Перекрывание и переопределение методов при наследовании

admin
January 16 edited January 16

В OverScript методы в производных классах перекрывают методы в базовом, но не переопределяют их, если вы не используете модификаторы virtual/override.

```
Child c=new Child();
Parent p=c;
c.Test(); //Hello
p.Test(); //Test

class Parent{
    public Test(){
        WriteLine(GetStr());
    }
    string GetStr(){
        return "Test";
    }
}
class Child:Parent{
    string GetStr(){
        return "Hello";
    }
}
```

Если нужно, чтобы p.Test() выводил "Hello", а не "Test", то сделаем GetStr() виртуальным:

```
Child c=new Child();
Parent p=c;
c.Test(); //Hello
p.Test(); //Hello

class Parent{
    public Test(){
        WriteLine(GetStr());
    }
    virtual string GetStr(){
        return "Test";
    }
}
class Child:Parent{
    override string GetStr(){
        return "Hello";
    }
}
```

Если же нужно, чтобы c.Test() выводил "Test", а не "Hello", то сделаем Test() фиксированным (fixed). В таком случае он всегда будет обращаться к GetStr() из Parent.

```
Child c=new Child();
Parent p=c;
c.Test(); //Test
p.Test(); //Test

class Parent{
    public fixed Test(){
        WriteLine(GetStr());
    }
```

```
    string GetStr(){
        return "Test";
    }
}
class Child:Parent{
    string GetStr(){
        return "Hello";
    }
}
```

Теперь посмотрим, как сделано в Python, PHP, Java, C#.
Python:

```
class Parent(object):
    def GetStr(self):
        return "Test"

    def Test(self):
        print(self.GetStr())


class Child(Parent):
    def GetStr(self):
        return "Hello"


c=Child()
c.Test() #Hello
```

Как видим, используется GetStr из Child. В Python все методы являются виртуальными.

PHP:

```
$child = new Child();
$child->Test(); //Hello

class ParentClass
{
    function GetStr(){
        return 'Test';
    }
    public function Test(){
        echo($this->GetStr());
    }
}

class Child extends ParentClass
{
    function GetStr(){
        return 'Hello';
    }
}
```

Тут также используется GetStr из Child.

Java:

```
public class MyClass{
    public static void main(String args[]){
        Child c=new Child();
        Parent p=c;
        c.Test(); //Hello
        p.Test(); //Hello
    }
}
```

```
class Parent {
    public void Test(){
        System.out.println(GetStr());
    }
    String GetStr() {
        return("Test");
    }
}

class Child extends Parent{
    String GetStr() {
        return("Hello");
    }
}
```

В обоих вызовах используется GetStr из Child. Т.е. методы виртуальные, даже если не используется аннотация `@Override`.

C#:

```
using System;
class Program{
    static void Main(string[] args)
    {
        Child c = new Child();
        Parent p = c;
        c.Test(); //Test
        p.Test(); //Test
    }
}
class Parent{
    public void Test(){
        Console.WriteLine(GetStr());
    }
    string GetStr() {
        return "Test";
    }
}
class Child:Parent{
    string GetStr(){
        return "Hello";
    }
}
```

В C# метод GetStr в производном классе не повлиял на работу метода Test. В терминологии OverScript все методы в C# фиксированные. Расфиксировать их нельзя, а можно только сделать виртуальными.

В OverScript методы по умолчанию наследуются с перекомпилированием (перестроением). Т.е. в производный класс копируется код метода, который компилируется в новом окружении, а значит будут использоваться методы, которые перекрывают методы из базового класса. Чтобы копировался не код, а только ссылка на метод, нужно использовать модификатор fixed. Чтобы метод вообще не копировался в производный класс, нужно пометить его модификатором exclusive.
Когда метод наследуется с перекомпилированием, то его новая версия будет использовать в том числе перекрывающие статические функции, поля и константы.

```
Child c=new Child();
Parent p=c;
c.Test(); //GetStr at Child: Hello; 2
p.Test(); //GetStr at Parent: Test; 1

class Parent{
    static string Str="Test";
    const int X=1;
```

```
    public Test(){
        WriteLine(GetStr());
    }
    static string GetStr(){
        return "GetStr at Parent: "+Str+"; "+X;
    }
}
class Child:Parent{
    static string Str="Hello";
    const int X=2;
    static string GetStr(){
        return "GetStr at Child: "+Str+"; "+X;
    }
}
```

Под компилированием я имею в виду построение из кода дерева операций, которое в абстрактном виде представляет структуру программы.
Проще всего понять наследование в OverScript, мысленно копируя код функций в производные классы. Методы копируются один за другим, и если в классе получается несколько с одинаковыми сигнатурами (имя + типы параметров) то вызывается всегда последний (на самом деле, интерпретатор их видит в обратном порядке и вызывает первый). При этом вы можете вызывать не только последнюю, а любую версию метода добавляя к имени метода соответствующее порядку кол-во символов $.

```
Baz b=new Baz();
b.Test(); //At Baz!
b.$Test(); //At Bar!
b.$$Test(); //At Foo!

class Foo{
    public Test(){WriteLine("At Foo!");}
}
class Bar:Foo{
    public Test(){WriteLine("At Bar!");}
}
class Baz:Bar{
    public Test(){WriteLine("At Baz!");}
}
```

Один $ - это вызвать предыдущий, $$ - предпредыдущий и т.д.

С конструкторами:

```
Baz b=new Baz(); //At Foo!

class Foo{
    New(){WriteLine("At Foo!");}
}
class Bar:Foo{
    New(){$New();}
}
class Baz:Bar{
    New(){$New();}
}
```

В этом примере конструктор Baz вызывает конструктор Bar, который вызывает конструктор Foo.