Подробно о виртуальных методах

admin
January 11 edited March 22

Для начала небольшой пример:

```
Bar b=new Bar();
b.Test(); //At Bar
Foo f=b;
f.Test(); //At Bar

class Foo{
    public virtual Test(){WriteLine("At Foo");}
}

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

Видим, что f.Test() - это вызов Test() из Bar, а не из Foo, хотя переменная f типа Foo. Это потому, что если метод помечен модификатором virtual, то каждый раз при обращении к нему производится поиск override -метода среди всех имеющихся у объекта. Если такой метод найден, то вызывается он, а если нет, то сам виртуальный. Ниже я подробно и, надеюсь, понятно расскажу, как происходит вызов виртуальных методов.
Когда я был начинающим программистом, я не очень хорошо понимал, как работают виртуальные методы. Это было из-за неправильного представления о наследовании методов. А ещё из-за непонимания того, что тип переменной и реальный тип объекта, который в ней хранится могут быть разными.
Итак, в OverScript наследование заключается в копировании в класс всех членов из родительского класса. Я пишу про OverScript, но это в той или иной мере справедливо и для других языков. В примере выше в классе Bar оказывается два метода Test. Если у одного метода есть несколько вариантов, то вызывается последний. Далее по шагам:
1. В первой строке создаётся экземпляр Bar, у которого один Test() из Foo, а другой из Bar.
2. Во второй строке происходит вызов Test(), но это не прямой вызов, а виртуальный, т.е. сначала происходит поиск нужного метода. Почему - напишу далее.
3. В третьей строке переменной типа Foo присваивается экземпляр Bar. Это возможно, т.к. Bar наследует от Foo. Простыми словами: переменная родительского типа может принимать объекты производных типов. В переменной f теперь лежит объект типа Bar (экземпляр класса Bar).
4. В четвёртой происходит вызов виртуального метода. Казалось бы, раз f типа Foo, значит должна вызываться Test() из Foo, но модификатор virtual указывает интерпретатору, что нужно произвести поиск override-метода. Этот поиск делается при каждом обращении к virtual или override методу. К override потому, что он сам тоже может быть перекрыт другим override-методом.

Как интерпретатор ищет нужный метод

Интерпретатор видит в экземпляре Bar (а переменная f ссылается именно на него, хоть и задана как Foo) методы в обратном порядке:

```
public override Test(){WriteLine("At Bar");}
public virtual Test(){WriteLine("At Foo");}
```

Он проходит от первого до последнего и, если метод помечен как override, то следующий метод заменяется на него. Этот следующий метод должен быть помечен как virtual или override, и не как sealed. Иначе при загрузке скрипта вылетит ошибка. В результате список методов будет такой:

```
public override Test(){WriteLine("At Bar");}
public override Test(){WriteLine("At Bar");}
```

Интерпретатор вызовет первый (т.е. последний в коде).
Теперь давайте сделаем Test() в Bar не override, а virtual.

```
Bar b=new Bar();
Foo f=b;
f.Test(); //At Foo

class Foo{
    public virtual Test(){WriteLine("At Foo");}
}

class Bar:Foo{
    public virtual Test(){WriteLine("At Bar");} //эта функция перекрывает Test из Foo, но не зам
}
```

Интерпретатор видит:

```
public virtual Test(){WriteLine("At Bar");}
public virtual Test(){WriteLine("At Foo");}
```

Получается, что первая - это Test() из Bar. Но интерпретатор отбрасывает методы, которые не в Foo или унаследованных Foo классах. Т.е. методы Bar отбрасываются и остаётся одна Test() из Foo. То же самое было бы и без модификатора virtual.
Посмотрим на следующий пример:

