Лекция №11. Полиморфизм как принцип объектно-ориентированного программирования в C#

ОГЛАВЛЕНИЕ

§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.

Механизм виртуальных методов обеспечивает так называемое позднее связывание — вызов метода определяется в момент выполнения программы на основе фактического типа объекта, а не типа ссылки. Это позволяет работать с объектами различных производных классов единообразно через

ссылку на базовый класс, обеспечивая гибкость и расширяемость кода. Например, рассмотрим следующие классы:

```
 1. class Person
 2. {
 3.     public string Name { get; set; }
 4.     public Person(string name)
 5.     {
 6.         Name = name;
 7.     }
 8.     public virtual void Print()
 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)
17.         : base(name)
18.     {
19.         Company = company;
20.     }
21. }
22.
```

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**:

```
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:

- 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**.
- 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**:

```
 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**:

```
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.

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:

```
abstract class Person
{
    public string Name { get; set; }
    public Person(string name)
    {
        Name = name;
    }
    public void Display()
    {
        Console.WriteLine(Name);
    }
}
class Client : Person
{
    public int Sum { get; set; } // сумма на счету
    public Client(string name, int sum) : base(name)
    {
        Sum = sum;
    }
}
class Employee : Person
{
    public string Position { get; set; } // должность
    public Employee(string name, string position)
        : base(name)
    {
        Position = position;
    }
}
```

Затем мы сможем использовать эти классы:

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

Или даже так:

```
1. Person client = new Client("Tom", 500)
2. Person employee = new Employee ("Bob", "Сотрудник")
```

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**. Так же, если хоть одно свойство или метод абстрактные – то класс должен быть абстрактным.

Например, сделаем в примере выше метод *Display* абстрактным:

```
 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($"У {Name} - {Sum}");
20.     }
21. }
```

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

```
 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 Понятия 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, добавив в него Circle, Triangle и Rectangle.

Ниже представлен класс Shape, который является базовым для иерархии фигур:

```
 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 class drawing");
12.     }
13. }
```

Далее необходимо определить три класса: класс окружности, класс прямоугольника и класс треугольника.

```
public class Circle : Shape
{
    public override void Draw()
    {
        // Код для рисования окружности
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
public class Rectangle : Shape
{
    public override void Draw()
    {
        // Код для рисования прямоугольника
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
public class Triangle : Shape
{
    public override void Draw()
    {
        // Код для рисования треугольника
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}
```

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).

```
var shapes = new List<Shape>
{
    new Rectangle(),
    new Triangle(),
    new Circle()
};
foreach (var shape in shapes)
{
    shape.Draw();
}
```

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.

§11.3.1 Разновидности полиморфизма. Статический полиморфизм

Статический полиморфизм, который реализуется на этапе компиляции и не требует механизма наследования, в C# чаще всего проявляется в двух формах: **перегрузка методов** и **обобщённые** (*generic*) **типы**. Перегрузка методов позволяет определять несколько методов с одинаковым именем, но различающихся по количеству, типу или порядку параметров. Компилятор автоматически выбирает подходящую версию метода в зависимости от переданных аргументов.

Рассмотрим пример:

```
 1. class Calculator
 2. {
 3.     public int Add(int a, int b)
 4.     {
 5.         return a + b;
 6.     }
 7.
 8.     public double Add(double a, double b)
 9.     {
10.         return a + b;
11.     }
12.
13.     public int Add(int a, int b, int c)
14.     {
15.         return a + b + c;
16.     }
17. }
```

Здесь метод *Add* перегружен тремя способами: для двух целых чисел, для двух дробных и для трёх целых. Вызов конкретной версии определяется ещё до выполнения программы — на этапе компиляции. Это и есть признак статического полиморфизма.

Ещё одним важным механизмом статического полиморфизма являются обобщённые типы. Они позволяют писать универсальные методы и классы, которые работают с различными типами данных, не теряя типобезопасности. Например:

```
1. public T Max<T>(T a, T b) where T : IComparable<T>
2. {
3.     return a.CompareTo(b) > 0 ? a : b;
4. }
```

Здесь метод *Max* может работать с любыми типами, реализующими интерфейс **IComparable<T>**, — будь то **int**, **double** или **строка**. Конкретная реализация подставляется компилятором при вызове метода с указанием типа. Такой подход позволяет избежать дублирования кода и обеспечивает высокую производительность, поскольку разрешение происходит на этапе компиляции.

§11.3.2 Разновидности полиморфизма. Динамический полиморфизм

Если статический полиморфизм реализуется на этапе компиляции, то динамический полиморфизм проявляется во время выполнения программы. Он основан на механизмах наследования и переопределения методов и позволяет объектам производных классов приобретать собственное поведение при вызове одного и того же метода через ссылку на базовый класс. Именно эта форма полиморфизма считается «классической» в объектно-ориентированном программировании.

Ключевую роль в реализации динамического полиморфизма в C# играют виртуальные методы. Метод в базовом классе помечается модификатором **virtual**, что делает его доступным для переопределения в наследниках. В производном классе этот метод может быть переопределён с помощью модификатора **override**, при этом его сигнатура должна точно соответствовать сигнатуре виртуального метода.

```
1. class Animal
2. {
3.     public virtual void MakeSound()
4.     {
5.         Console.WriteLine("Животное издаёт звук");
6.     }
7. }
```

```
 1. class Dog : Animal
 2. {
 3.     public override void MakeSound()
 4.     {
 5.         Console.WriteLine("Гав-гав!");
 6.     }
 7. }
 8.
 9. class Cat : Animal
10. {
11.     public override void MakeSound()
12.     {
13.         Console.WriteLine("Мяу-мяу!");
14.     }
15. }
```

Теперь, если мы создадим массив ссылок на базовый тип **Animal** и поместим в него объекты **Dog** и **Cat**, вызов метода *MakeSound()* будет корректно определяться в зависимости от реального типа объекта:

```
 1. Animal[] animals = {
 2.     new Dog(),
 3.     new Cat(),
 4.     new Animal()
 5. };
 6.
 7. foreach (Animal animal in animals)
 8. {
 9.     animal.MakeSound();
10. }
```

На экран будет выведено:

```
Гав-гав!
Мяу-мяу!
Животное издаёт звук
```

Хотя все переменные имеют тип **Animal**, вызывается именно та реализация метода *MakeSound*, которая определена в конкретном классе. Это и есть позднее связывание — механизм, лежащий в основе динамического полиморфизма.

Важно понимать, что возможность такого поведения обеспечивается **виртуальной таблицей вызовов** (*vtable*), которую формирует среда выполнения **.NET**. При вызове виртуального метода **CLR** определяет, какую

именно реализацию следует выполнить, исходя из реального типа объекта в момент выполнения.

Динамический полиморфизм позволяет строить гибкие и расширяемые архитектуры, например, при реализации плагинов, стратегий или обработчиков событий. Он лежит в основе принципа «один интерфейс — множество реализаций», что делает код более модульным и удобным для сопровождения.

§11.4 Интерфейсы в C#

Интерфейсы – это еще один инструмент реализации полиморфизма в C#. Интерфейс представляет ссылочный тип, который может определять некоторый функционал - набор методов и свойств без реализации. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

Интерфейс может содержать только сигнатуры (имя и типы параметров) своих членов. Интерфейс не может содержать конструкторы, поля. Создавать объекты интерфейса невозможно.

Интерфейс – это единица уровня класса, он объявляется за пределами класса, при помощи ключевого слова **interface**:

```
1. interface ISomeInterface
2. {
3.  // тело интерфейса
4. }
```

Как правило, названия интерфейсов в **C#** начинаются с заглавной буквы **I**, например, **IComparable**, **IEnumerable** (так называемая *венгерская нотация*), однако это не обязательное требование, а больше стиль программирования.

Внутри **интерфейса** объявляются сигнатуры его членов, модификаторы доступа указывать не нужно:

```
1. interface ISomeInterface
2. {
3.    string SomeProperty { get; set; } // свойство
4.    void SomeMethod(int a); // метод
5. }
```

Чтобы указать, что класс реализует интерфейс, необходимо, так же, как и при наследовании, после имени класса и двоеточия указать имя интерфейса:

```
1. class SomeClass : ISomeInterface
2. {
3.     // тело класса
4. }
```

Класс, который реализует интерфейс, должен предоставить реализацию всех членов интерфейса:

```
 1. class SomeClass : ISomeInterface
 2. {
 3.     public string SomeProperty
 4.     {
 5.         get
 6.         {
 7.             // тело get аксессора
 8.         }
 9.         set
10.         {
11.             // тело set аксессора
12.         }
13.     }
14.     public void SomeMethod(int a)
15.     {
16.         // тело метода
17.     }
18. }
```

В целом интерфейсы могут определять следующие сущности:

- Методы.
- Свойства.
- Индексаторы.
- События.
- Статические поля и константы.

