape là m?t l?
p abstract abstract class Shape { public abstract int  area (); } // L?
p HinhChuNhat là l?p d?n xu?t t? l?
p Shape class HinhChuNhat : Shape { private int  dai ; private int  rong ; public
?n tích hình ch? nh?t" ); return ( dai  *  rong ); } }
 // L?p TimDienTich ch?a ph??ng th?c main() ??
 thao tác trên HinhChuNhat class TimDienTich { static void Main ( string []  args
?n tích là: {0}" , a ); Console . ReadKey (); } } }
```

Compiling and running the above C # program will produce the following results:

```
Find the area of ??a rectangle
Area is: 90
```

When you have a function defined in a class that you want to deploy a derived class, you use the virtual function in C #. Virtual functions can be deployed in different ways in different inherited classes and calling these functions will be decided at runtime.

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

The following example illustrates this: creating 5 classes named as follows, Image, Image, Timeline, HienThiDienTich and TimDienTich.

```
 using System ; namespace VdLopAbstract {
 // L?
p Hinh là lop abstract. class Hinh { protected int  dai ,  rong ; public Hinh ( i
?n tích c?a l?p cha là:" ); return 0 ; } }
 // L?p HinhChuNhat k? th?a t? l?
p Hinh. class HinhChuNhat : Hinh { public HinhChuNhat ( int  a  = 0 , int  b  = 0
?n tích l?p HinhChuNhat là:" ); return ( dai  *  rong ); } }
 // L?p HinhTamGiac k? th?a t? l?
p Hình class HinhTamGiac : Hinh { public HinhTamGiac ( int  a  = 0 , int  b  = 0
?n tích l?p HinhTamGiac là:" ); return ( dai  *  rong  / 2 ); } }
 // In d? li?u di?
n tích ra màn hình. class HienThiDienTich { public void CallArea ( Hinh  sh ) { i
?n tích: {0}" ,  a ); } }
 // TimDienTich ch?a main() thao tác v?i các ??i t??
ng. class TimDienTich { static void Main ( string []  args ) { HienThiDienTich  o
```

Compiling and running the above C # program will produce the following results:

```
Class area of ??the image is:
Area: 70
The area of ??Hinh Tam's class is:
Area: 25
```

Follow tutorialspoint

Previous article: Calculating inheritance in C #

Next lesson: Operator overloading in C #