```
Baz b=new Baz();
Bar br=b;
br.Test(); //At Foo //а если Test() в Baz пометить override, то будет выведено: At Baz

class Foo{
    public virtual Test(){WriteLine("At Foo");}
```

```
}

class Bar:Foo{}

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

В Bar своего Test() нет, а есть из Foo. Test() в Baz без override, поэтому она не подменит Test() из Foo.
Когда я говорю про подмену метода, я имею в виду замену в списке всех доступных методов, который составляется при каждом вызове виртуального метода. Т.е. при загрузке скрипта override не заменяет методы в его реальной структуре, а действует только при вызове virtual/override методов, и так же никаких постоянных замен методов не делает. Вот пример, демонстрирующий это:

```
Foo f=new Foo();
f.Test(); //At Foo

class Foo{
    public virtual Test(){WriteLine("At Foo");}
}

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

В этом примере Test() в Bar никак не влияет на Test() в Foo потому, что f ссылается на экземпляр Foo, в котором есть только один свой метод. Если вместо new Foo() написать new Bar(), то было бы два метода, и второй заменил бы первый. Ещё раз повторюсь: override влияет только на поиск метода при его вызове, а не меняет методы в коде (структуре программы).

Для закрепления посмотрим ещё пример, в котором видно, что override-метод может быть подменён другим override-методом:

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

class Foo{
    public virtual Test(){WriteLine("At Foo");}
}

class Bar:Foo{
    public override Test(){WriteLine("At Bar");}
}

class Baz:Bar{
```

```
    public override Test(){WriteLine("At Baz");}
}
```

Интерпретатор видит, что объект в переменной b это экземпляр класса Baz, в котором есть следующие методы (в обратном порядке):

```
public override Test(){WriteLine("At Baz");}
public override Test(){WriteLine("At Bar");}
public virtual Test(){WriteLine("At Foo");}
```

Интерпретатор идёт по списку сверху вниз. Первый метод с override, а значит интерпретатор заменяет им следующий за ним метод.

```
public override Test(){WriteLine("At Baz");}
public override Test(){WriteLine("At Baz");}
public virtual Test(){WriteLine("At Foo");}
```

Теперь второй метод, который был подменён первым. Он с override, и происходит то же самое, что на предыдущем шаге.

```
public override Test(){WriteLine("At Baz");}
public override Test(){WriteLine("At Baz");}
public override Test(){WriteLine("At Baz");}
```

Теперь все три метода - это один и тот же метод из Baz. Вызывается первый в списке. И так при всех вызовах: b.Test(), (Bar\b).Test(), (Foo\b).Test().
Выше я писал про отбрасывание методов из классов, которые не относятся к типу, с которым делается вызов.

```
Baz b=new Baz();
b.Test(); //это вызов с типом Baz
(Bar\b).Test(); //вызов с типом Bar
(Foo\b).Test(); //вызов с типом Foo
```

Так вот: когда интерпретатор проходит по списку и находит родной метод, то отбрасывает все перед ним (если не находит, то не отбрасывает). Родной метод - это метод класса (или из унаследованных), в контексте которого делается вызов (вызов с типом).
В примере выше ничего не отбрасывается. Когда вызывается b.Test(), то все три метода - родные (они все из Baz, ну т.е. это один и тот же метод, который подменил собой остальные). Когда (Bar\b).Test() и (Foo\b).Test(), то все три неродные, но т.к. родного нет, то отбрасывания не происходит (значит родной подменён через override).
Ещё хочу сказать про exclusive-методы. При загрузке скрипта они не копируются в производные классы.

```
Baz b=new Baz();
b.Test(); //At Baz
```

```
(Bar\b).Test(); //At Bar //это прямой, а не виртуальный вызов
(Foo\b).Test(); //At Baz

class Foo{
    public virtual Test(){WriteLine("At Foo");}
}

class Bar:Foo{
    public exclusive Test(){WriteLine("At Bar");}
}

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

В этом случае у экземпляра Baz есть только два метода - из Baz и Foo, а из Bar нет, т.к. он эксклюзивный. Если убрать модификатор exclusive, то при загрузке скрипта вылетит ошибка потому, что Test() в Baz помечен как override, а метод для подмены не помечен как virtual/override.

Как внутри интерпретатора устроен вызов обычных и виртуальных методов

Обычный (прямой) вызов метода - это вызов делегата, в котором есть ссылка на функцию, которую нужно выполнить. Эта функция находится во время загрузки скрипта. Виртуальный вызов - это тоже вызов делегата, но в котором нет ссылки на функцию, а каждый раз производится её поиск с учётом типа переменной, в которой находится объект (экземпляр). Поиск занимает время, поэтому виртуальные вызовы медленные. Для ускорения вызова virtual/override методов делается кэширование результатов поиска.
