**Полиморфизм, переопределение методов, абстрактные классы**

Полиморфизм является третьим фундаментальным свойством объектно-ориентированного программирования и предполагает способность к изменению функционала, унаследованного от базового класса.

**Полиморфизм предоставляет подклассу способ определения собственной версии метода**, определенного в его базовом классе, с использованием процесса, который называется *переопределением метода (method overriding)*. Чтобы пересмотреть текущий дизайн, нужно понять значение ключевых слов virtual и override.

***Виртуальным*** называется такой метод, который объявляется как **virtual** в базовом классе. Виртуальный метод отличается тем, что он может быть переопределен в одном или нескольких производных классах. Следовательно, у каждого производного класса может быть свой вариант виртуального метода. Кроме того, виртуальные методы интересны тем, что именно происходит при их вызове по ссылке на базовый класс. В этом случае средствами языка C# определяется именно тот вариант виртуального метода, который следует вызывать, исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения. Поэтому при ссылке на разные типы объектов выполняются разные варианты виртуального метода. Иными словами, вариант выполняемого виртуального метода выбирается по типу объекта, а не по типу ссылки на этот объект.

Так, если базовый класс содержит виртуальный метод и от него получены производные классы, то при обращении к разным типам объектов по ссылке на базовый класс выполняются разные варианты этого виртуального метода.

Метод объявляется как виртуальный в базовом классе с помощью ключевого слова virtual, указываемого перед его именем. Когда же виртуальный метод переопределяется в производном классе, то для этого используется модификатор **override**. А сам процесс повторного определения виртуального метода в производном классе называется переопределением метода. При переопределении метода - имя, возвращаемый тип и сигнатура переопределяющего метода должны быть точно такими же, как и у того виртуального метода, который переопределяется. Кроме того, виртуальный метод не может быть объявлен как static или abstract.

Переопределение метода служит основанием для воплощения одного из самых эффективных в C# принципов: ***динамической диспетчеризации методов***, которая представляет собой механизм разрешения вызова во время выполнения, а не компиляции. Значение динамической диспетчеризации методов состоит в том, что именно благодаря ей в C# реализуется динамический полиморфизм.

Если при наличии многоуровневой иерархии виртуальный метод не переопределяется в производном классе, то выполняется ближайший его вариант, обнаруживаемый вверх по иерархии.

И еще одно замечание: свойства также подлежат модификации ключевым словом virtual и переопределению ключевым словом override. Это же относится и к индексаторам.

**Полиморфизм предполагает** определение полиморфного интерфейса в базовом классе - набор членов класса, которые могут быть переопределены в классе-наследнике. Методы, которые мы хотим сделать доступными для переопределения, в базовом классе помечается модификатором virtual. Такие методы называют виртуальными. Они и представляют полиморфный интерфейс (также частью полиморфного интерфейса могут быть абстрактные члены класса).

При определении класса-наследника и наследовании методов базового класса мы можем выбрать одну из следующих стратегий:

* Обычное наследование всех членов базового класса в классе-наследнике
* Переопределение членов базового класса в классе-наследнике
* Скрытие членов базового класса в классе-наследнике

Первая стратегия довольно проста. Допустим, есть следующая пара классов Person и Employee:

**class Person**

**{**

**public string FirstName { get; set; }**

**public string LastName { get; set; }**

**public Person(string lName, string fName)//***кон-р*

**{**

**FirstName = fName;**

**LastName = lName;**

**}**

**public virtual void Display()**

**{**

**Console.WriteLine(FirstName + " " + LastName);**

**}**

**}**

**class Employee : Person**

**{**

**public string Company { get; set; }**

**public Employee(string lName, string fName, string comp) :base(fName, lName) //***кон-р*

**{**

**Company = comp;**

**}**

**}**

|  |  |
| --- | --- |
|  |  |

**В базовом классе Person метод Display() определен с модификаторами virtual, поэтому данный метод может быть переопределен.** Но класс Employee наследует его как есть:

**class Program**

**{**

**static void Main(string[] args)**

**{**

**Person p1 = new Person("Bill", "Gates");**

**p1.Display(); //** *вызов метода Display из класса Person*

**Person p2 = new Employee("Tom", "Johns", "UnitBank");**

**p2.Display(); //** *вызов метода Display из класса Person*

**Employee p3 = new Employee("Sam", "Toms", "CreditBank");**

**p3.Display(); //** *вызов метода Display из класса Person*

**Console.Read();**

**}**

**}**

|  |  |
| --- | --- |
|  |  |

Консольный вывод:

