11. ПОЛИМОРФИЗМ

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.

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 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:

```
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:

```
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, 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:

```
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:

```
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", "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:

```
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:

```
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`.

```
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).

```
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.

§11.4 Определение и применение интерфейсов

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**:

```
interface ISomeInterface
{
   // тело интерфейса
}
```

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:

```
class SomeClass : ISomeInterface
{
   // тело класса
}
```

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

```
class SomeClass : ISomeInterface
{
   public string SomeProperty
   {
     get
     {
       // тело get аксессора
     }
```

```
    set
    {
      // тело set аксессора
    }
  }
  public void SomeMethod(int a)
  {
    // тело метода
  }
}
```

В целом интерфейсы могут определять следующие сущности:

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

Однако интерфейсы не могут определять нестатические переменные. Например, простейший интерфейс, который определяет все эти компоненты:

Листинг 11.12

```
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 delegate void MoveHandler(string message);   // определение
   делегата для события
12 // событие
13 event MoveHandler MoveEvent;    // событие движения
14 }
```

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`:

Листинг 11.13

```
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.

§11.5 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  IMovable m = new IMovable()
                                  // ! Ошибка, так сделать 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:

Листинг 11.14

```
1  // применение интерфейса в классе
2  class Person : IMovable
3  {
4      public void Move()
5      {
6          Console.WriteLine("Человек идет");
7      }
8  }
9  // применение интерфейса в структуре
10 struct Car : IMovable
11 {
12     public void Move()
13     {
14         Console.WriteLine("Машина едет");
15     }
16 }
```

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.

Применение интерфейса в программе:

Листинг 11.15

```
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      struct Car : IMovable
16      {
17          public void Move()
18          {
19              Console.WriteLine("Машина едет");
20          }
21      }
22      class Program
23      {
24          static void Action(IMovable movable)
25          {
26              movable.Move();
27          }
28          static void Main(string[] args)
29          {
30              Person person = new Person();
31              Car car = new Car();
32              Action(person);
33              Action(car);
34              Console.Read();
35          }
36      }
37  }
```

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.

Множественное наследование интерфейсов

Множественное наследование – это когда один класс сразу наследуется от нескольких классов. Но бывает так, что базовые классы содержат методы с одинаковыми именами, в результате чего возникают определенные неточности и

ошибки. Множественное наследование есть в языке C++, а в C# от него отказались и внесли интерфейсы. **В C# класс может реализовать сразу несколько интерфейсов.** Это и является главным отличием использования интерфейсов и абстрактных классов. Кроме того, конечно же, абстрактные классы могут содержать все остальные члены, которых не может быть в интерфейсе, и не все методы/свойства в абстрактном классе должны быть абстрактными.

Если класс реализует несколько интерфейсов, они разделяются запятыми:

Листинг 11.16

```
1  using System;
2  namespace HelloApp
3  {
4      interface IAccount
5      {
6          int CurrentSum { get; }  // Текущая сумма на счету
7          void Put(int sum);      // Положить деньги на счет
8          void Withdraw(int sum); // Взять со счета
9      }
10     interface IClient
11     {
12         string Name { get; set; }
13     }
14     class Client : IAccount, IClient
15     {
16         int _sum; // Переменная для хранения суммы
17         public string Name { get; set; }
18         public Client(string name, int sum)
19         {
20             Name = name;
21             _sum = sum;
22         }
23         public int CurrentSum { get { return _sum; } }
24         public void Put(int sum) {  _sum += sum; }
25         public void Withdraw(int sum)
26         {
27             if (_sum >= sum)
28             {
29                 _sum -= sum;
30             }
31         }
32     }
```

```
33      class Program
34      {
35          static void Main(string[] args)
36          {
37              Client client = new Client("Tom", 200);
38              client.Put(30);
39              Console.WriteLine(client.CurrentSum); //230
40              client.Withdraw(100);
41              Console.WriteLine(client.CurrentSum); //130
42              Console.Read();
43          }
44      }
45  }
```

В данном случае определены два интерфейса. Интерфейс `IAccount` определяет свойство `CurrentSum` для текущей суммы денег на счете и два метода `Put` и `Withdraw` для добавления денег на счет и изъятия денег. Интерфейс `IClient` определяет свойство для хранения имени клиента.

Обратите внимание, что свойства `CurrentSum` и `Name` в интерфейсах похожи на автосвойства, но это не автосвойства. При реализации мы можем развернуть их в полноценные свойства, либо же сделать автосвойствами.

Класс Client реализует оба интерфейса и затем применяется в программе.

Интерфейсы в преобразованиях типов

Поскольку класс `Client` реализует интерфейс `IAccount`, то переменная типа `IAccount` может хранить ссылку на объект типа `Client`:

Листинг 11.17

