**ГЛАВА 11. ПОЛИМОРФИЗМ**

**Оглавление**

[§11.1 Виртуальные члены класса 1](#_Toc61461500)

[§11.2 Абстрактные классы 3](#_Toc61461501)

[§11.3 Понятие полиморфизма 7](#_Toc61461502)

# §11.1 Виртуальные члены класса

При наследовании нередко возникает необходимость изменить в классе-наследнике функционал метода, который был унаследован от базового класса. В этом случае класс-наследник может переопределять методы и свойства базового класса.

Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечается модификатором **virtual**. Такие методы и свойства называют виртуальными.

А чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором **override**. Переопределенный метод в классе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

Например, рассмотрим следующие классы:

**Листинг 11.1**

|  |  |
| --- | --- |
| 1 | class Person |
| 2 | { |
| 3 | public string Name { get; set; } |
| 4 | public Person(string name) |
| 5 | { |
| 6 | Name = name; |
| 7 | } |
| 8 | public virtual void Display() |
| 9 | { |
| 10 | Console.WriteLine(Name); |
| 11 | } |
| 12 | } |
| 13 | class Employee : Person |
| 14 | { |
| 15 | public string Company { get; set; } |
| 16 | public Employee(string name, string company) : base(name) |
| 17 | { |
| 18 | Company = company; |
| 19 | } |
| 20 | } |

Здесь класс Person представляет человека. Класс Employee наследуется от Person и представляет сотрудника предприятия. Этот класс кроме унаследованного свойства Name имеет еще одно свойство - Company.

Чтобы сделать метод Display доступным для переопределения, этот метод определен с модификатором virtual. Поэтому мы можем переопределить этот метод, но можем и не переопределять. Допустим, нас устраивает реализация метода из базового класса. В этом случае объекты Employee будут использовать реализацию метода Display из класса Person:

**Листинг 11.2**

|  |  |
| --- | --- |
| 1 | static void Main(string[] args) |
| 2 | { |
| 3 | Person p1 = new Person("Bill"); |
| 4 | p1.Display(); // вызов метода Display из класса Person |
| 5 | Employee p2 = new Employee("Tom", "Microsoft"); |
| 6 | p2.Display(); // вызов метода Display из класса Person |
| 7 | Console.ReadKey(); |
| 8 | } |

При переопределении виртуальных методов следует учитывать ряд ограничений:

1. Виртуальный и переопределенный методы должны иметь один и тот же модификатор доступа. То есть если виртуальный метод определен с помощью модификатора public, то и переопредленный метод также должен иметь модификатор public.
2. Нельзя переопределить или объявить виртуальным статический метод.

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:

**Листинг 11.3**

|  |  |
| --- | --- |
| 1 | class Employee : Person |
| 2 | { |
| 3 | public string Company { get; set; } |
| 4 |  |
| 5 | public Employee(string name, string company) |
| 6 | :base(name) |
| 7 | { |
| 8 | Company = company; |
| 9 | } |
| 10 |  |
| 11 | public override void Display() |
| 12 | { |
| 13 | base.Display(); |
| 14 | Console.WriteLine($"работает в {Company}"); |
| 15 | } |
| 16 | } |

# §11.2 Абстрактные классы

Кроме обычных классов в C# есть абстрактные классы. Абстрактный класс похож на обычный класс. Он также может иметь переменные, методы, конструкторы, свойства. Единственное, что при определении абстрактных классов используется ключевое слово abstract:

**Листинг 11.4**

|  |  |
| --- | --- |
| 1 | **abstract** class Human |
| 2 | { |
| 3 | public int Length { get; set; } |
| 4 | public double Weight { get; set; } |
| 5 | } |

Но главное **отличие** состоит в том, что мы не можем использовать конструктор абстрактного класса для создания его объекта.