Bill Gates

Tom Johns

Sam Toms

**Вторая стратегия - переопределение методов базового класса в классе-наследнике предполагает использование ключевого слова override:**

**class Employee : Person**

**{**

**public string Company { get; set; }**

**public Employee(string lName, string fName, string comp)**

**:base(fName, lName)**

**{**

**Company = comp;**

**}**

**public override void Display() //***переопределение метода*

**{**

**Console.WriteLine(FirstName + " " + LastName + " работает в компании "+ Company);**

**}**

**}**

|  |  |
| --- | --- |
|  |  |

Класс Person остается тем же, в нем так же метод Display объявляется как виртуальный. В этом случае поведение объекта Employee изменится:

**class Program**

**{**

**static void Main(string[] args)**

**{**

**Person p1 = new Person("Bill", "Gates");**

**p1.Display(); //** *вызов метода Display из класса Person*

**Person p2 = new Employee("Tom", "Johns", "UnitBank");**

**p2.Display(); //** *вызов метода Display из класса Employee*

**Employee p3 = new Employee("Sam", "Toms", "CreditBank");**

**p3.Display(); //** *вызов метода Display из класса Employee*

**}**

**}**

Консольный вывод:

**Bill Gates**

**Tom Johns работает в компании UnitBank**

**Sam Toms работает в компании CreditBank**

При третьей стратегии можно просто определить в классе-наследнике метод с тем же именем, без переопределения с помощью слова override:

**class Employee : Person**

**{**

**public string Company { get; set; }**

**public Employee(string lName, string fName, string comp)**

**:base(fName, lName)**

**{**

**Company = comp;**

**}**

**public new void Display()**

**{**

**Console.WriteLine(FirstName + " " + LastName + " работает в компании "+ Company);**

**}**

**}**

В этом случае метод Display() в Employee скрывает метод Display() из класса Person. Чтобы явно скрыть метод из базового класса, используется ключевое слово new, хотя в принципе оно необязательно, по умолчанию система это делает неявно.

Использование в программе:

**Person p1 = new Person("Bill", "Gates");**

**p1.Display(); // вызов метода Display из класса Person**

**Person p2 = new Employee("Tom", "Johns", "UnitBank");**

**p2.Display(); // вызов метода Display из класса Person**

**Employee p3 = new Employee("Sam", "Toms", "CreditBank");**

**p3.Display(); // вызов метода Display из класса Employee**

|  |  |
| --- | --- |
|  |  |

Консольный вывод:

Bill Gates

Tom Johns

Sam Toms работает в компании CreditBank

Обратите внимание на различия во втором случае (p2.Display()) при разных стратегиях.

**Ключевое слово base**

Кроме конструкторов, мы можем обратиться с помощью ключевого слова base к другим членам базового класса. В нашем случае вызов base.Display(); будет обращением к методу Display() в классе Person:

**class Employee : Person**

**{**

**public string Company { get; set; }**

**public Employee(string lName, string fName, string comp)**

**:base(fName, lName)**

**{**

**Company = comp;**

**}**

**public override void Display()**

**{**

**base.Display();**

**Console.WriteLine("Место работы : " + Company);**

**}**

**}**

**Запрет переопределения методов**

Также можно запретить переопределение методов и свойств. В этом случае их надо объявлять с модификатором sealed:

**class Employee : Person**

**{**

**public string Company { get; set; }**

**public Employee(string lName, string fName, string comp)**

**:base(fName, lName)**

**{**

**Company = comp;**

**}**

**public override sealed void Display()**

**{**

**base.Display();**

**Console.WriteLine("Место работы : " + Company);**

**}**

**}**

|  |  |
| --- | --- |
|  |  |

При создании методов с модификатором sealed надо учитывать, что sealed применяется в паре с override, то есть только в переопределяемых методах.

И в этом случае мы не сможем переопределить метод Display в классе, унаследованном от Employee.