```
1  // Все объекты Client являются объектами IAccount
2  IAccount account = new Client("Том", 200);
3  account.Put(200);
4  Console.WriteLine(account.CurrentSum); // 400
5  // Не все объекты IAccount являются объектами Client, необходимо явное приведение
6  Client client = (Client)account;
7  // Интерфейс IAccount не имеет свойства Name, необходимо явное приведение
8  string clientName = ((Client)account).Name;
```

Преобразование от класса к его интерфейсу, как и преобразование от производного типа к базовому, выполняется автоматически. Так как любой объект `Client` реализует интерфейс `IAccount`.

Обратное преобразование - от интерфейса к реализующему его классу будет аналогично преобразованию от базового класса к производному. Так как не каждый объект `IAccount` является объектом `Client` (ведь интерфейс `IAccount` могут реализовать и другие классы), то для подобного преобразования необходима операция приведения типов. И если мы хотим обратиться к методам класса `Client`, которые не определены в интерфейсе `IAccount`, но являются частью класса `Client`, то нам надо явным образом выполнить преобразование типов:

```
string clientName = ((Client)account).Name;
```

§11.6 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».

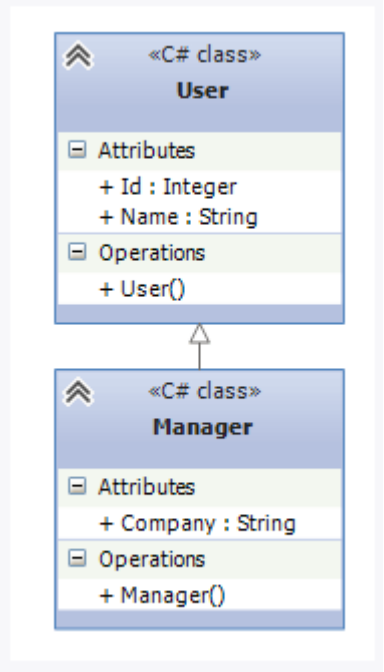
Например:

Листинг 11.18

```
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:

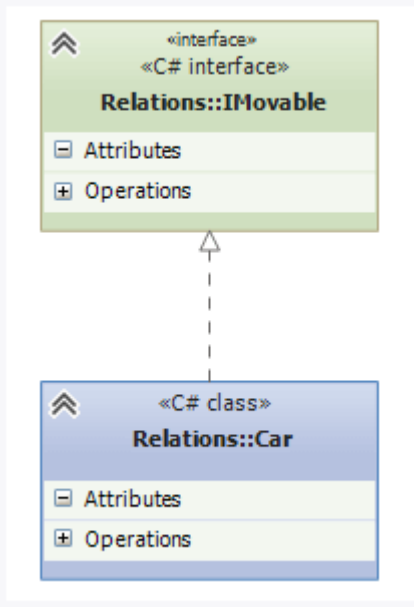


Реализация предполагает определение интерфейса и его реализация в классах. Например, имеется интерфейс `IMovable` с методом `Move`, который реализуется в классе `Car`:

Листинг 11.19

```
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:

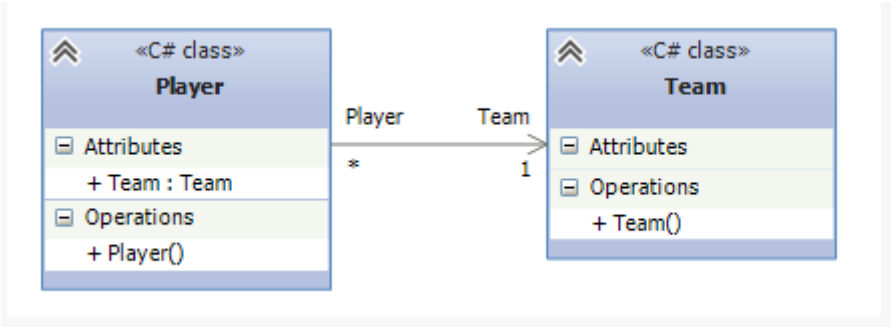


Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Например, объект одного типа содержит или использует объект другого типа. Например, игрок играет в определенной команде:

Листинг 11.20

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

Класс `Player` связан отношением ассоциации с класом `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 1 ко многим. То есть одна команда будет соответствовать многим игрокам.

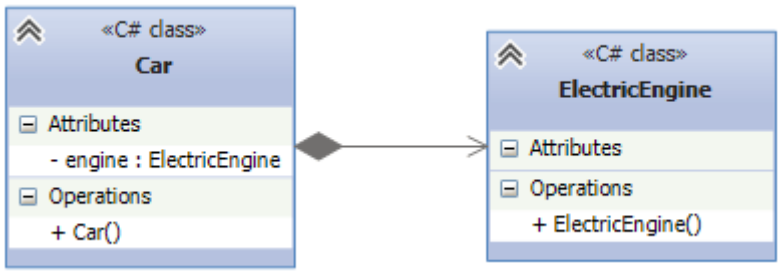
Композиция определяет отношение HAS A, то есть отношение «имеет». Например, в класс автомобиля содержит объект класса электрического двигателя:

Листинг 11.21

```
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:

Листинг 11.22

```
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: