# Наследование и Полиморфизм

Наследование

Представим себя, на минуту, инженерами автомобильного завода. Нашей задачей является разработка современного автомобиля. У нас уже есть предыдущая модель, которая отлично зарекомендовала себя в течение многолетнего использования. Всё бы хорошо, но времена и технологии меняются, а наш современный завод должен стремиться повышать удобство и комфорт выпускаемой продукции и соответствовать современным стандартам.  
  
Нам необходимо выпустить целый модельный ряд автомобилей: седан, универсал и малолитражный хэтч-бэк. Очевидно, что мы не собираемся проектировать новый автомобиль с нуля, а, взяв за основу предыдущее поколение, внесём ряд конструктивных изменений. Например, добавим гидроусилитель руля и уменьшим зазоры между крыльями и крышкой капота, поставим противотуманные фонари. Кроме того, в каждой модели будет изменена форма кузова.  
  
Очевидно, что все три модификации будут иметь большинство свойств прежней модели (старый добрый двигатель 1970 года, непробиваемая ходовая часть, зарекомендовавшая себя отличным образом на отечественных дорогах, коробку передач и т.д.). При этом каждая из моделей будет реализовать некоторую новую функциональность или конструктивную особенность. В данном случае, мы имеем дело с наследованием.  
Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым или родительским. Новый класс – потомком, наследником или производным классом.  
  
Необходимо отметить, что производный класс полностью удовлетворяет спецификации родительского, однако может иметь дополнительную функциональность. С точки зрения интерфейсов, каждый производный класс полностью реализует интерфейс родительского класса. Обратное не верно.  
  
Действительно, в нашем примере мы могли бы произвести с новыми автомобилями все те же действия, что и со старым: увеличить или уменьшить скорость, повернуть, включить сигнал поворота. Однако, дополнительно у нас бы появилась возможность, например, включить противотуманные фонари.  
  
Отсутствие обратной совместимости означает, что мы не должны ожидать от старой модели корректной реакции на такие действия, как включения противотуманок (которых просто нет в данной модели).

Синаксис

class имя\_класса : имя\_класса\_от\_которого\_наследуемся

Давайте сразу тезисно опишем, что такое наследование:

* Это механизм создания нового класса на основе уже существующего старого.
* Старый класс называется «родительским», «предком».
* Новый класс называется «дочерним», «наследником».
* Наследование нужно для повторного использования кода, которое облегчает следование принципу DRY (Don’t Repeat Yourself — Не повторяйся).
* Дочерний класс содержит методы и переменные родительского.

|  |
| --- |
| class Animal  {  public string name;  public int age;  public void Say()  {  Console.WriteLine("Animal somthing says");  }  public void ShowInfo()  {  Console.WriteLine($"name: {name}, age: {age}");  }  }  class Cat : Animal  {  }  class Program  {  static void Main(string[] args)  {  Cat cat = new Cat()  {  name = "Барсик",  age = 3  };  cat.Say();  cat.ShowInfo();  Console.Read();  }  } |

В примере выше мы создали два класса, один из которых наследуется от другого, так класс Cat наследуется от класса Animal. В данном случае в классе Cat не определено ни одного поля, ни одного метода, однако мы можем иметь доступ ко всем полям родительского класса. В прочем мы спокойно можем определить и собственные.

|  |
| --- |
| class Cat : Animal  {  public void Sleep()  {  Console.WriteLine("Cat sleep");  }  } |

При этом этот метод будет доступен только из класса Cat, а в классе Animal он не существует.

В нашем случае, класс Cat является частным случаем класса Animal, поэтому запись следующего вида допустима:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  Animal cat = new Cat();  Console.Read();  }  } |

Таким образом, наследование реализует отношение is-a (является): Животное является кошкой.

Класс предок наследует все поля базового класса и его методы, кроме конструкторов. Если в базовом классе нет пустого конструктора, то класс предок обязан определить у себя один конструкторов базового класса.