Зачем нужны абстрактные классы? Допустим, в нашей программе для банковского сектора мы можем определить две основных сущности: клиента банка и сотрудника банка. Каждая из этих сущностей будет отличаться, например, для сотрудника надо определить его должность, а для клиента - сумму на счете. Соответственно клиент и сотрудник будут составлять отдельные классы Client и Employee. В то же время обе этих сущности могут иметь что-то общее, например, имя и фамилию, какую-то другую общую функциональность. И эту общую функциональность лучше вынести в какой-то отдельный класс, например, Person, который описывает человека. То есть классы Employee (сотрудник) и Client (клиент банка) будут производными от класса Person. И так как все объекты в нашей системе будут представлять либо сотрудника банка, либо клиента, то напрямую мы от класса Person создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным:

**Листинг 11.5**

|  |  |
| --- | --- |
| 1 | abstract class Person |
| 2 | { |
| 3 | public string Name { get; set; } |
| 4 | public Person(string name) |
| 5 | { |
| 6 | Name = name; |
| 7 | } |
| 8 | public void Display() |
| 9 | { |
| 10 | Console.WriteLine(Name); |
| 11 | } |
| 12 | } |
| 13 | class Client : Person |
| 14 | { |
| 15 | public int Sum { get; set; }    // сумма на счету |
| 16 | public Client(string name, int sum) : base(name) |
| 17 | { |
| 18 | Sum = sum; |
| 19 | } |
| 20 | } |
| 21 | class Employee : Person |
| 22 | { |
| 23 | public string Position { get; set; } // должность |
| 24 | public Employee(string name, string position) : base(name) |
| 25 | { |
| 26 | Position = position; |
| 27 | } |
| 28 | } |

Затем мы сможем использовать эти классы:

**Листинг 11.6**

|  |  |
| --- | --- |
| 1 | Client client = new Client("Tom", 500) |
| 2 | Employee employee = new Employee ("Bob", "Apple") |
| 3 | client.Display() |
| 4 | employee.Display() |

Или даже так:

**Листинг 11.7**

|  |  |
| --- | --- |
| 1 | Person client = new Client("Tom", 500) |
| 2 | Person employee = new Employee ("Bob", "Операционист") |

Кроме обычных свойств и методов абстрактный класс может иметь абстрактные члены классов (свойства, методы, события), которые определяются с помощью ключевого слова abstract и не имеют никакого функционала.

Абстрактные члены классов не должны иметь модификатор private. 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 или свойство также объявляются с модификатором override. Также следует учесть, что если класс имеет хотя бы один абстрактный метод (или абстрактные свойство, событие), то этот класс должен быть определен как абстрактный.

Например, сделаем в примере выше метод Display абстрактным:

**Листинг 11.8**

|  |  |
| --- | --- |
| 1 | abstract class Person |
| 2 | { |
| 3 | public string Name { get; set; } |
| 4 | public Person(string name) |
| 5 | { |
| 6 | Name = name; |
| 7 | } |
| 8 | public abstract void **Display**(); |
| 9 | } |
| 10 | class Client : Person |
| 11 | { |
| 12 | public int Sum { get; set; } // сумма на счету |
| 13 | public Client(string name, int sum) : base(name) |
| 14 | { |
| 15 | Sum = sum; |
| 16 | } |
| 17 | public override void **Display**() |
| 18 | { |
| 19 | Console.WriteLine($"{Name} имеет счет на сумму {Sum}"); |
| 20 | } |
| 21 | } |
| 22 | class Employee : Person |
| 23 | { |
| 24 | public string Position { get; set; } // должность |
| 25 | public Employee(string name, string position) : base(name) |
| 26 | { |
| 27 | Position = position; |
| 28 | } |
| 29 | public override void **Display**() |
| 30 | { |
| 31 | Console.WriteLine($"{Position} {Name}"); |
| 32 | } |
| 33 | } |

Следует отметить использование **абстрактных свойств**. Их определение похоже на определение автосвойств. Например:

**Листинг 11.9**

|  |  |
| --- | --- |
| 1 | abstract class Person |
| 2 | { |
| 3 | public abstract string Name { get; set; } |
| 4 | } |
| 5 | class Client : Person |
| 6 | { |
| 7 | private string name; |
| 8 | public override string Name |
| 9 | { |
| 10 | get { return "Mr/Ms. " + name; } |
| 11 | set { name = value; } |
| 12 | } |
| 13 | } |

# §11.3 Понятие полиморфизма

Полиморфизм часто называется третьим столпом объектно-ориентированного программирования после инкапсуляции и наследования. Полиморфизм — слово греческого происхождения, означающее "многообразие форм" и имеющее **несколько аспектов**.

1. Во время выполнения объекты производного класса могут обрабатываться как объекты базового класса в таких местах, как параметры метода и коллекции или массивы. Когда возникает полиморфизм, объявленный тип объекта перестает соответствовать своему типу во время выполнения.
2. Базовые классы могут определять и реализовывать виртуальные методы, а производные классы — переопределять их, т. е. предоставлять свое собственное определение и реализацию. Во время выполнения, когда клиент вызывает метод, общеязыковая исполняющая среда (CLR) выполняет поиск типа объекта во время выполнения и вызывает перезапись виртуального метода. В исходном коде можно вызвать метод в базовом классе и обеспечить выполнение версии метода, относящейся к производному классу.

Виртуальные методы позволяют работать с группами связанных объектов универсальным способом. Представим, например, приложение, позволяющее пользователю создавать различные виды фигур на поверхности для рисования. Во время компиляции вы еще не знаете, какие именно виды фигур создаст пользователь. При этом приложению необходимо отслеживать все различные типы создаваемых фигур и обновлять их в ответ на движения мыши. Для решения этой проблемы можно использовать полиморфизм, выполнив два основных действия.

1. Создать иерархию классов, в которой каждый отдельный класс фигур является производным из общего базового класса.
2. Применить виртуальный метод для вызова соответствующего метода на любой производный класс через единый вызов в метод базового класса.

Для начала создадим базовый класс с именем Shape и производные классы, например, Rectangle, Circle и Triangle. Присвоим классу Shape виртуальный метод с именем Draw и переопределите его в каждом производном классе для рисования конкретной фигуры, которую этот класс представляет. Создадим коллекцию фигур List<Shape>, добавив в него Circle, Triangle и Rectangle.

**Листинг 11.10**

|  |  |
| --- | --- |
| 1 | public class Shape |
| 2 | { |
| 3 | // Свойства абстрактного класса |
| 4 | public int X { get; private set; } |
| 5 | public int Y { get; private set; } |
| 6 | public int Height { get; set; } |
| 7 | public int Width { get; set; } |
| 8 | // Виртуальный метод |
| 9 | public virtual void Draw() |
| 10 | { |
| 11 | Console.WriteLine("Performing base class drawing tasks"); |
| 12 | } |
| 13 | } |
| 14 | public class Circle : Shape |
| 15 | { |
| 16 | public override void Draw() |
| 17 | { |
| 18 | // Код для рисования окружности |
| 19 | Console.WriteLine("Drawing a circle"); |
| 20 | base.Draw(); |
| 21 | } |
| 22 | } |
| 23 | public class Rectangle : Shape |
| 24 | { |
| 25 | public override void Draw() |
| 26 | { |
| 27 | // Код для рисования прямоугольника |
| 28 | Console.WriteLine("Drawing a rectangle"); |
| 29 | base.Draw(); |
| 30 | } |
| 31 | } |
| 32 | public class Triangle : Shape |
| 33 | { |
| 34 | public override void Draw() |
| 35 | { |
| 36 | // Код для рисования треугольника |
| 37 | Console.WriteLine("Drawing a triangle"); |
| 38 | base.Draw(); |
| 39 | } |
| 40 | } |

Для обновления поверхности рисования используйте цикл foreach, чтобы выполнить итерацию списка и вызвать метод Draw на каждом объекте Shape в списке. Несмотря на то, что каждый объект в списке имеет объявленный тип Shape, будет вызван тип времени выполнения (переопределенная версия метода в каждом производном классе).

**Листинг 11.11**

|  |  |
| --- | --- |
| 1 | var shapes = new List<Shape> |
| 2 | { |
| 3 | new Rectangle(), |
| 4 | new Triangle(), |
| 5 | new Circle() |
| 6 | }; |
| 7 | foreach (var shape in shapes) |
| 8 | { |
| 9 | shape.Draw(); |
| 10 | } |

Здесь мы видим два проявления принципа полиморфизма:

1) Классы Rectangle, Triangle и Circle можно использовать везде, где ожидается класс Shape. Приведение не требуется, поскольку существует неявное преобразование производного класса в его базовый класс.

2) Виртуальный метод Draw вызывается для каждого из производных классов, а не для базового класса.