**Абстрактные классы**

Кроме обычных классов в C# есть абстрактные классы. Абстрактный класс похож на обычный класс. Он также может иметь переменные, методы, конструкторы, свойства. Но мы не можем создать объект или экземпляр абстрактного класса. Абстрактные классы лишь предоставляют базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

При определении абстрактных классов используется ключевое слово abstract:

**abstract class Human**

**{**

**public int Length { get; set; }**

**public double Weight { get; set; }**

**}**

**abstract class Human**

**{**

**public int Length { get; set; }**

**public double Weight { get; set; }**

**}**

|  |  |
| --- | --- |
|  |  |

**Кроме обычных методов абстрактный класс может иметь абстрактные методы**. Подобные методы определяются с помощью ключевого слова abstract и не имеют никакого функционала:

**public abstract void Display();**

|  |  |
| --- | --- |
|  |  |

При этом производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе. При переопределении в производном классе такой метод также объявляется с модификатором override. Также следует учесть, что если класс имеет хотя бы одно абстрактное свойство или метод, то он должен быть определен как абстрактный.

Абстрактные методы также, как и виртуальные, являются частью полиморфного интерфейса. Но если в случае с виртуальными методами мы говорим, что класс-наследник наследует реализацию, то в случае с абстрактными методами наследуется интерфейс, представленный этими абстрактными методами.

Зачем нужны абстрактные классы? Допустим, в нашей программе для банковского сектора мы можем определить три класса: Person, который описывает человека, Employee, который описывает сотрудника банка, и класс Client, который будет представлять клиента банка. Очевидно, что классы Employee и Client будут производными от класса Person. И так как все объекты будут представлять либо сотрудника банка, либо клиента, то напрямую мы от класса Person создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным.

**abstract class Person**

**{**

**public string FirstName { get;  set; }**

**public string LastName {get; set; }**

**public Person(string lName, string fName)**

**{**

**FirstName = fName;**

**LastName = lName;**

**}**

**public abstract void Display();**

**}**

**class Client : Person**

**{**

**public string Bank { get; set; }**

**public Client(string lName, string fName, string comp)**

**: base(fName, lName)**

**{**

**Bank = comp;**

**}**

**public override void Display()**

**{**

**Console.WriteLine(FirstName + " " + LastName + " имеет счет в банке " + Bank);**

**}**

**}**

|  |  |
| --- | --- |
|  |  |

Другим хрестоматийным примером является система геометрических фигур. В реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами:

**// абстрактный класс фигуры**

**abstract class Figure**

**{**

**float x; //** *x-координата точки*

**float y; //** *y-координата точки*

**public Figure(float x, float y)**

**{**

**this.x=x;**

**this.y=y;**

**}**

*// абстрактный метод для получения периметра*

**public abstract float Perimeter();**

*// абстрактный метод для получения площади*

**public abstract float Area();**

**}**

**//** *производный класс прямоугольника*

**class Rectangle : Figure**

**{**

**public float Width { get; set; }**

**public float Height { get; set; }**

*// конструктор с обращением к конструктору класса Figure*

**public Rectangle(float x, float y, float width, float height)**

**: base(x, y)**

**{**

**this.Width = width;**

**this.Height = height;**

**}**

*// переопределение получения периметра*

**public override float Perimeter()**

**{**

**return Width \* 2 + Height \* 2;**

**}**

*// переопрелеление получения площади*

**public override float Area()**

**{**

**return Width \* Height;**

**}**

**}**

Абстрактный класс может содержать и реализованные методы.

Пример

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Abstrsct\_01

{

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication1

{

// Создаем абстрактный класс

abstract class UserInfo

{

protected string Name;

protected byte Age;

public UserInfo(string Name, byte Age)

{

this.Name = Name;

this.Age = Age;

}

public int f1() { return this.Name.Length; } //метод с реализацией

//расчет длины имени один и тот же для любой конкретизации

// Абстрактный метод

public abstract string ui();

}

class UserFamily : UserInfo

{

string Family;

public UserFamily(string Family, string Name, byte Age)

: base(Name, Age)

{

this.Family = Family;

}

// Переопределяем метод ui

public override string ui()

{

return Family + " " + Name + " " + Age;

}

}

class Program

{

static void Main(string[] args)

{

UserFamily user1 = new UserFamily("Иванов", "Павел", 26);

Console.WriteLine(user1.f1());

Console.WriteLine(user1.ui());

Console.ReadLine();

}

}

}

}