Ключевое словое base

Хоть объект предок и должен определять хотя бы из один возможных конструкторов базового объекта, но ему ни кто не запрещает определить собственный, отличающийся от базового объекта конструктор и вызвать из него конструктор базового объекта:

|  |
| --- |
| class Animal  {  public string name;  public int age;  public Animal(string name)  {  this.name = name;  }  public void Say()  {  Console.WriteLine("Animal somthing says");  }  public void ShowInfo()  {  Console.WriteLine($"name: {name}, age: {age}");  }  }  class Cat : Animal  {  public int tailLength;  public Cat(string name, int tailLength): base(name)  {  this.tailLength = tailLength;  }  public void Sleep()  {  Console.WriteLine("Cat sleep");  }  } |

Здесь в конструкторе класса Cat мы вызываем конструктор базового класса Animal.

Но класс предок всегда обязан вызывать один из конструкторов базового класса, если он этого не сделает, то будет ошибка компиляции. Либо можно определить конструктор без параметров.

Порядок выполнения конструкторов примерно похожий, как при вызове нескольких конструкторов:

|  |
| --- |
| static void Main(string[] args)  {  Cat cat = new Cat("Барсик", 20);  Console.Read();  } |

Если рассматривать наш пример, то:

Конструктор класса Cat вызывает конструктор базового класса,

Выполняется конструктор базового класса,

Выполняется конструктор класса Cat.

Класс предок имеет доступ ко всем НЕ приватным полям и методам, т.е. к тем полям и методам, которые не помечены модификатором доступа private. Если какое-то поле помечено модификатором private, то доступ к нему будет, как из внешнего класса с соблюдением всех правил инкапсуляции.

Ограничение:

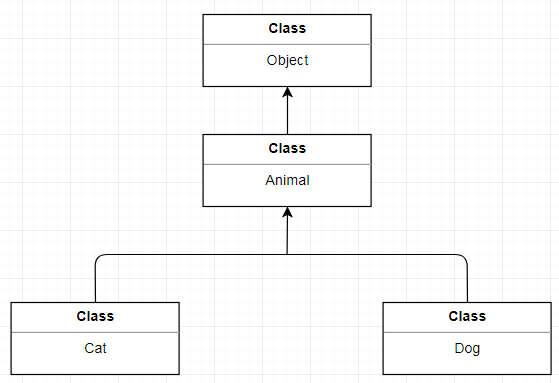
Не может наследоваться больше чем от одного класса, множественное наследование (от нескольких классов) в C# запрещено.

Преобразование типов

К имеющимся уже у нас двум классам Animal и Cat создадим ещё один производный класс от Animal класс – Dog.

### Восходящие преобразования.

Так выглядит иерархия классов:



Объекты производного типа находят внизу иерархии классов. Например объект Dog является и объектом Animal.

И мы можем написать примерно следующее:

|  |
| --- |
| static void Main(string[] args)  {  Cat cat = new Cat("Барсик", 20);  Animal animal = cat; // Преобразование от кошки к животному  Console.Read();  } |

В данном случае переменной animal, которая представляет тип Animal, присваивается ссылка на объект Cat. Но чтобы сохранить ссылку на объект одного класса в переменную другого класса, необходимо выполнить преобразование типов - в данном случае от типа Cat к типу Animal. И так как Cat наследуется от класса Animal, то автоматически выполняется неявное восходящее преобразование - преобразование к типу, которые находятся вверху иерархии классов, то есть к базовому классу.

В итоге переменные cat и animal будут указывать на один и тот же объект в памяти, но переменной animal будет доступна только та часть, которая представляет функционал типа Animal.

Подобным образом поизводятся и другие восходящие преобразования:

|  |
| --- |
| Dog dog = new Animal("Кузя"); |

### Нисходящее преобразования.

Но кроме восходящих преобразований от производного к базовому типу есть нисходящие преобразования или downcasting - от базового типа к производному.

|  |
| --- |
| static void Main(string[] args)  {  Cat cat = new Cat("Барсик", 20);  Animal animal = cat; // Преобразование от кошки к животному  Console.Read();  } |

И может возникнуть вопрос, можно ли обратиться к функционалу типа Cat через переменную типа Animal. Но автоматически такие преобразования не проходят, ведь не каждое животное (объект Animal) является кошкой (объектом Cat). И для нисходящего преобразования необходимо применить явное преобразования, указав в скобках тип, к которому нужно выполнить преобразование:

|  |
| --- |
| static void Main(string[] args)  {  Cat cat = new Cat("Барсик", 20);  Animal animal = cat; // Преобразование от кошки к животному  cat = (Cat)animal; // Преобразование от животного к кошке  } |

В то же время нужно соблюдать некоторую осторожность с такими преобразованиями, к примеру:

|  |
| --- |
| static void Main(string[] args)  {  object obj = new Cat("Tom", 17);  ((Dog)obj).Bark();  } |

В этом случае мы получим ошибку, так как объект obj хранит в себе ссылку на объект Cat, а мы пытаемся вызывать метод объекта Dog.

Есть несколько способов преобразования, самые частые это ключевые слова as и is.

Ключевое слово as. С помощью него программа пытается преобразовать выражение к определенному типу, при этом не выбрасывает исключение. В случае неудачного преобразования выражение будет содержать значение null:

|  |
| --- |
| Animal animal = new Animal("Барсик");  Cat cat = animal as Cat;  if (cat == null)  {  Console.WriteLine("Преобразование прошло неудачно");  }  else  {  cat.Sleep();  } |

Ключевое слово is. В блоке if проверяется допустимость преобразования:

|  |
| --- |
| Animal animal = new Animal("Барсик");  if (animal is Cat)  {  Cat cat = (Cat)animal;  cat.Sleep();  }  else  {  Console.WriteLine("Преобразование не допустимо");  } |

Переопределение методов

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 класе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

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.

Например, рассмотрим следующие классы:

|  |
| --- |
| class Person  {  public string FirstName { get; set; }  public string LastName { get; set; }  public Person(string firstName, string lastName)  {  FirstName = firstName;  LastName = lastName;  }  public virtual void Display()  {  Console.WriteLine($"{FirstName} {LastName}");  }  }  class Employee : Person  {  public string Company { get; set; }  public Employee(string firstName, string lastName, string company)  : base(firstName, lastName)  {  Company = company;  }  } |

Здесь класс Person представляет человека. Класс Employee наследуется от Person и представляет сотруднника предприятия. Этот класс кроме унаследованных свойств 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:

|  |
| --- |
| static void Main(string[] args)  {  Person p1 = new Person("Bill", "Gates");  p1.Display(); // вызов метода Display из класса Person  Employee p2 = new Employee("Tom", "Smith", "Microsoft");  p2.Display(); // вызов метода Display из класса Person  Console.ReadKey();  } |

Вывод:

Bill Gates

Tom Smith

Но также можем переопределить виртуальный метод. Для этого в классе-наследнике определяется метод с модификатором override, который имеет то же именем и набор параметров:

|  |
| --- |
| class Employee : Person  {  public string Company { get; set; }  public Employee(string firstName, string lastName, string company)  : base(firstName, lastName)  {  Company = company;  }  public override void Display()  {  Console.WriteLine($"{FirstName} {LastName} работает в {Company}");  }  } |

Вывод:

Bill Gates

Tom Smith работает в Microsoft

Виртуальные методы базового класса определяют интерфейс всей иерархии, то есть в любом производном классе, который не является прямым наследником от базового класса, можно переопределить виртуальные методы. Например, мы можем определить класс Manager, который будет производным от Employee, и в нем также переопределить метод Display.

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 переопределённый метод также должен иметь модификатор public.
* Нельзя переопределить или объявить виртуальным статический метод.

Переопределение свойств происходит точно также.

### 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 lastName, string firstName, string company)  : base(firstName, lastName)  {  Company = company;  }  public override void Display()  {  base.Display();  Console.WriteLine($"работает в {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 firstName, string lastName, string company)  : base(firstName, lastName)  {  Company = company;  }  public override sealed void Display()  {  Console.WriteLine($"{FirstName} {LastName} работает в {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.

Перезагрузка операторов

### Сокрытие методов

Другим способом изменить функциональность метода, унаследованного от базового класса, является сокрытие (shadowing / hiding).

В производном классе можно определить член с таким же именем, как и у члена его базового класса. В этом случае член базового класса скрывается в производном классе. И хотя формально в C# это не считается ошибкой, компилятор все же выдаст сообщение, предупреждающее о том, что имя скрывается. Если член базового класса требуется скрыть намеренно, то перед его именем следует указать ключевое слово new, чтобы избежать появления подобного предупреждающего сообщения. Следует, однако, иметь в виду, что это совершенно отдельное применение ключевого слова new, не похожее на его применение при создании экземпляра объекта:

Фактически сокрытие представляет определение в классе-наследнике метода или свойства, которые соответствует по имени и набору параметров методу или свойству базового класса. Для сокрытия членов класса применяется ключевое слово new, например:

|  |
| --- |
| class Person  {  public string FirstName { get; set; }  public string LastName { get; set; }  public Person(string firstName, string lastName)  {  FirstName = firstName;  LastName = lastName;  }  public void Display()  {  Console.WriteLine($"{FirstName} {LastName}");  }  }  class Employee : Person  {  public string Company { get; set; }  public Employee(string firstName, string lastName, string company)  : base(firstName, lastName)  {  Company = company;  }  public new void Display()  {  Console.WriteLine($"{FirstName} {LastName} работает в {Company}");  }  } |

Здесь определен класс Person, представляющий человека, и класс Employee, представляющий работника предприятия. Employee наследует от Person все свойства и методы. Но в классе Employee кроме унаследованных свойств есть также и собственное свойство Company, которое хранит название компании. И мы хотели бы в методе Display выводить информацию о компании вместе с именем и фамилией на консоль. Для этого определяется метод Display с ключевым словом new, который скрывает реализацию данного метода из базового класса.

В каких ситуациях можно использовать сокрытие? Например, в примере выше метод Display в базовом классе не является виртуальным, мы не можем его переопределить, но, допустим, нам не устраивает его реализация для производного класса, поэтому мы можем воспользоваться сокрытием, чтобы определить нужный нам функционал.

|  |
| --- |
| static void Main(string[] args)  {  Person bob = new Person("Bob", "Robertson");  bob.Display(); // Bob Robertson  Employee tom = new Employee("Tom", "Smith", "Microsoft");  tom.Display(); // Tom Smith работает в Microsoft  Console.ReadKey();  } |

Таким же образом можно организовать сокрытие свойств.

При этом если мы хотим обратиться именно к реализации свойства или метода в базовом классе, то опять же мы можем использовать ключевое слово base и через него обращаться к функциональности базового класса.

### Разница между сокрытием и переопределением

Мы рассмотрели два способа изменения функциональности методов, унаследованных от базового класса - сокрытие и переопределение. В чем разница между двумя этими способами?

**Переопределение:**

|  |
| --- |
| class Person  {  public string FirstName { get; set; }  public string LastName { get; set; }  public Person(string firstName, string lastName)  {  FirstName = firstName;  LastName = lastName;  }  public virtual void Display()  {  Console.WriteLine($"{FirstName} {LastName}");  }  }  class Employee : Person  {  public string Company { get; set; }  public Employee(string firstName, string lastName, string company)  : base(firstName, lastName)  {  Company = company;  }  public override void Display()  {  Console.WriteLine($"{FirstName} {LastName} работает в {Company}");  }  } |

Также создадим объект Employee и передадим его переменной типа Person:

|  |
| --- |
| Person tom = new Employee("Tom", "Smith", "Microsoft");  tom.Display(); // Tom Smith работает в Microsoft |

Для работы с виртуальными методами компилятор формирует таблицу виртуальных методов (Virtual Method Table или VMT). В нее записывается адреса виртуальных методов. Для каждого класса создается своя таблица.

Когда создается объект класса, то компилятор передает в конструктор объекта специальный код, который связывает объект и таблицу VMT.

А при вызове виртуального метода из объекта берется адрес его таблицы VMT. Затем из VMT извлекается адрес метода и ему передается управление. То есть процесс выбора реализации метода производится во время выполнения программы. Собственно так и выполняется виртуальный метод. Следует учитывать, что так как среде выполнения вначале необходимо получить из таблицы VMT адрес нужного метода, то это немного замедляет выполнение программы.

**Сокрытие:**

Теперь возьмем те же классы Person и Employee, но вместо переопределения используем сокрытие:

|  |
| --- |
| class Person  {  public string FirstName { get; set; }  public string LastName { get; set; }  public Person(string firstName, string lastName)  {  FirstName = firstName;  LastName = lastName;  }  public void Display()  {  Console.WriteLine($"{FirstName} {LastName}");  }  }  class Employee : Person  {  public string Company { get; set; }  public Employee(string firstName, string lastName, string company)  : base(firstName, lastName)  {  Company = company;  }  public new void Display()  {  Console.WriteLine($"{FirstName} {LastName} работает в {Company}");  }  } |

Результат теперь такой:

|  |
| --- |
| Person tom = new Employee("Tom", "Smith", "Microsoft");  tom.Display(); // Tom Smith |

Переменная tom представляет тип Person, но хранит ссылку на объект Employee. Однако при вызове метода Display будет выполняться та версия метода, которая определена именно в классе Person, а не в классе Employee. Почему? Класс Employee никак не переопределяет метод Display, унаследованный от базового класса, а фактически определяет новый метод. Поэтому при вызове tom.Display() вызывается метод Display из класса Person.

# Абстрактные классы

Что такое 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:

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

Иногда требуется создать базовый класс, в котором определяется лишь самая общая форма для всех его производных классов, а наполнение ее деталями предоставляется каждому из этих классов. В таком классе определяется лишь характер методов, которые должны быть конкретно реализованы в производных классах, а не в самом базовом классе. Подобная ситуация возникает, например, в связи с невозможностью получить содержательную реализацию метода в базовом классе.

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:

|  |
| --- |
| 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;  }  } |

Однако несмотря на то, что напрямую мы не можем вызвать конструктор класса Person для создания объекта, тем не менее конструктор в абстрактных классах то же может играть важную роль, в частности, инициализировать некоторые общие для производных классов переменные и свойства, как в случае со свойством Name. И хотя в примере выше конструктор класса Person не вызывается, тем не менее производные классы Client и Employee могут обращаться к нему.

Абстрактные члены класса

Кроме обычных свойств и методов абстрактный класс может иметь абстрактные члены классов, 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 (как и при обычном переопределении виртуальных методов и свойств). Также следует учесть, что если класс имеет хотя бы одный абстрактный метод (или абстрактные свойство, индексатор, событие), то этот класс должен быть определен как абстрактный.

Абстрактный метод создается с помощью указываемого модификатора типа abstract. У абстрактного метода отсутствует тело, и поэтому он не реализуется в базовом классе. Это означает, что он должен быть переопределен в производном классе, поскольку его вариант из базового класса просто непригоден для использования. Нетрудно догадаться, что абстрактный метод автоматически становится виртуальным и не требует указания модификатора virtual. В действительности совместное использование модификаторов virtual и abstract считается ошибкой.

Когда производный класс наследует абстрактный класс, в нем должны быть реализованы все абстрактные методы базового класса. В противном случае производный класс должен быть также определен как abstract. Таким образом, атрибут abstract наследуется до тех пор, пока не будет достигнута полная реализация класса.

# Интерфейсы

Для начала ознакомимся с формальным определением типа интерфейса. Интерфейс (interface) представляет собой не более чем просто именованный набор абстрактных членов. Абстрактные методы являются чистым протоколом, поскольку не имеют никакой стандартной реализации. Конкретные члены, определяемые интерфейсом, зависят от того, какое поведение моделируется с его помощью. Это действительно так. Интерфейс выражает поведение, которое данный класс или структура может избрать для поддержки. Более того, каждый класс (или структура) может поддерживать столько интерфейсов, сколько необходимо, и, следовательно, тем самым поддерживать множество поведений.

В интерфейсе ни у одного из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации. В нем указывается только, что именно следует делать, но не как это делать. Как только интерфейс будет определен, он может быть реализован в любом количестве классов. Кроме того, в одном классе может быть реализовано любое количество интерфейсов.

Если абстрактные классы описывают форму класса, то интерфейсы описывают действия объекта.

Интерфейс представляет ссылочный тип, который определяет набор методов и свойств, но не реализует их. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

Для определения интерфейса используется ключевое слово interface. 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.

|  |
| --- |
| interface IMovable  {  void Move();  } |

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

Еще один момент в объявлении интерфейса: все 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. Поэтому весь функционал должен быть открыт для реализации.

В целом интерфейсы могут определять следующие сущности:

* Методы
* Свойства
* Индексаторы
* События

Однако интерфейсы не могут определять статические члены, переменные, константы.

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

|  |
| --- |
| // применение интерфейса в классе  class Person : IMovable  {  public void Move()  {  Console.WriteLine("Человек идет");  }  }  // применение интерфейса в структуре  struct Car : IMovable  {  public void Move()  {  Console.WriteLine("Машина едет");  }  } |

При применени 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. При этом поскольку все методы и свойства интерфейса являются публичными, при реализации этих методов и свойств в классе к ним можно применять только модификатор public. Поэтому если класс должен иметь метод с каким-то другим модификатором, например, protected, то интерфейс не подходит для определения подобного метода.

Если класс или структура не реализуют какие-либо свойства или методы интерфейса, то мы столкнемся с ошибкой на этапе компиляции.

|  |
| --- |
| using System;  namespace HelloApp  {  interface IMovable  {  void Move();  }  class Person : IMovable  {  public void Move()  {  Console.WriteLine("Человек идет");  }  }  struct Car : IMovable  {  public void Move()  {  Console.WriteLine("Машина едет");  }  }  class Program  {  static void Action(IMovable movable)  {  movable.Move();  }  static void Main(string[] args)  {  Person person = new Person();  Car car = new Car();  Action(person);  Action(car);  Console.Read();  }  }  } |

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.

|  |
| --- |
| Консольный вывод данной программы:  Человек идет  Машина едет |

Если класс применяет интерфейс, то этот класс должен реализовать все методы и свойства интерфейса. Однако также можно и не реализовать методы, сделав их абстрактными, переложив право их реализации на производные классы.

При реализации интерфейса учитываются также методы и свойства, унаследованные от базового класса. Например:

|  |
| --- |
| interface IAction  {  void Move();  }  class BaseAction  {  public void Move()  {  Console.WriteLine("Move in BaseAction");  }  }  class HeroAction : BaseAction, IAction  {  } |

Здесь класс HeroAction реализует интерфейс IAction, однако для реализации метода Move из интерфейса применяется метод Move, унаследованный от базового класса BaseAction. Таким образом, класс HeroAction может не реализовать метод Move, так как этот метод уже определен в базовом классе BaseAction.

Следует отметить, что если класс одновременно наследует другой класс и реализует интерфейс, как в примере выше класс HeroAction, то название базового класса должно быть указано до реализуемых интерфейсов: class HeroAction : BaseAction, IAction

Множественная реализация интерфейсов

Интерфейсы имеют еще одну важную функцию: в C# не поддерживается множественное наследование, то есть мы можем унаследовать класс только от одного класса, в отличие, скажем, от языка С++, где множественное наследование можно использовать. Интерфейсы позволяют частично обойти это ограничение, поскольку в C# класс может реализовать сразу несколько интерфейсов. Все реализуемые интерфейсы указываются через запятую:

|  |
| --- |
| interface IAccount  {  int CurrentSum { get; } // Текущая сумма на счету  void Put(int sum); // Положить деньги на счет  void Withdraw(int sum); // Взять со счета  }  interface IClient  {  string Name { get; set; }  }  class Client : IAccount, IClient  {  int sum; // Переменная для хранения суммы  public string Name { get; set; }  public Client(string name, int sum)  {  Name = name;  this.sum = sum;  }  public int CurrentSum { get { return sum; } }  public void Put(int sum) { this.sum += sum; }  public void Withdraw(int sum)  {  if (this.sum >= sum)  {  this.sum -= sum;  }  }  }  class Program  {  static void Main(string[] args)  {  Client client = new Client("Tom", 200);  client.Put(30);  Console.WriteLine(client.CurrentSum); //230  client.Withdraw(100);  Console.WriteLine(client.CurrentSum); //130  Console.Read();  }  } |

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.

Обатите 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.

Тема на самостоятельное изучение:

Перезагрузка операторов

Индексаторы