Однако интерфейсы не могут определять нестатические поля — это ограничение подчёркивает их назначение как контрактов, описывающих поведение, а не хранение состояния. Все члены интерфейса по умолчанию считаются открытыми и не могут иметь модификаторов доступа.

Начиная с C# 8.0, в интерфейсах появились реализации по умолчанию

для методов, что позволяет задавать поведение по умолчанию, не нарушая принципов абстракции. Например, простейший интерфейс, который определяет некоторые из этих компонентов:

```
 1. interface IMovable
 2. {
 3.    // константа
 4.    const int minSpeed = 0; // минимальная скорость
 5.    // статическая переменная
 6.    static int maxSpeed = 60; // максимальная скорость
 7.    // метод
 8.    void Move(); // движение
 9.    // свойство
10.     string Name { get; set; } // название
11.    // определение делегата для события
12.    delegate void MoveHandler(string message);
13.    // событие
14.    event MoveHandler MoveEvent; // событие движения
15. }
```

В данном случае определен интерфейс **IMovable**, который представляет некоторый движущийся объект. Данный интерфейс содержит различные компоненты, которые описывают возможности движущегося объекта. То есть интерфейс описывает некоторый функционал, который должен быть у движущегося объекта.

Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами и свойствами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не имеет реализации, не принимает никаких параметров и ничего не возвращает.

То же самое в данном случае касается свойства Name. На первый взгляд оно похоже на автоматическое свойство. Но в реальности это определение свойства в интерфейсе, которое не имеет реализации, а не автосвойство.

Еще один момент в объявлении интерфейса: если его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Это касается также и констант, и статических

переменных, которые в классах и структурах по умолчанию имееют модификатор **private**. В интерфейсах же они имеют по умолчанию модификатор **public**. И например, мы могли бы обратиться к константе **minSpeed** и переменной **maxSpeed** интерфейса **IMovable**:

```
1. static void Main(string[] args)
2. {
3.     Console.WriteLine(IMovable.maxSpeed);
4.     Console.WriteLine(IMovable.minSpeed);
5. }
```

Но также, начиная с версии C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса.

§11.4.1 Применение интерфейсов

Интерфейс представляет некое описание типа, набор компонентов, который должен иметь тип данных. И, собственно, мы не можем создавать объекты интерфейса напрямую с помощью конструктора, как например, в классах:

```
1. Movable m = new IMovable() // ✕ Ошибка, так нельзя
```

В конечном счете интерфейс предназначен для реализации в классах и структурах. Например, возьмем следующий интерфейс **IMovable**:

```
1. interface IMovable
2. {
3.     void Move();
4. }
```

Затем какой-нибудь класс или структура могут применить данный интерфейс:

```
1. // применение интерфейса в классе
2. class Person : IMovable
3. {
4.     public void Move()
5.     {
6.         Console.WriteLine("Человек идет");
7.     }
8. }
```

При применении интерфейса, как и при наследовании после имени класса или структуры указывается двоеточие и затем идут названия

применяемых интерфейсов. При этом класс должен реализовать все методы и свойства применяемых интерфейсов, если эти методы и свойства не имеют реализации по умолчанию.

Если методы и свойства интерфейса не имеют модификатора доступа, то по умолчанию они являются публичными, при реализации этих методов и свойств в классе и структуре к ним можно применять только модификатор **public**.

```
 1. using System;
 2. namespace HelloApp
 3. {
 4.     interface IMovable
 5.     {
 6.         void Move();
 7.     }
 8.     class Person : IMovable
 9.     {
10.         public void Move()
11.         {
12.             Console.WriteLine("Человек идет");
13.         }
14.     }
15.     class Program
16.     {
17.         static void Action(IMovable movable)
18.         {
19.             movable.Move();
20.         }
21.         static void Main(string[] args)
22.         {
23.             Person person = new Person();
24.             Action(person);
25.
26.           Console.Read();
27.         }
28.     }
29. }
```

В данной программе определен метод *Action()*, который в качестве параметра принимает объект интерфейса **IMovable**. На момент написания кода мы можем не знать, что это будет за объект - какой-то класс или структура. Единственное, в чем мы можем быть уверены, что этот объект

обязательно реализует метод *Move* и мы можем вызвать этот метод.

Иными словами, интерфейс — это контракт, что какой-то определенный тип обязательно реализует некоторый функционал.

§11.4.2 Множественное наследование интерфейсов

Одно из ключевых преимуществ интерфейсов в **C#** — возможность **множественного наследования**. В отличие от классов, где наследование только одиночное (класс может наследовать только от одного базового класса), один интерфейс может наследовать от нескольких других интерфейсов, а класс — реализовывать любое количество интерфейсов одновременно. Это позволяет гибко комбинировать поведение и создавать богатые контракты, не нарушая принципов объектно-ориентированного проектирования.

Например, рассмотрим два интерфейса — **IMovable** и **IDrawable**:

```
 1. interface IMovable
 2. {
 3.     void Move();
 4. }
 5.
 6. interface IDrawable
 7. {
 8.     void Draw();
 9. }
```

Теперь мы можем определить класс, который реализует оба интерфейса:

```
 1. class Sprite : IMovable, IDrawable
 2. {
 3.     public void Move()
 4.     {
 5.         Console.WriteLine("Спрайт перемещается");
 6.     }
 7.
 8.     public void Draw()
 9.     {
10.         Console.WriteLine("Спрайт отрисовывается");
11.     }
12. }
```

Здесь класс **Sprite** поддерживает оба поведения — перемещение и отрисовку. При этом переменная любого из этих интерфейсных типов может

ссылаться на объект **Sprite**, обеспечивая полиморфный доступ:

```
IMovable obj1 = new Sprite();
IDrawable obj2 = new Sprite();

obj1.Move();  // вызов через IMovable
obj2.Draw();  // вызов через IDrawable
```

Множественное наследование интерфейсов особенно полезно при проектировании сложных систем, где объекты должны реализовывать разнородные функции — например, быть одновременно сериализуемыми, сравнимыми и наблюдаемыми. При этом **C#** остаётся в рамках безопасной типизации и избегает проблем, характерных для множественного наследования классов, таких как «*алмазная проблема*», поскольку интерфейсы не содержат состояния, а только определяют контракт.

§11.4.3 Интерфейсы в преобразованиях типов

Интерфейсы играют важную роль в механизмах преобразования типов в C#. Поскольку интерфейс определяет контракт поведения, любой класс, реализующий этот интерфейс, может быть неявно преобразован к типу интерфейса. Это позволяет работать с объектами разных классов единообразно, если они поддерживают один и тот же интерфейс, что является проявлением полиморфизма.

Например, рассмотрим интерфейс **ISpeakable** и два класса, его реализующие:

```
interface ISpeakable
{
    void Speak();
}

class Dog : ISpeakable
{
    public void Speak() => Console.WriteLine("Гав!");
}

class Cat : ISpeakable
{
    public void Speak()=> Console.WriteLine("Мяу!");
}
```

Теперь мы можем создать массив типа **ISpeakable** и поместить в него объекты как **Dog**, так и **Cat**:

```
1. ISpeakable[] animals = { new Dog(), new Cat() };
2.
3. foreach (ISpeakable animal in animals)
4. {
5.     animal.Speak();
6. }
```

Здесь происходит неявное преобразование типов: объекты производных классов автоматически приводятся к типу **ISpeakable**. Это безопасно, поскольку компилятор гарантирует, что все элементы массива реализуют метод *Speak()*.

Однако возможны и явные преобразования — когда требуется получить доступ к членам конкретного класса через ссылку на интерфейс. В таком случае используется оператор приведения:

```
1. ISpeakable animal = new Dog();
2. Dog dog = (Dog)animal;
3. dog.Speak();
```

Если же тип несовместим, будет выброшено исключение **InvalidCastException**. Чтобы избежать ошибок, рекомендуется использовать оператор **is** или **as**:

```
 1. if (animal is Dog dog)
 2. {
 3.     dog.Speak();
 4. }
 5. // или другая версия
 6. Dog dog = animal as Dog;
 7. if (dog != null)
 8. {
 9.     dog.Speak();
10. }
```

§11.5 Отношения между классами и объектами

Наследование является базовым принципом **ООП** и позволяет одному классу (*наследнику*) унаследовать функционал другого класса (*родительского*). Нередко отношения наследования еще называют *генерализацией* или *обобщением*. Наследование определяет отношение **IS A**,

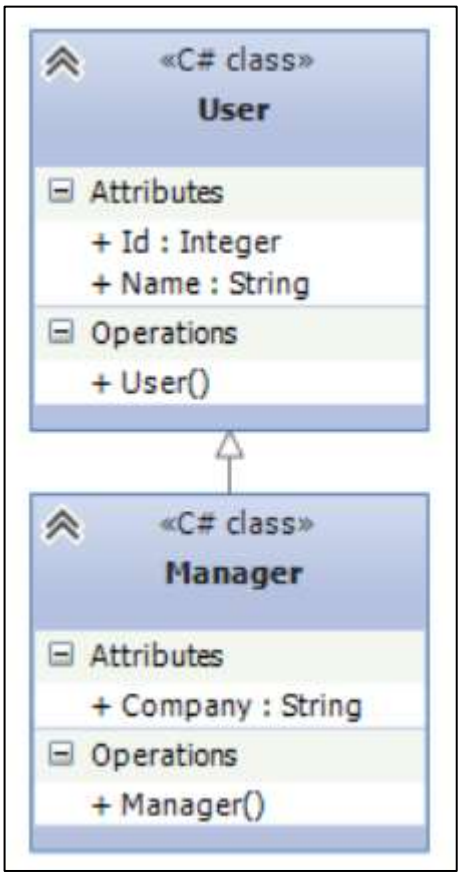
то есть «*является*».

Например:

```
1. class User
2. {
3.     public int Id { get; set; }
4.     public string Name { get; set; }
5. }
6. class Manager : User
7. {
8.     public string Company{ get; set; }
9. }
```

В данном случае используется наследование, а объекты класса **Manager** также являются и объектами класса **User**.

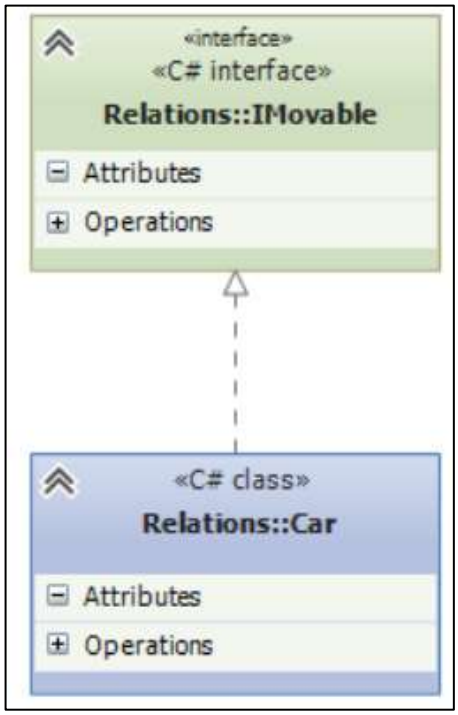
С помощью диаграмм **UML** (*диаграмма классов*) отношение между классами выражается в не закрашенной стрелочке от класса-наследника к классу-родителю:



Реализация предполагает определение интерфейса и его реализация в классах. Например, имеется интерфейс **IMovable** с методом *Move*, который реализуется в классе **Car**:

```
 1. public interface IMovable
 2. {
 3.     void Move();
 4. }
 5. public class Car : IMovable
 6. {
 7.     public void Move()
 8.     {
 9.         Console.WriteLine("Машина едет");
10.     }
11. }
```

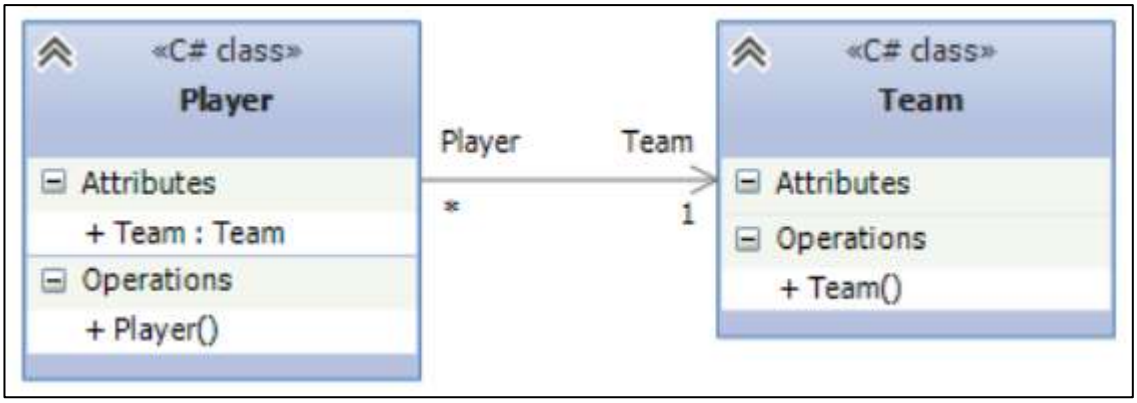
С помощью диаграмм **UML** отношение реализации также выражается в не закрашенной стрелочке от класса к интерфейсу, только линия теперь пунктирная:



Ассоциация — это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Например, объект одного типа содержит или использует объект другого типа. Например, игрок играет в определенной команде:

```
1. class Team
2. { }
3. class Player
4. {
5.     public Team Team { get; set; }
6. }
```

Класс **Player** связан отношением ассоциации с классом **Team**. На схемах **UML** ассоциация обозначается в виде обычно стрелки:



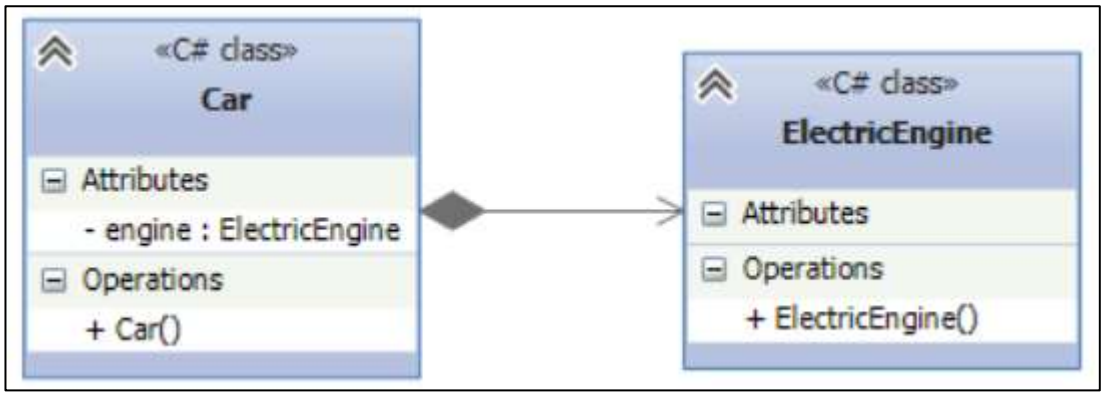
Нередко при отношении **ассоциации** указывается кратность связей. В данном случае единица у **Team** и звездочка у **Player** на диаграмме отражает связь «**один ко многим**». То есть одна команда будет соответствовать многим игрокам.

Композиция определяет отношение **HAS A**, то есть отношение «*имеет*». Например, в класс автомобиля содержит объект класса электрического двигателя:

```
 1. public class ElectricEngine
 2. { }
 3. public class Car
 4. {
 5.     ElectricEngine engine;
 6.     public Car()
 7.     {
 8.         engine = new ElectricEngine();
 9.     }
10. }
```

При этом класс автомобиля полностью управляет жизненным циклом объекта двигателя. При уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен и объект двигателя. И в этом плане объект автомобиля является главным, а объект двигателя – зависимой.

На диаграммах **UML** отношение **композиции** проявляется в обычной стрелке от главной сущности к зависимой, при этом со стороны главной сущности, которая содержит, объект второй сущности, располагается закрашенный ромбик:



От **композиции** следует отличать **агрегацию**. Она также предполагает отношение **HAS A**, но реализуется она иначе:

```
 1. public abstract class Engine
 2. { }
 3. public class Car
 4. {
 5.     Engine engine;
 6.     public Car(Engine eng)
 7.     {
 8.         engine = eng;
 9.     }
10. }
```

При **агрегации** реализуется **слабая связь**, то есть в данном случае объекты **Car** и **Engine** будут равноправны. В конструктор **Car** передается ссылка на уже имеющийся объект **Engine**. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс, что увеличивает гибкость программы.

Отношение **агрегации** на диаграммах **UML** отображается также, как и отношение композиции, только теперь ромбик будет не закрашенным: