**4-bob.**

**Наследование**

Наследование (inheritance) является одним из ключевых моментов ООП. Благодаря наследованию один класс может унаследовать функциональность другого класса.

Пусть у нас есть следующий класс Person, который описывает отдельного человека:

class Person

{

    private string \_name = "";

    public string Name

    {

        get { return \_name; }

        set { \_name = value; }

    }

    public void Print()

    {

        Console.WriteLine(Name);

    }

}

Но вдруг нам потребовался класс, описывающий сотрудника предприятия - класс Employee. Поскольку этот класс будет реализовывать тот же функционал, что и класс Person, так как сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (или наследником, или подклассом) от класса Person, который, в свою очередь, называется базовым классом или родителем (или суперклассом):

class Employee : Person

{

}

После двоеточия мы указываем базовый класс для данного класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же свойства, методы, поля, которые есть в классе Person. Единственное, что не передается при наследовании, это конструкторы базового класса с параметрами.

Таким образом, наследование реализует отношение **is-a** (является), объект класса Employee также является объектом класса Person:

|  |  |
| --- | --- |
| 1  2  3  4 | Person person = new Person { Name = "Tom" };  person.Print();   // Tom  person = new Employee { Name = "Sam" };  person.Print();   // Sam |

И поскольку объект Employee является также и объектом Person, то мы можем так определить переменную: Person p = new Employee().

По умолчанию все классы наследуются от базового класса **Object**, даже если мы явным образом не устанавливаем наследование. Поэтому выше определенные классы Person и Employee кроме своих собственных методов, также будут иметь и методы класса Object: ToString(), Equals(), GetHashCode() и GetType().

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

* Не поддерживается множественное наследование, класс может наследоваться только от одного класса.
* При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**.

Однако следует также учитывать, что если базовый и производный класс находятся в разных сборках (проектах), то в этом случае производый класс может наследовать только от класса, который имеет модификатор public.

* Если класс объявлен с модификатором **sealed**, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:
* sealed class Admin
* {
* }
* Нельзя унаследовать класс от статического класса.

### Доступ к членам базового класса из класса-наследника

Вернемся к нашим классам Person и Employee. Хотя Employee наследует весь функционал от класса Person, посмотрим, что будет в следующем случае:

class Employee : Person

{

    public void PrintName()

    {

        Console.WriteLine(\_name);

    }

}

Этот код не сработает и выдаст ошибку, так как переменная \_name объявлена с модификатором private и поэтому к ней доступ имеет только класс Person. Но зато в классе Person определено общедоступное свойство Name, которое мы можем использовать, поэтому следующий код у нас будет работать нормально:

|  |
| --- |
| class Employee : Person  {      public void PrintName()      {          Console.WriteLine(Name);      }  } |

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами **private protected** (если базовый и производный класс находятся в одной сборке), **public**, **internal** (если базовый и производный класс находятся в одной сборке), **protected** и **protected internal**.

### Ключевое слово base

Теперь добавим в наши классы конструкторы:

class Person

{

    public string Name { get; set;}

    public Person(string name)

    {

        Name = name;

    }

    public void Print()

    {

        Console.WriteLine(Name);

    }

}

class Employee : Person

{

    public string Company { get; set; }

    public Employee(string name, string company)

        : base(name)

    {

        Company = company;

    }

}

Класс Person имеет конструктор, который устанавливает свойство Name. Поскольку класс Employee наследует и устанавливает то же свойство Name, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса Person. К тому же свойств, которые надо установить в конструкторе базового класса, и параметров может быть гораздо больше.

С помощью ключевого слова **base** мы можем обратиться к базовому классу. В нашем случае в конструкторе класса Employee нам надо установить имя и компанию. Но имя мы передаем на установку в конструктор базового класса, то есть в конструктор класса Person, с помощью выражения base(name).

|  |  |
| --- | --- |
| 1  2  3  4 | Person person = new Person("Bob");  person.Print();     // Bob  Employee employee = new Employee("Tom", "Microsoft");  employee.Print();   // Tom |

### Конструкторы в производных классах

Конструкторы не передаются производному классу при наследовании. И если в базовом классе **не определен** конструктор по умолчанию без параметров, а только конструкторы с параметрами (как в случае с базовым классом Person), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово base. Например, из класса Employee уберем определение конструктора:

|  |  |
| --- | --- |
| 1  2  3  4 | class Employee : Person  {      public string Company { get; set; } = "";  } |

В данном случае мы получим ошибку, так как класс Employee не соответствует классу Person, а именно не вызывает конструктор базового класса. Даже если бы мы добавили какой-нибудь конструктор, который бы устанавливал все те же свойства, то мы все равно бы получили ошибку:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Employee : Person  {      public string Company { get; set; } = "";      public Employee(string name, string company)    // ! Ошибка      {          Name = name;          Company = company;      }  } |

То есть в классе Employee через ключевое слово **base** надо явным образом вызвать конструктор класса Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Employee : Person  {      public string Company { get; set; } = "";      public Employee(string name, string company)          : base(name)      {          Company = company;      }  } |

Либо в качестве альтернативы мы могли бы определить в базовом классе конструктор без параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Person  {      public string Name { get; set; }      // конструктор без параметров      public Person()      {          Name = "Tom";          Console.WriteLine("Вызов конструктора без параметров");      }      public Person(string name)      {          Name = name;      }      public void Print()      {          Console.WriteLine(Name);      }  } |

Тогда в любом конструкторе производного класса, где нет обращения конструктору базового класса, все равно неявно вызывался бы этот конструктор по умолчанию. Например, следующий конструктор

|  |  |
| --- | --- |
| 1  2  3  4 | public Employee(string company)  {      Company = company;  } |

Фактически был бы эквивалентен следующему конструктору:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public Employee(string company)      :base()  {      Company = company;  } |

### Порядок вызова конструкторов

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных. Например, возьмем следующие классы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class Person  {      string name;      int age;        public Person(string name)      {          this.name = name;          Console.WriteLine("Person(string name)");      }      public Person(string name, int age) : this(name)      {          this.age = age;          Console.WriteLine("Person(string name, int age)");      }  }  class Employee : Person  {      string company;        public Employee(string name, int age, string company) : base(name, age)      {          this.company = company;          Console.WriteLine("Employee(string name, int age, string company)");      }  } |

При создании объекта Employee:

|  |  |
| --- | --- |
| 1 | Employee tom = new Employee("Tom", 22, "Microsoft"); |

Мы получим следующий консольный вывод:

Person(string name)

Person(string name, int age)

Employee(string name, int age, string company)

В итоге мы получаем следующую цепь выполнений.

1. Вначале вызывается конструктор Employee(string name, int age, string company). Он делегирует выполнение конструктору Person(string name, int age)
2. Вызывается конструктор Person(string name, int age), который сам пока не выполняется и передает выполнение конструктору Person(string name)
3. Вызывается конструктор Person(string name), который передает выполнение конструктору класса System.Object, так как это базовый по умолчанию класс для Person.
4. Выполняется конструктор System.Object.Object(), затем выполнение возвращается конструктору Person(string name)
5. Выполняется тело конструктора Person(string name), затем выполнение возвращается конструктору Person(string name, int age)
6. Выполняется тело конструктора Person(string name, int age), затем выполнение возвращается конструктору Employee(string name, int age, string company)
7. Выполняется тело конструктора Employee(string name, int age, string company). В итоге создается объект Employee

## Преобразование типов

В предыдущей главе мы говорили о преобразованиях объектов простых типов. Сейчас затронем тему преобразования объектов классов. Допустим, у нас есть следующая иерархия классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class Person  {      public string Name { get; set; }      public Person(string name)      {          Name = name;      }      public void Print()      {          Console.WriteLine($"Person {Name}");      }  }    class Employee : Person  {      public string Company { get; set; }      public Employee(string name, string company) : base(name)      {          Company = company;      }  }    class Client : Person  {      public string Bank { get; set; }      public Client(string name, string bank) : base(name)      {          Bank = bank;      }  } |

В этой иерархии классов мы можем проследить следующую цепь наследования: Object (все классы неявно наследуются от типа Object) -> Person -> Employee|Client.

Причем в этой иерархии классов базовые типы находятся вверху, а производные типы - внизу.

### Восходящие преобразования. Upcasting

Объекты производного типа (который находится внизу иерархии) в то же время представляют и базовый тип. Например, объект Employee в то же время является и объектом класса Person. Что в принципе естественно, так как каждый сотрудник (Employee) является человеком (Person). И мы можем написать, например, следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4 | Employee employee = new Employee("Tom", "Microsoft");  Person person = employee;   // преобразование от Employee к Person    Console.WriteLine(person.Name); |

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

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

Подобным образом поизводятся и другие восходящие преобразования:

|  |  |
| --- | --- |
| 1 | Person bob = new Client("Bob", "ContosoBank");   // преобразование от Client к Person |

Здесь переменная bob, которая представляет тип Person, хранит ссылку на объект Client, поэтому также выполняется восходящее неявное преобразование от производного класса Client к базовому типу Person.

Восходящее неявное преобразование будет происходить и в следующем случае:

|  |  |
| --- | --- |
| 1  2  3 | object person1 = new Employee("Tom", "Microsoft");  // от Employee к object  object person2 = new Client("Bob", "ContosoBank");  // от Client к object  object person3 = new Person("Sam");                 // от Person к object |

Так как тип object - базовый для всех остальных типов, то преобразование к нему будет производиться автоматически.

### Нисходящие преобразования. Downcasting

Но кроме восходящих преобразований от производного к базовому типу есть нисходящие преобразования или downcasting - от базового типа к производному. Например, в следующем коде переменная person хранит ссылку на объект Employee:

|  |  |
| --- | --- |
| 1  2 | Employee employee = new Employee("Tom", "Microsoft");  Person person = employee;   // преобразование от Employee к Person |

И может возникнуть вопрос, можно ли обратиться к функционалу типа Employee через переменную типа Person. Но автоматически такие преобразования не проходят, ведь не каждый человек (объект Person) является сотрудником предприятия (объектом Employee). И для нисходящего преобразования необходимо применить явное преобразование, указав в скобках тип, к которому нужно выполнить преобразование:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Employee employee1 = new Employee("Tom", "Microsoft");  Person person = employee1;   // преобразование от Employee к Person    //Employee employee2 = person;    // так нельзя, нужно явное преобразование  Employee employee2 = (Employee)person;  // преобразование от Person к Employee |

Рассмотрим некоторые примеры преобразований:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // Объект Employee также представляет тип object  object obj = new Employee("Bill", "Microsoft");    // чтобы обратиться к возможностям типа Employee, приводим объект к типу Employee  Employee employee = (Employee) obj;    // объект Client также представляет тип Person  Person person = new Client("Sam", "ContosoBank");  // преобразование от типа Person к Client  Client client = (Client)person; |

В первом случае переменной obj присвоена ссылка на объект Employee, поэтому мы можем преобразовать объект obj к любому типу который располагается в иерархии классов между типом object и Employee.

Если нам надо обратиться к каким-то отдельным свойствам или методам объекта, то нам необязательно присваивать преобразованный объект переменной:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // Объект Employee также представляет тип object  object obj = new Employee("Bill", "Microsoft");    // преобразование к типу Person для вызова метода Print  ((Person)obj).Print();  // либо так  // ((Employee)obj).Print();    // преобразование к типу Employee, чтобы получить свойство Company  string company = ((Employee)obj).Company; |

В то же время необходимо соблюдать осторожность при подобных преобразованиях. Например, что будет в следующем случае:

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Объект Employee также представляет тип object  object obj = new Employee("Bill", "Microsoft");    // преобразование к типу Client, чтобы получить свойство Bank  string bank = ((Client)obj).Bank; |

В данном случае мы получим ошибку, так как переменная obj хранит ссылку на объект Employee. Данный объект является также объектом типов object и Person, поэтому мы можем преобразовать его к этим типам. Но к типу Client мы преобразовать не можем.

Другой пример:

|  |  |
| --- | --- |
| 1  2  3  4 | Employee employee1 = new Person("Tom"); // ! Ошибка    Person person = new Person("Bob");  Employee employee2 = (Employee) person; // ! Ошибка |

В данном случае мы пытаемся преобразовать объект типа Person к типу Employee, а объект Person не является объектом Employee. Причем в последнем случае Visual Studio не подскжет, что в данной строке ошибка, и данная строка даже нормально скомилируется, тем не менее в процессе выполнения программы мы получим ощибку. В этом в том числе и кроектся коварство преобразований, поэтому в подобных ситуациях надо проявлять осторожность.

Существует ряд способов, чтобы избежать подобных ошибок преобразования.

### Способы преобразований

Во-первых, можно использовать 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Person person = new Person("Tom");  Employee? employee = person as Employee;  if (employee == null)  {      Console.WriteLine("Преобразование прошло неудачно");  }  else  {      Console.WriteLine(employee.Company);  } |

Стоит отметить, что переменная employee здесь определяется не просто как переменная Employee, а именно **Employee?** - после названия типа ставится вопросительный знак. Что указывает, что переменная может хранить как значение null, так и значение Employee.

Второй способ заключается в проверке допустимости преобразования с помощью ключевого слова **is**:

|  |  |
| --- | --- |
| 1 | значение is тип |

Если значение слева от оператора представляет тип, указаный справа от оператора, то оператор is возвращает true, иначе возвращается false.

Причем оператор **is** позволяет автоматически преобразовать значение к типу, если это значение представляет данный тип. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Person person = new Person("Tom");  if (person is Employee employee)  {      Console.WriteLine(employee.Company);  }  else  {      Console.WriteLine("Преобразование не допустимо");  } |

Выражение if (person is Employee employee) проверяет, является ли переменная person объектом типа Employee. И если person является объектом Employee, то автоматически преобразует значение переменной person в тип Employee и преобразованное значение сохраняет в переменную employee. Далее в блоке if мы можем использовать объект employee как значение типа Employee.

Однако, если person не является объектом Employee, как в данном случае, то такая проверка вернет значение false, и преобразование не сработает.

Оператор **is** также можно применять и без преобразования, просто проверяя на соответствие типу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Person person = new Person("Tom");  if (person is Employee)  {      Console.WriteLine("Представляет тип Employee");  }  else  {      Console.WriteLine("НЕ является объектом типа Employee");  } |

## Виртуальные методы и свойства

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

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

Чтобы сделать метод Print 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 Print из класса Person:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Person bob = new Person("Bob");  bob.Print(); // вызов метода Print из класса Person    Employee tom = new Employee("Tom", "Microsoft");  tom.Print(); // вызов метода Print из класса Person |

Консольный вывод:

Bob

Tom

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

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

Возьмем те же самые объекты:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Person bob = new Person("Bob");  bob.Print(); // вызов метода Print из класса Person    Employee tom = new Employee("Tom", "Microsoft");  tom.Print(); // вызов метода Print из класса Employee |

Консольный вывод:

Bob

Tom 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 Print.

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.

### Ключевое слово base

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

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

### Переопределение свойств

Также как и методы, можно переопределять свойства:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class Person  {      int age = 1;      public virtual int Age      {          get => age;          set{ if(value > 0 && value < 110) age = value; }      }      public string Name { get; set; }      public Person(string name)      {          Name = name;      }      public virtual void Print() => Console.WriteLine(Name);  }  class Employee : Person  {      public override int Age      {          get => base.Age;          set { if (value > 17 && value < 110) base.Age = value; }      }      public string Company { get; set; }      public Employee(string name, string company)          : base(name)      {          Company = company;          base.Age = 18; // возраст для работников по умолчанию      }  } |

В данном случае в классе Person определено виртуальное свойство Age, которое устанавливает значение, если оно больше 0 и меньше 110. В классе Employee это свойство переопределено - возраст работника должен быть не меньше 18.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Person bob = new Person("Bob");  Console.WriteLine(bob.Age); // 1    Employee tom = new Employee("Tom", "Microsoft");  Console.WriteLine(tom.Age); // 18  tom.Age = 22;  Console.WriteLine(tom.Age); // 22  tom.Age = 12;  Console.WriteLine(tom.Age); // 22 |

### Запрет переопределения методов

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

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

## Скрытие методов и свойств

В прошлой теме было рассмотрено определение и переопределение виртуальных методов. Другим способом изменить функциональность метода, унаследованного от базового класса, является **скрытие** (shadowing / hiding).

Фактически скрытие метода/свойства представляет определение в классе-наследнике метода или свойства, которые соответствует по имени и набору параметров методу или свойству базового класса. Для скрытия членов класса применяется ключевое слово **new**. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class Person  {      public string Name { get; set; }      public Person(string name)      {          Name = name;      }      public void Print()      {          Console.WriteLine($"Name: {Name}");      }  }  class Employee : Person  {      public string Company { get; set; }      public Employee(string name, string company)                  : base(name)      {          Company = company;      }      public new void Print()      {          Console.WriteLine($"Name: {Name}   Company: {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 Print выводить информацию о компании вместе с именем на консоль. Для этого определяется метод Print с ключевым словом **new**, который скрывает реализацию данного метода из базового класса.

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

Используем эти классы в программе в методе Main:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Person bob = new Person("Bob");  bob.Print();    // Name: Bob    Employee tom = new Employee("Tom", "Microsoft");  tom.Print();    // Name: Tom  Company: Microsoft |

Консольный вывод программы:

Name: Bob

Name: Tom Company: Microsoft

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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Employee : Person  {      public string Company { get; set; }      public Employee(string name, string company)                  : base(name)      {          Company = company;      }      public new void Print()      {          base.Print();   // вызываем метод Print из базового класса Person          Console.WriteLine($"Company: {Company}");      }  } |

### Скрытие свойств

Подобным обазом мы можем организовать скрытие свойств:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | Person bob = new Person("Bob");  Console.WriteLine(bob.Name);    // Bob    Employee tom = new Employee("Tom", "Microsoft");  Console.WriteLine(tom.Name);    // Mr./Ms. Tom    class Person  {      public string Name { get; set; }      public Person(string name)      {          Name = name;      }  }  class Employee : Person  {      // скрываем свойство Name базового класса      public new string Name      {          get => $"Mr./Ms. {base.Name}";          set => base.Name = value;        }      public string Company { get; set; }      public Employee(string name, string company)                  : base(name)      {          Company = company;      }  } |

В данном случае в классе Employee переопределено свойство Name. В блоке get нем мы берем значение свойства из базовового класса P erson и присоединяем к нему "Mr./Ms.". В блоке set передаем полученное значение в реализацию свойства Name базового класса Person

### Скрытие переменных и констант

В отличие от переопределения C# позволяет применять скрытие к переменным (как к статическим, так и нестатическим) и константам, также используя ключевое слово **new**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Console.WriteLine(Person.minAge);     // 1  Console.WriteLine(Person.typeName);   // Person    Console.WriteLine(Employee.minAge);     // 18  Console.WriteLine(Employee.typeName);   // Employee    class Person  {      public readonly static int minAge = 1;      public const string typeName = "Person";  }  class Employee : Person  {      // скрываем поля и константы базового класса      public new readonly static int minAge = 18;      public new const string typeName = "Employee";  } |

## Различие переопределения и скрытия методов

Ранее было рассмотрена два способа изменения функциональности методов, унаследованных от базового класса - скрытие и переопределение. В чем разница между двумя этими способами?

### Переопределение

Возьмем пример с переопределением методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class Person  {      public string Name { get; set; }      public Person(string name)      {          Name = name;      }      public virtual void Print()      {          Console.WriteLine(Name);      }  }  class Employee : Person  {      public string Company { get; set; }      public Employee(string name, string company)          : base(name)      {          Company = company;      }        public override void Print()      {          Console.WriteLine($"{Name} работает в {Company}");      }  } |

Используем классы в программе:

|  |  |
| --- | --- |
| 1  2 | Person tom = new Employee("Tom", "Microsoft");  tom.Print();        // Tom работает в Microsoft |

При вызове tom.Print() выполняется реализация метода Print из класса Employee, несмотря на то, что переменная tom - переменная типа Person.

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.

### Скрытие

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class Person  {      public string Name { get; set; }      public Person(string name)      {          Name = name;      }        public void Print()      {          Console.WriteLine(Name);      }  }    class Employee : Person  {      public string Company { get; set; }      public Employee(string name, string company)              : base(name)      {          Company = company;      }      public new void Print()      {          Console.WriteLine($"{Name} работает в {Company}");      }  } |

И посмотрим, что будет в следующем случае:

|  |  |
| --- | --- |
| 1  2 | Person tom = new Employee("Tom", "Microsoft");  tom.Print();        // Tom |

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

## Абстрактные классы и члены классов

### Абстрактные классы

Кроме обычных классов в C# есть **абстрактные классы**. Зачем они нужны? Классы обычно представляют некий план определенного рода объектов или сущностей. Например, мы можем определить класс Car для преставления машин или класс Person для представления людей, вложив в эти классы соответствующие свойства, поля, методы, которые будут описывать данные объекты. Однако некоторые сущности, которые мы хотим выразить с помощью языка программирования, могут не иметь конкретного воплощения. Например, в реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами. И для описания подобных сущностей, которые не имеют конкретного воплощения, предназначены 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**. Например, определим абстрактный класс, который представляет некое транспортное средство:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | abstract class Transport  {      public void Move()      {          Console.WriteLine("Транспортно средство движется");      }  } |

Транспортное средство представляет некоторую абстракцию, которая не имеет конкретного воплощения. То есть есть легковые и грузовые машины, самолеты, морские судна, кто-то на космическом корабле любит покататься, но как такового транспортного средства нет. Тем не менее все транспортные средства имеют нечто общее - они могут перемещаться. И для этого в классе определен метод Move, который эмулирует перемещение.

Но главное отличие абстрактных классов от обычных состоит в том, что мы **НЕ можем** использовать конструктор абстрактного класса для создания экземпляра класса. Например, следующим образом:

|  |  |
| --- | --- |
| 1 | Transport tesla = new Transport(); |

Тем не менее абстрактные классы полезны для описания некоторого общего функционала, который могут наследовать и использовать производные классы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Transport car = new Car();  Transport ship = new Ship();  Transport aircraft = new Aircraft();    car.Move();  ship.Move();  aircraft.Move();  abstract class Transport  {      public void Move()      {          Console.WriteLine("Транспортное средство движется");      }  }  // класс корабля  class Ship : Transport { }  // класс самолета  class Aircraft : Transport { }  // класс машины  class Car : Transport { } |

В данном случае от класса Transport наследуются три класса, которые представляют различные типы транспортных средств. Тем не менее они имеют общую черту - они могут перемещаться с помощью метода Move().

Выше писалось, что мы не можем использовать конструктор абстрактного класса для создания экземпляра этого класса. Тем не менее такой класс также может определять конструкторы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | Transport car = new Car("машина");  Transport ship = new Ship("корабль");  Transport aircraft = new Aircraft("самолет");    car.Move();         // машина движется  ship.Move();        // корабль движется  aircraft.Move();    // самолет движется  abstract class Transport  {      public string Name { get; }      // конструктор абстрактного класса Transport      public Transport(string name)      {          Name = name;      }      public void Move() =>Console.WriteLine($"{Name} движется");  }  // класс корабля  class Ship : Transport  {      // вызываем конструктор базового класса      public Ship(string name) : base(name) { }  }  // класс самолета  class Aircraft : Transport  {      public Aircraft(string name) : base(name) { }  }  // класс машины  class Car : Transport  {      public Car(string name) : base(name) { }  } |

В данном случае в абстрактном классе Transport определен конструктор - с помощью параметра он устанавливает значение свойства Name, которое хранит название транспортного средства. И в этом случае производные классы должны в своих конструкторах вызвать этот конструктор.

### Абстрактные члены классов

Кроме обычных свойств и методов абстрактный класс может иметь абстрактные члены классов, которые определяются с помощью ключевого слова **abstract** и не имеют никакого функционала. В частности, абстрактными могут быть:

* Методы
* Свойства
* Индексаторы
* События

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**.

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

#### Абстрактные методы

Например, выше в примере с транспортными средствами метод Move описывает передвижение транспортного средства. Однако различные типы транспорта перемещаются по разному - едят по земле, летят по воздуху, плывут на воде и т.д. В этом случае мы можем сделать метод Move абстрактным, а его реализацию переложить на производные классы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | abstract class Transport  {      public abstract void Move();  }  // класс корабля  class Ship : Transport  {      // мы должны реализовать все абстрактные методы и свойства базового класса      public override void Move()      {          Console.WriteLine("Корабль плывет");      }  }  // класс самолета  class Aircraft : Transport  {      public override void Move()      {          Console.WriteLine("Самолет летит");      }  }  // класс машины  class Car : Transport  {      public override void Move()      {          Console.WriteLine("Машина едет");      }  } |

Применение классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Transport car = new Car();  Transport ship = new Ship();  Transport aircraft = new Aircraft();    car.Move();         // машина едет  ship.Move();        // корабль плывет  aircraft.Move();    // самолет летит |

#### Абстрактные свойства

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | abstract class Transport  {      // абстрактное свойство для хранения скорости      public abstract int Speed { get; set; }  }  // класс корабля  class Ship: Transport  {      int speed;      public override int Speed      {          get => speed;          set => speed = value;      }  }    class Aircraft : Transport  {      public override int Speed { get; set; }  } |

В классе Transport определено абстрактное свойство Speed, которое должно хранить скорость транспортного средства. Оно похоже на автосвойство, но это не автосвойство. Так как данное свойство не должно иметь реализацию, то оно имеет только пустые блоки get и set. В производных классах мы можем переопределить это свойство, сделав его полноценным свойством (как в классе Ship), либо же сделав его автоматическим (как в классе Aircraft).

#### Отказ от реализации абстрактных членов

Производный класс обязан реализовать все абстрактные члены базового класса. Однако мы можем отказаться от реализации, но в этом случае производный класс также должен быть определен как абстрактный:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Transport tesla = new Auto();  tesla.Move();           // легковая машина едет  abstract class Transport  {      public abstract void Move();  }  // класс машины  abstract class Car :Transport{}    class Auto: Car  {      public override void Move()      {          Console.WriteLine("легковая машина едет");      }  } |

В данном случае класс Car не реализует абстрактный метод Move базового класса Transport и поэтому также определен как абстрактный. Однако любые неабстрактные классы, производные от Car, все равно должны реализовать все унаследованные абстрактные методы и свойства.

### Пример абстрактного класса

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | // абстрактный класс фигуры  abstract class Shape  {      // абстрактный метод для получения периметра      public abstract double GetPerimeter();      // абстрактный метод для получения площади      public abstract double GetArea();  }  // производный класс прямоугольника  class Rectangle : Shape  {      public float Width { get; set; }      public float Height { get; set; }        // переопределение получения периметра      public override double GetPerimeter() => Width \* 2 + Height \* 2;      // переопрелеление получения площади      public override double GetArea() => Width \* Height;  }  // производный класс окружности  class Circle : Shape  {      public double Radius { get; set; }        // переопределение получения периметра      public override double GetPerimeter() => Radius \* 2 \* 3.14;      // переопрелеление получения площади      public override double GetArea() => Radius \* Radius \* 3.14;  } |

Применение классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var rectanle = new Rectangle { Width = 20, Height = 20 };  var circle = new Circle { Radius = 200 };  PrintShape(rectanle); // Perimeter: 80   Area: 400  PrintShape(circle); // Perimeter: 1256  Area: 125600    void PrintShape(Shape shape)  {      Console.WriteLine($"Perimeter: {shape.GetPerimeter()}  Area: {shape.GetArea()}");  } |

## Класс System.Object и его методы

Все классы в .NET, даже те, которые мы сами создаем, а также базовые типы, такие как **System.Int32**, являются неявно производными от класса Object. Даже если мы не указываем класс Object в качестве базового, по умолчанию неявно класс Object все равно стоит на вершине иерархии наследования. Поэтому все типы и классы могут реализовать те методы, которые определены в классе System.Object. Рассмотрим эти методы.

### ToString

Метод ToString служит для получения строкового представления данного объекта. Для базовых типов просто будет выводиться их строковое значение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int i = 5;  Console.WriteLine(i.ToString()); // выведет число 5    double d = 3.5;  Console.WriteLine(d.ToString()); // выведет число 3,5 |

Для классов же этот метод выводит полное название класса с указанием пространства имен, в котором определен этот класс. И мы можем переопределить данный метод. Посмотрим на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Person person = new Person { Name = "Tom" };  Console.WriteLine(person.ToString()); // выведет название класса Person    Clock clock = new Clock { Hours = 15, Minutes = 34, Seconds = 53 };  Console.WriteLine(clock.ToString()); // выведет 15:34:53    class Clock  {      public int Hours { get; set; }      public int Minutes { get; set; }      public int Seconds { get; set; }      public override string ToString()      {          return $"{Hours}:{Minutes}:{Seconds}";      }  }  class Person  {      public string Name { get; set; } = "";  } |

Для переопределения метода ToString() в классе Clock, который представляет часы, используется ключевое слово **override** (как и при обычном переопределении виртуальных или абстрактных методов). В данном случае метод ToString() выводит в строке значения свойств Hours, Minutes, Seconds.

Класс Person не переопределяет метод ToString, поэтому для этого класса срабатывает стандартная реализация этого метода, которая выводит просто название класса.

Кстати в данном случае мы могли задействовать обе реализации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Person tom = new Person { Name = "Tom" };  Console.WriteLine(tom.ToString()); // Tom    Person undefined = new Person();  Console.WriteLine(undefined.ToString()); // Person    class Person  {      public string Name { get; set; } = "";        public override string? ToString()      {          if (string.IsNullOrEmpty(Name))              return base.ToString();          return Name;      }  } |

То есть если имя - свойство Name не имеет значения, оно представляет пустую строку, то возвращается базовая реализация - название класса. Стоит отметить, что базовая реализация возвращает не просто строку, а объект **string?** - то есть это может быть строка string, либо значение **null**, которое указывает на отсутствие значения. И в реальности в качестве возвращаемого типа для метода мы можем использовать как **string**, так и **string?**

Если же имя у объекта Person установлено, то возвращается значение свойства Name. Для проверки строки на наличие значения применяется метод String.IsNullOrEmpty().

Стоит отметить, что различные технологии на платформе .NET активно используют метод ToString для разных целей. В частности, тот же метод Console.WriteLine() по умолчанию выводит именно строковое представление объекта. Поэтому, если нам надо вывести строковое представление объекта на консоль, то при передаче объекта в метод Console.WriteLine необязательно использовать метод ToString() - он вызывается неявно:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Person person = new Person { Name = "Tom" };  Console.WriteLine(person);  // Tom    Clock clock = new Clock { Hours = 15, Minutes = 34, Seconds = 53 };  Console.WriteLine(clock); // выведет 15:34:53 |

### Метод GetHashCode

Метод **GetHashCode** позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты. Можно определять самые разные алгоритмы генерации подобного числа или взять реализаци базового типа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Person  {      public string Name { get; set; } = "";        public override int GetHashCode()      {          return Name.GetHashCode();      }  } |

В данном случае метод GetHashCode возвращает хеш-код для значения свойства Name. То есть два объекта Person, которые имеют одно и то же имя, будут возвращать один и тот же хеш-код. Однако в реальности алгоритм может быть самым различным.

### Получение типа объекта и метод GetType

Метод **GetType** позволяет получить тип данного объекта:

|  |  |
| --- | --- |
| 1  2 | Person person = new Person { Name = "Tom" };  Console.WriteLine(person.GetType());    // Person |

Этот метод возвращает объект **Type**, то есть тип объекта.

С помощью ключевого слова **typeof** мы получаем тип класса и сравниваем его с типом объекта. И если этот объект представляет тип Client, то выполняем определенные действия.

|  |  |
| --- | --- |
| 1  2  3 | object person = new Person { Name = "Tom" };  if (person.GetType() == typeof(Person))      Console.WriteLine("Это реально класс Person"); |

Причем поскольку класс Object является базовым типом для всех классов, то мы можем переменной типа object присвоить объект любого типа. Однако для этой переменной метод GetType все равно вернет тот тип, на объект которого ссылается переменная. То есть в данном случае объект типа Person.

Стоит отметить, что проверку типа можно скоратить с помощью оператора **is**:

|  |  |
| --- | --- |
| 1  2  3 | object person = new Person { Name = "Tom" };  if (person is Person)      Console.WriteLine("Это реально класс Person"); |

В отличие от методов ToString, Equals, GetHashCode метод **GetType()** не переопределяется.

### Метод Equals

Метод Equals позволяет сравнить два объекта на равенство. В качестве параметра он принимает объект для сравнения в виде типа object и возврашает true, если оба объекта равны:

|  |  |
| --- | --- |
| 1 | public override bool Equals(object? obj) {......} |

Например, реализуем данный метод в классе Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Person  {      public string Name { get; set; } = "";      public override bool Equals(object? obj)      {          // если параметр метода представляет тип Person          // то возвращаем true, если имена совпадают          if (obj is Person person) return Name == person.Name;          return false;      }      // вместе с методом Equals следует реализовать метод GetHashCode      public override int GetHashCode() => Name.GetHashCode();  } |

Метод Equals принимает в качестве параметра объект любого типа, который мы затем приводим к текущему классу - классу Person.

Если переданный объект представляет тип Person, то возвращаем результат сравнения имен двух объектов Person. Если же объект представляет другой тип, то возвращается false.

В данном случае для примера применяется довольно простой алгоритм сравнения, однако при необходимости реализацию метода можно сделать более сложной, например, сравнивать по нескольким свойствам при их наличии.

Стоит отметить, что вместе с методом Equals следует реализовать метод GetHashCode.

Применение метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var person1 = new Person { Name = "Tom" };  var person2 = new Person { Name = "Bob" };  var person3 = new Person { Name = "Tom" };    bool person1EqualsPerson2 = person1.Equals(person2);   // false  bool person1EqualsPerson3 = person1.Equals(person3);   // true    Console.WriteLine(person1EqualsPerson2);    // false  Console.WriteLine(person1EqualsPerson3);    // true |

И если следует сравнивать два сложных объекта, как в данном случае, то лучше использовать метод Equals, а не стандартную операцию ==.

## Обобщения

Кроме обычных типов фреймворк .NET также поддерживает обобщенные типы (generics), а также создание обобщенных методов. Чтобы разобраться в особенности данного явления, сначала посмотрим на проблему, которая могла возникнуть до появления обобщенных типов. Посмотрим на примере. Допустим, мы определяем класс для хранения данных пользователя:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Person  {      public int Id { get;}      public string Name { get;}      public Person(int id, string name)      {          Id = id;          Name = name;      }  } |

Класс Person определяет два свойства: Id - уникальный идентификатор пользователя и Name - имя пользователя.

Здесь идентификатор пользователя задан как числовое значение, то есть это будут значения 1, 2, 3, 4 и так далее.

Однако также нередко для идентификатора используются и строковые значения. И у числовых, и у строковых значений есть свои плюсы и минусы. И на момент написания класса мы можем точно не знать, что лучше выбрать для хранения идентификатора - строки или числа. Либо, возможно, этот класс будет использоваться другими разработчиками, которые могут иметь свое мнение по данной проблеме, например, они могут для представления идентификатора создать специальный класс.

И на первый взгляд, чтобы выйти из подобной ситуации, мы можем определить свойство Id как свойство типа object. Так как тип object является универсальным типом, от которого наследуется все типы, соответственно в свойствах подобного типа мы можем сохранить и строки, и числа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Person  {      public object Id { get;}      public string Name { get;}      public Person(object id, string name)      {          Id = id;          Name = name;      }  } |

Затем этот класс можно было использовать для создания пользователей в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Person tom = new Person(546, "Tom");  Person bob = new Person("abc123", "Bob");    int tomId = (int)tom.Id;  string bobId = (string) bob.Id;    Console.WriteLine(tomId);   // 546  Console.WriteLine(bobId);   // abc123 |

Все вроде замечательно работает, но такое решение является не очень оптимальным. Дело в том, что в данном случае мы сталкиваемся с такими явлениями как **упаковка (boxing)** и **распаковка (unboxing)**.

Так, при передаче в конструктор значения типа int, происходит упаковка этого значения в тип Object:

|  |  |
| --- | --- |
| 1 | Person tom = new Person(546, "Tom");    // упаковка в значения int в тип Object |

Чтобы обратно получить данные в переменную типов int, необходимо выполнить распаковку:

|  |  |
| --- | --- |
| 1 | int tomId = (int)tom.Id;        // Распаковка в тип int |

Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа **System.Object** и сохраняет его в управляемой куче (хипе). Распаковка (unboxing), наоборот, предполагает преобразование объекта типа object к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.

Кроме того, существует другая проблема - проблема безопасности типов. Так, мы получим ошибку во время выполнения программы, если напишем следующим образом:

|  |  |
| --- | --- |
| 1  2  3 | Person tom = new Person(546, "Tom");  string tomId = (string)tom.Id;  // !Ошибка  - Исключение InvalidCastException  Console.WriteLine(tomId);   // 546 |

Мы можем не знать, какой именно объект представляет Id, и при попытке получить число в данном случае мы столкнемся с исключением InvalidCastException. Причем с исключением мы столкнемся на этае выполнения программы.

Для решения этих проблем в язык C# была добавлена поддержка **обобщенных типов** (также часто называют универсальными типами). Обобщенные типы позволяют указать конкретный тип, который будет использоваться. Поэтому определим класс Person как обощенный:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Person<T>  {      public T Id { get; set; }      public string Name { get; set; }      public Person(T id, string name)      {          Id = id;          Name = name;      }  } |

Угловые скобки в описании class Person<T> указывают, что класс является обобщенным, а тип T, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву T, это может быть и любая другая буква или набор символов. Причем сейчас на этапе написания кода нам неизвестно, что это будет за тип, это может быть любой тип. Поэтому параметр **T** в угловых скобках еще называется **универсальным параметром**, так как вместо него можно подставить любой тип.

Например, вместо параметра T можно использовать объект int, то есть число, представляющее номер пользователя. Это также может быть объект string, либо или любой другой класс или структура:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Person<int> tom = new Person<int>(546, "Tom");  // упаковка не нужна  Person<string> bob = new Person<string>("abc123", "Bob");    int tomId = tom.Id;      // распаковка не нужна  string bobId = bob.Id;  // преобразование типов не нужно    Console.WriteLine(tomId);   // 546  Console.WriteLine(bobId);   // abc123 |

Поскольку класс Person является обобщенным, то при определении переменной после названия типа в угловых скобках необходимо указать тот тип, который будет использоваться вместо универсального параметра T. В данном случае объекты Person типизируются типами int и string:

|  |  |
| --- | --- |
| 1  2 | Person<int> tom = new Person<int>(546, "Tom");  // упаковка не нужна  Person<string> bob = new Person<string>("abc123", "Bob"); |

Поэтому у первого объекта tom свойство Id будет иметь тип int, а у объекта bob - тип string. И в случае с типом int упаковки происходить не будет.

При попытке передать для параметра id значение другого типа мы получим ошибку компиляции:

|  |  |
| --- | --- |
| 1 | Person<int> tom = new Person<int>("546", "Tom");  // ошибка компиляции |

А при получении значения из Id нам больше не потребуется операция приведения типов и распаковка тоже применяться не будет:

|  |  |
| --- | --- |
| 1 | int tomId = tom.Id;      // распаковка не нужна |

Тем самым мы избежим проблем с типобезопасностью. Таким образом, используя обобщенный вариант класса, мы снижаем время на выполнение и количество потенциальных ошибок.

При этом универсальный параметр также может представлять обобщенный тип:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // класс компании  class Company<P>  {      public P CEO { get; set; }  // президент компании      public Company(P ceo)      {          CEO = ceo;      }  }  class Person<T>  {      public T Id { get;}      public string Name { get;}      public Person(T id, string name)      {          Id = id;          Name = name;      }  } |

Здесь класс компании определяет свойство CEO, которое хранит президента компании. И мы можем передать для этого свойства значение типа Person, типизированного каким-нибудь типом:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Person<int> tom = new Person<int>(546, "Tom");  Company<Person<int>> microsoft =  new Company<Person<int>>(tom);    Console.WriteLine(microsoft.CEO.Id);  // 546  Console.WriteLine(microsoft.CEO.Name);  // Tom |

### Статические поля обобщенных классов

При типизации обобщенного класса определенным типом будет создаваться свой набор статических членов. Например, в классе Person определено следующее статическое поле:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Person<T>  {      public static T? code;      public T Id { get; set; }      public string Name { get; set; }      public Person(T id, string name)      {          Id = id;          Name = name;      }  } |

Теперь типизируем класс двумя типами int и string:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Person<int> tom = new Person<int>(546, "Tom");  Person<int>.code = 1234;    Person<string> bob = new Person<string>("abc", "Bob");  Person<string>.code = "meta";    Console.WriteLine(Person<int>.code);       // 1234  Console.WriteLine(Person<string>.code);   // meta |

В итоге для Person<string> и для Person<int> будет создана своя переменная code.

### Использование нескольких универсальных параметров

Обобщения могут использовать несколько универсальных параметров одновременно, которые могут представлять одинаковые или различные типы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Person<T, K>  {      public T Id { get;}      public K Password { get; set; }      public string Name { get;}      public Person(T id, K password, string name)      {          Id = id;          Name = name;          Password = password;      }  } |

Здесь класс Person использует два универсальных параметра: один параметр для идентификатора, другой параметр - для свойства-пароля. Применим данный класс:

|  |  |
| --- | --- |
| 1  2  3 | Person<int, string> tom = new Person<int, string>(546, "qwerty", "Tom");  Console.WriteLine(tom.Id);  // 546  Console.WriteLine(tom.Password);// qwerty |

Здесь объект Person типизируется типами int и string. То есть в качестве универсального параметра T используется тип int, а для параметра K - тип string.

### Обобщенные методы

Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int x = 7;  int y = 25;  Swap<int>(ref x, ref y); // или так Swap(ref x, ref y);  Console.WriteLine($"x={x}    y={y}");   // x=25   y=7    string s1 = "hello";  string s2 = "bye";  Swap<string>(ref s1, ref s2); // или так Swap(ref s1, ref s2);  Console.WriteLine($"s1={s1}    s2={s2}"); // s1=bye   s2=hello    void Swap<T>(ref T x, ref T y)  {      T temp = x;      x = y;      y = temp;  } |

Здесь определен обощенный метод Swap, который принимает параметры по ссылке и меняет их значения. При этом в данном случае не важно, какой тип представляют эти параметры.

При вызове метода Swap типизируем его определенным типом и передаем ему соответствующие этому типу значения.

## Ограничения обобщений

С помощью универсальных параметров мы можем типизировать обобщенные классы любым типом. Однако иногда возникает необходимость конкретизировать тип. Например, у нас есть следующий класс Message, который представляет некоторое сообщение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Message  {      public string Text { get; } // текст сообщения      public Message(string text)      {          Text = text;      }  } |

И, допустим, мы хотим определить метод для отправки сообщений в виде объектов Message. На первый взгляд мы можем определить и использовать следующий метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SendMessage(new Message("Hello World"));    void SendMessage(Message message)  {      Console.WriteLine($"Отправляется сообщение: {message.Text}");  } |

Метод SendMessage в качестве параметра message принимает объект Message и эмулирует его отправку. Вроде все нормально и что-то лучше вряд ли можно придумать. Но у класса Message могут быть классы-наследники. Например, класс EmailMessage для email-сообщений, SmsMessage - для sms-сообщений и так далее

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class EmailMessage : Message  {      public EmailMessage(string text) : base(text) { }  }  class SmsMessage : Message  {      public SmsMessage(string text) : base(text) { }  } |

Что если мы хотим также отправлять сообщения, которые представляют эти классы? Проблемы вроде нет, поскольку метод SendMessage принимает объект Message и соответственно также и объекты производных классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SendMessage(new EmailMessage("Hello World"));    void SendMessage(Message message)  {      Console.WriteLine($"Отправляется сообщение: {message.Text}");  } |

Но здесь мы сталкиваемся с преобразованием типов: от EmailMessage к Message. Кроме того, опять же возможна проблема типобезопасности, если мы захотим преобразовать объект message в объект производных классов. И в этом случае чтобы избежать преобразований, мы можем применить обобщения:

|  |  |
| --- | --- |
| 1  2  3  4 | void SendMessage<T>(T message)  {      Console.WriteLine($"Отправляется сообщение: {message.Text}");   // ! Ошибка - свойство Text  } |

Обобщения позволяют избежать преобразований, но теперь мы сталкиваемся с другой проблемой - универсальный параметр T подразумевает любой тип. Но не любой тип имеет свойство Text. Соответственно свойство Text для объекта типа T не определено и мы не можем это свойство использоваться. Более того для объекта T по умолчанию нам достуны только методы типа object.

Таким образом, возникает проблема: надо избежать преобразований типов и соответственно использовать обобщения, а с другой стороны, необходимо обращаться внутри метода к функционалу класса Message. И ограничения обобщений позволяют решить эту проблему.

### Ограничения методов

Ограничения методов указываются после списка параметров после оператора **where**:

|  |  |
| --- | --- |
| 1 | имя\_метода<T>(параметры) where T: тип\_ограничения |

После оператора **where** указывается универсальный параметр, для которого применяется ограничение. И через двоеточие указывается тип ограничения - обычно в качестве ограничения выступает конкретный тип.

Например, применим ограничения к методу SendMessage, который отправляет объекты Message

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | SendMessage(new Message("Hello World"));  SendMessage(new EmailMessage("Bye World"));    void SendMessage<T>(T message) where T: Message  {      Console.WriteLine($"Отправляется сообщение: {message.Text}");  }  class Message  {      public string Text { get; } // текст сообщения      public Message(string text)      {          Text = text;      }  }  class EmailMessage : Message  {      public EmailMessage(string text) : base(text) { }  } |

Выражение where T: Message в определении метода SendMessage говорит, что через универсальный параметр T будут передаваться объекты класса Message и производных классов. Благодаря этому компилятор будет знать, что T будет иметь функционал класса Message, и соответственно мы сможем обратиться к методам и свойствам класса Message внутри метода без проблем.

При вызове метода нам необязательно указывать тип в угловых скобках - компилятор на основании переданного значения сам определит каким тиом типизиуется метод:

|  |  |
| --- | --- |
| 1 | SendMessage(new EmailMessage("Bye World")); |

Однако это можно сделать и явно

|  |  |
| --- | --- |
| 1 | SendMessage<EmailMessage>(new EmailMessage("Bye World")); |

### Ограничения обобщений в типах

Подобным образом можно определять и ограничения обобщенных типов. Например, ограничения обобщенных классов:

|  |  |
| --- | --- |
| 1 | class имя\_класса<T> where T: тип\_ограничения |

В качестве примера определим класс мессенджера, который будет отправлять сообшения в виде объектов Message:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class Messenger<T> where T : Message  {      public void SendMessage(T message)      {          Console.WriteLine($"Отправляется сообщение: {message.Text}");      }  }    class Message  {      public string Text { get; } // текст сообщения      public Message(string text)      {          Text = text;      }  }  class EmailMessage : Message  {      public EmailMessage(string text) : base(text) { }  } |

Здесь для класса Messenger опять же установлено ограничение where T : Message. То есть внутри класса Messenger все объекты типа T можно использовать как объекты Message. И в данном случае в классе Messenger в методе SendMessage опять эмулируется отправка сообшений.

Применим класс для отправки сообщений:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Messenger<Message> telegram = new Messenger<Message>();  telegram.SendMessage(new Message("Hello World"));    Messenger<EmailMessage> outlook = new Messenger<EmailMessage>();  outlook.SendMessage(new EmailMessage("Bye World")); |

### Типы ограничений и стандартные ограничения

В качестве ограничений мы можем использовать следующие типы:

* Классы
* Интерфейсы
* class - универсальный параметр должен представлять класс
* struct - универсальный параметр должен представлять структуру
* new() - универсальный параметр должен представлять тип, который имеет общедоступный (public) конструктор без параметров

Есть ряд стандартных ограничений, которые мы можем использовать. В частности, можно указать ограничение, чтобы использовались только структуры или другие типы значений:

|  |  |
| --- | --- |
| 1  2 | class Messenger<T> where T : struct  {} |

При этом использовать в качестве ограничения конкретные структуры в отличие от классов нельзя.

Также можно задать в качестве ограничения ссылочные типы:

|  |  |
| --- | --- |
| 1  2 | class Messenger<T> where T : class  {} |

А также можно задать с помощью слова **new** в качестве ограничения класс или структуру, которые имеют общедоступный конструктор без параметров:

|  |  |
| --- | --- |
| 1  2 | class Messenger<T> where T : new()  {} |

Если для универсального параметра задано несколько ограничений, то они должны идти в определенном порядке:

1. Название класса, class, struct. Причем мы можем одновременно определить только одно из этих ограничений
2. Название интерфейса
3. new()

|  |  |
| --- | --- |
| 1  2  3  4 | class Smartphone<T> where T: Messenger, new()  {    } |

### Использование нескольких универсальных параметров

Если класс использует несколько универсальных параметров, то последовательно можно задать ограничения к каждому из них:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class Messenger<T, P>      where T : Message      where P: Person  {      public void SendMessage(P sender, P receiver, T message)      {          Console.WriteLine($"Отправитель: {sender.Name}");          Console.WriteLine($"Получатель: {receiver.Name}");          Console.WriteLine($"Сообщение: {message.Text}");      }  }  class Person  {      public string Name { get;}      public Person(string name) => Name = name;  }    class Message  {      public string Text { get; } // текст сообщения      public Message(string text) =>  Text = text;  } |

В данном случае для параметра P будут передаваться объекты типа Person, а для параметра T - объекты Message.

Применим классы:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Messenger<Message, Person> telegram = new Messenger<Message, Person>();  Person tom = new Person("Tom");  Person bob = new Person("Bob");  Message hello = new Message("Hello, Bob!");  telegram.SendMessage(tom, bob, hello); |

Консольный вывод:

Отправитель: Tom

Получатель: Bob

Сообщение: Hello, Bob!

## Наследование обобщенных типов

Один обобщенный класс может быть унаследован от другого обобщенного. При этом можно использовать различные варианты наследования.

Допустим, у нас есть следующий базовый класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Person<T>  {      public T Id { get;}      public Person(T id)      {          Id = id;      }  } |

Первый вариант заключается в создание класса-наследника, который типизирован тем же типом, что и базовый:

|  |  |
| --- | --- |
| 1  2  3  4 | class UniversalPerson<T> : Person<T>  {      public UniversalPerson(T id) : base(id) { }  } |

Применение класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Person<string> person1 = new Person<string>("34");  Person<int> person3 = new UniversalPerson<int>(45);  UniversalPerson<int> person2 = new UniversalPerson<int>(33);  Console.WriteLine(person1.Id);  Console.WriteLine(person2.Id);  Console.WriteLine(person3.Id); |

Второй вариант представляет создание обычного необобщенного класса-наследника. В этом случае при наследовании у базового класса надо явным образом определить используемый тип:

|  |  |
| --- | --- |
| 1  2  3  4 | class StringPerson : Person<string>  {      public StringPerson(string id) : base(id) { }  } |

Теперь в производном классе в качестве типа будет использоваться тип string. Применение класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | StringPerson person4 = new StringPerson("438767");  Person<string> person5 = new StringPerson("43875");  // так нельзя написать  //Person<int> person6 = new StringPerson("45545");  Console.WriteLine(person4.Id);  Console.WriteLine(person5.Id); |

Третий вариант представляет типизацию производного класса параметром совсем другого типа, отличного от универсального параметра в базовом классе. В этом случае для базового класса также надо указать используемый тип:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class IntPerson<T> : Person<int>  {      public T Code { get; set; }      public IntPerson(int id, T code) : base(id)      {          Code = code;      }  } |

Здесь тип IntPerson типизирован еще одним типом, который может не совпадать с типом, который используется базовым классом. Применение класса:

|  |  |
| --- | --- |
| 1  2  3  4 | IntPerson<string> person7 = new IntPerson<string>(5, "r4556");  Person<int> person8 = new IntPerson<long>(7, 4587);  Console.WriteLine(person7.Id);  Console.WriteLine(person8.Id); |

И также в классах-наследниках можно сочетать использование универсального параметра из базового класса с применением своих параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class MixedPerson<T, K> : Person<T>      where K : struct  {      public K Code { get; set; }      public MixedPerson(T id, K code) : base(id)      {          Code = code;      }  } |

Здесь в дополнение к унаследованному от базового класса параметру T добавляется новый параметр K. Также если необходимо при этом задать ограничения, мы их можем указать после названия базового класса. Применение класса:

|  |  |
| --- | --- |
| 1  2  3  4 | MixedPerson<string, int> person9 = new MixedPerson<string, int>("456", 356);  Person<string> person10 = new MixedPerson<string, int>("9867", 35678);  Console.WriteLine(person9.Id);  Console.WriteLine(person10.Id); |

При этом стоит учитывать, что если на уровне базового класса для универсального параметра установлено ограничение, то подобное ограничение должно быть определено и в производных классах, которые также используют этот параметр:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Person<T> where T : class  {      public T Id { get;}      public Person(T id) => Id = id;  }  class UniversalPerson<T> : Person<T> where T: class  {      public UniversalPerson(T id) : base(id) { }  } |

То есть если в базовом классе в качестве ограничение указано class, то есть любой класс, то в производном классе также надо указать в качестве ограничения class, либо же какой-то конкретный класс.

**5-bob**

# **Обработка исключений**

## Конструкция try..catch..finally

Иногда при выполнении программы возникают ошибки, которые трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Например, при передачи файла по сети может неожиданно оборваться сетевое подключение. такие ситуации называются **исключениями**. Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция **try...catch...finally**.

|  |
| --- |
| try  {    }  catch  {    }  finally  {    } |

При использовании блока **try...catch..finally** вначале выполняются все инструкции в блоке **try**. Если в этом блоке не возникло исключений, то после его выполнения начинает выполняться блок **finally**. И затем конструкция try..catch..finally завершает свою работу.

Если же в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается, и среда CLR начинает искать блок **catch**, который может обработать данное исключение. Если нужный блок catch найден, то он выполняется, и после его завершения выполняется блок finally.

Если нужный блок catch не найден, то при возникновении исключения программа аварийно завершает свое выполнение.

Рассмотрим следующий пример:

|  |  |
| --- | --- |
| 1  2  3  4 | int x = 5;  int y = x / 0;  Console.WriteLine($"Результат: {y}");  Console.WriteLine("Конец программы"); |

В данном случае происходит деление числа на 0, что приведет к генерации исключения. И при запуске приложения в режиме отладки мы увидим в Visual Studio окошко, которое информирует об исключении:

В этом окошке мы видим, что возникло исключение, которое представляет тип **System.DivideByZeroException**, то есть попытка деления на ноль. С помощью пункта **View Details** можно посмотреть более детальную информацию об исключении.

И в этом случае единственное, что нам остается, это завершить выполнение программы.

Чтобы избежать подобного аварийного завершения программы, следует использовать для обработки исключений конструкцию **try...catch...finally**. Так, перепишем пример следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | try  {      int x = 5;      int y = x / 0;      Console.WriteLine($"Результат: {y}");  }  catch  {      Console.WriteLine("Возникло исключение!");  }  finally  {      Console.WriteLine("Блок finally");  }  Console.WriteLine("Конец программы"); |

В данном случае у нас опять же возникнет исключение в блоке try, так как мы пытаемся разделить на ноль. И дойдя до строки

|  |  |
| --- | --- |
| 1 | int y = x / 0; |

выполнение программы остановится. CLR найдет блок **catch** и передаст управление этому блоку.

После блока catch будет выполняться блок finally.

Возникло исключение!

Блок finally

Конец программы

Таким образом, программа по-прежнему не будет выполнять деление на ноль и соответственно не будет выводить результат этого деления, но теперь она не будет аварийно завершаться, а исключение будет обрабатываться в блоке catch.

Следует отметить, что в этой конструкции обязателен блок **try**. При наличии блока catch мы можем опустить блок finally:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | try  {      int x = 5;      int y = x / 0;      Console.WriteLine($"Результат: {y}");  }  catch  {      Console.WriteLine("Возникло исключение!");  } |

И, наоборот, при наличии блока finally мы можем опустить блок catch и не обрабатывать исключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | try  {      int x = 5;      int y = x / 0;      Console.WriteLine($"Результат: {y}");  }  finally  {      Console.WriteLine("Блок finally");  } |

Однако, хотя с точки зрения синтаксиса C# такая конструкция вполне корректна, тем не менее, поскольку CLR не сможет найти нужный блок catch, то исключение не будет обработано, и программа аварийно завершится.

### Обработка исключений и условные конструкции

Ряд исключительных ситуаций может быть предвиден разработчиком. Например, пусть в программе есть метод, который принимает строку, конвертирует ее в число и вычисляет квадрат этого числа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Square("12"); // Квадрат числа 12: 144  Square("ab"); // !Исключение    void Square(string data)  {      int x = int.Parse(data);      Console.WriteLine($"Квадрат числа {x}: {x \* x}");  } |

Если пользователь передаст в метод не число, а строку, которая содежит нецифровые символы, то программа выпадет в ошибку. С одной стороны, здесь как раз та ситуация, когда можно применить блок try..catch, чтобы обработать возможную ошибку. Однако гораздо оптимальнее было бы проверить допустимость преобразования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Square("12"); // Квадрат числа 12: 144  Square("ab"); // Некорректный ввод    void Square(string data)  {      if (int.TryParse(data, out var x))      {          Console.WriteLine($"Квадрат числа {x}: {x \* x}");      }      else      {          Console.WriteLine("Некорректный ввод");      }  } |

Метод int.TryParse() возвращает true, если преобразование можно осуществить, и false - если нельзя. При допустимости преобразования переменная x будет содержать введенное число. Так, не используя try...catch можно обработать возможную исключительную ситуацию.

С точки зрения производительности использование блоков try..catch более накладно, чем применение условных конструкций. Поэтому по возможности вместо try..catch лучше использовать условные конструкции на проверку исключительных ситуаций.

## Блок catch и фильтры исключений

### Определение блока catch

За обработку исключения отвечает блок **catch**, который может иметь следующие формы:

|  |  |
| --- | --- |
| 1  2  3  4 | catch  {      // выполняемые инструкции  } |

* Обрабатывает любое исключение, которое возникло в блоке try. Выше уже был продемонстрирован пример подобного блока.

|  |  |
| --- | --- |
| 1  2  3  4 | catch (тип\_исключения)  {      // выполняемые инструкции  } |

* Обрабатывает только те исключения, которые соответствуют типу, указаному в скобках после оператора catch.
* Например, обработаем только исключения типа DivideByZeroException:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | try  {      int x = 5;      int y = x / 0;      Console.WriteLine($"Результат: {y}");  }  catch(DivideByZeroException)  {      Console.WriteLine("Возникло исключение DivideByZeroException");  } |

* Однако если в блоке try возникнут исключения каких-то других типов, отличных от DivideByZeroException, то они не будут обработаны.

|  |  |
| --- | --- |
| 1  2  3  4 | catch (тип\_исключения имя\_переменной)  {      // выполняемые инструкции  } |

* Обрабатывает только те исключения, которые соответствуют типу, указаному в скобках после оператора catch. А вся информация об исключении помещается в переменную данного типа. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | try  {      int x = 5;      int y = x / 0;      Console.WriteLine($"Результат: {y}");  }  catch(DivideByZeroException ex)  {      Console.WriteLine($"Возникло исключение {ex.Message}");  } |

* Фактически этот случай аналогичен предыдущему за тем исключением, что здесь используется переменная. В данном случае в переменную ex, которая представляет тип DivideByZeroException, помещается информация о возникшем исключени. И с помощью свойства Message мы можем получить сообщение об ошибке.
* Если нам не нужна информация об исключении, то переменную можно не использовать как в предыдущем случае.

### Фильтры исключений

Фильтры исключений позволяют обрабатывать исключения в зависимости от определенных условий. Для их применения после выражения catch идет выражение **when**, после которого в скобках указывается условие:

|  |  |
| --- | --- |
| 1  2  3  4 | catch when(условие)  {    } |

В этом случае обработка исключения в блоке catch производится только в том случае, если условие в выражении when истинно. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int x = 1;  int y = 0;    try  {      int result1 = x / y;      int result2 = y / x;  }  catch (DivideByZeroException) when (y == 0)  {      Console.WriteLine("y не должен быть равен 0");  }  catch(DivideByZeroException ex)  {      Console.WriteLine(ex.Message);  } |

В данном случае будет выброшено исключение, так как y=0. Здесь два блока catch, и оба они обрабатывают исключения типа DivideByZeroException, то есть по сути все исключения, генерируемые при делении на ноль. Но поскольку для первого блока указано условие **y == 0**, то именно этот блок будет обрабатывать данное исключение - условие, указанное после оператора **when** возвращает true.

Противоположная ситуация:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int x = 0;  int y = 1;    try  {      int result1 = x / y;      int result2 = y / x;  }  catch (DivideByZeroException) when (y == 0)  {      Console.WriteLine("y не должен быть равен 0");  }  catch(DivideByZeroException ex)  {      Console.WriteLine(ex.Message);  } |

В данном случае будет выброшено исключение, так как x=0. Условие первого блока catch - **y == 0** теперь возвращает false. Поэтому CLR будет дальше искать соответствующие блоки catch далее и для обработки исключения выберет второй блок catch. В итоге если мы уберем второй блок catch, то исключение вобще не будет обрабатываться.

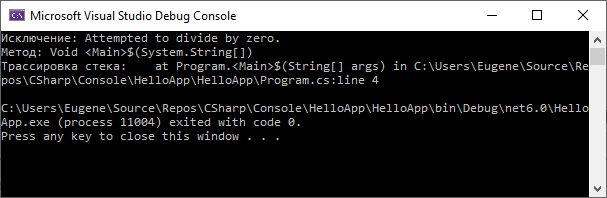
## Типы исключений. Класс Exception

Базовым для всех типов исключений является тип **Exception**. Этот тип определяет ряд свойств, с помощью которых можно получить информацию об исключении.

* **InnerException**: хранит информацию об исключении, которое послужило причиной текущего исключения
* **Message**: хранит сообщение об исключении, текст ошибки
* **Source**: хранит имя объекта или сборки, которое вызвало исключение
* **StackTrace**: возвращает строковое представление стека вызывов, которые привели к возникновению исключения
* **TargetSite**: возвращает метод, в котором и было вызвано исключение

Например, обработаем исключения типа Exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | try  {      int x = 5;      int y = x / 0;      Console.WriteLine($"Результат: {y}");  }  catch (Exception ex)  {      Console.WriteLine($"Исключение: {ex.Message}");      Console.WriteLine($"Метод: {ex.TargetSite}");      Console.WriteLine($"Трассировка стека: {ex.StackTrace}");  } |



Однако так как тип Exception является базовым типом для всех исключений, то выражение catch (Exception ex) будет обрабатывать все исключения, которые могут возникнуть.

Но также есть более специализированные типы исключений, которые предназначены для обработки каких-то определенных видов исключений. Их довольно много, я приведу лишь некоторые:

* **DivideByZeroException**: представляет исключение, которое генерируется при делении на ноль
* **ArgumentOutOfRangeException**: генерируется, если значение аргумента находится вне диапазона допустимых значений
* **ArgumentException**: генерируется, если в метод для параметра передается некорректное значение
* **IndexOutOfRangeException**: генерируется, если индекс элемента массива или коллекции находится вне диапазона допустимых значений
* **InvalidCastException**: генерируется при попытке произвести недопустимые преобразования типов
* **NullReferenceException**: генерируется при попытке обращения к объекту, который равен null (то есть по сути неопределен)

И при необходимости мы можем разграничить обработку различных типов исключений, включив дополнительные блоки catch:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | static void Main(string[] args)  {      try      {          int[] numbers = new int[4];          numbers[7] = 9;     // IndexOutOfRangeException            int x = 5;          int y = x / 0;  // DivideByZeroException          Console.WriteLine($"Результат: {y}");      }      catch (DivideByZeroException)      {          Console.WriteLine("Возникло исключение DivideByZeroException");      }      catch (IndexOutOfRangeException ex)      {          Console.WriteLine(ex.Message);      }        Console.Read();  } |

В данном случае блоки catch обрабатывают исключения типов IndexOutOfRangeException и DivideByZeroException. Когда в блоке try возникнет исключение, то CLR будет искать нужный блок catch для обработки исключения. Так, в данном случае на строке

|  |  |
| --- | --- |
| 1 | numbers[7] = 9; |

происходит обращение к 7-му элементу массива. Однако поскольку в массиве только 4 элемента, то мы получим исключение типа IndexOutOfRangeException. CLR найдет блок catch, который обрабатывает данное исключение, и передаст ему управление.

Следует отметить, что в данном случае в блоке try есть ситуация для генерации второго исключения - деление на ноль. Однако поскольку после генерации IndexOutOfRangeException управление переходит в соответствующий блок catch, то деление на ноль int y = x / 0 в принципе не будет выполняться, поэтому исключение типа DivideByZeroException никогда не будет сгенерировано.

Однако рассмотрим другую ситуацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | try  {      object obj = "you";      int num = (int)obj;     // System.InvalidCastException      Console.WriteLine($"Результат: {num}");  }  catch (DivideByZeroException)  {      Console.WriteLine("Возникло исключение DivideByZeroException");  }  catch (IndexOutOfRangeException)  {      Console.WriteLine("Возникло исключение IndexOutOfRangeException");  } |

В данном случае в блоке try генерируется исключение типа InvalidCastException, однако соответствующего блока catch для обработки данного исключения нет. Поэтому программа аварийно завершит свое выполнение.

Мы также можем определить для InvalidCastException свой блок catch, однако суть в том, что теоретически в коде могут быть сгенерированы самыи разные типы исключений. А определять для всех типов исключений блоки catch, если обработка исключений однотипна, не имеет смысла. И в этом случае мы можем определить блок catch для базового типа Exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | try  {      object obj = "you";      int num = (int)obj;     // System.InvalidCastException      Console.WriteLine($"Результат: {num}");  }  catch (DivideByZeroException)  {      Console.WriteLine("Возникло исключение DivideByZeroException");  }  catch (IndexOutOfRangeException)  {      Console.WriteLine("Возникло исключение IndexOutOfRangeException");  }  catch (Exception ex)  {      Console.WriteLine($"Исключение: {ex.Message}");  } |

И в данном случае блок catch (Exception ex){} будет обрабатывать все исключения кроме DivideByZeroException и IndexOutOfRangeException. При этом блоки catch для более общих, более базовых исключений следует помещать в конце - после блоков catch для более конкретный, специализированных типов. Так как CLR выбирает для обработки исключения первый блок catch, который соответствует типу сгенерированного исключения. Поэтому в данном случае сначала обрабатывается исключение DivideByZeroException и IndexOutOfRangeException, и только потом Exception (так как DivideByZeroException и IndexOutOfRangeException наследуется от класса Exception).

## Генерация исключения и оператор throw

Обычно система сама генерирует исключения при определенных ситуациях, например, при делении числа на ноль. Но язык C# также позволяет генерировать исключения вручную с помощью оператора **throw**. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения.

Например, в нашей программе происходит ввод имени пользователя, и мы хотим, чтобы, если длина имени меньше 2 символов, то возникало исключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | try  {      Console.Write("Введите имя: ");      string? name = Console.ReadLine();      if (name== null || name.Length < 2)      {          throw new Exception("Длина имени меньше 2 символов");      }      else      {          Console.WriteLine($"Ваше имя: {name}");      }  }  catch (Exception e)  {      Console.WriteLine($"Ошибка: {e.Message}");  } |

После оператора **throw** указывается объект исключения, через конструктор которого мы можем передать сообщение об ошибке. Естественно вместо типа Exception мы можем использовать объект любого другого типа исключений.

Затем в блоке catch сгенерированное нами исключение будет обработано.

Подобным образом мы можем генерировать исключения в любом месте программы. Но существует также и другая форма использования оператора throw, когда после данного оператора не указывается объект исключения. В подобном виде оператор throw может использоваться только в блоке catch:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | try  {      try      {          Console.Write("Введите имя: ");          string? name = Console.ReadLine();          if (name == null || name.Length < 2)          {              throw new Exception("Длина имени меньше 2 символов");          }          else          {              Console.WriteLine($"Ваше имя: {name}");          }      }      catch (Exception e)      {          Console.WriteLine($"Ошибка: {e.Message}");          throw;      }  }  catch (Exception ex)  {      Console.WriteLine(ex.Message);  } |

В данном случае при вводе имени с длиной меньше 2 символов возникнет исключение, которое будет обработано внутренним блоком catch. Однако поскольку в этом блоке используется оператор throw, то исключение будет передано дальше внешнему блоку catch, который получит то же смое исключение и выведет то же самое сообщение на консоль.

## Создание классов исключений

Если нас не устраивают встроенные типы исключений, то мы можем создать свои типы. Базовым классом для всех исключений является класс Exception, соответственно для создания своих типов мы можем унаследовать данный класс.

Допустим, у нас в программе будет ограничение по возрасту:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | try  {      Person person = new Person { Name = "Tom", Age = 17 };  }  catch (Exception ex)  {      Console.WriteLine($"Ошибка: {ex.Message}");  }    class Person  {      private int age;      public string Name { get; set; } = "";      public int Age      {          get => age;          set          {              if (value < 18)                  throw new Exception("Лицам до 18 регистрация запрещена");              else                  age = value;          }      }  } |

В классе Person при установке возраста происходит проверка, и если возраст меньше 18, то выбрасывается исключение. Класс Exception принимает в конструкторе в качестве параметра строку, которое затем передается в его свойство Message.

Но иногда удобнее использовать свои классы исключений. Например, в какой-то ситуации мы хотим обработать определенным образом только те исключения, которые относятся к классу Person. Для этих целей мы можем сделать специальный класс PersonException:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class PersonException : Exception  {      public PersonException(string message)          : base(message) { }  } |

По сути класс кроме пустого конструктора ничего не имеет, и то в конструкторе мы просто обращаемся к конструктору базового класса Exception, передавая в него строку message. Но теперь мы можем изменить класс Person, чтобы он выбрасывал исключение именно этого типа и соответственно в основной программе обрабатывать это исключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | try  {      Person person = new Person { Name = "Tom", Age = 17 };  }  catch (PersonException ex)  {      Console.WriteLine($"Ошибка: {ex.Message}");  }    class Person  {      private int age;      public string Name { get; set; } = "";      public int Age      {          get => age;          set          {              if (value < 18)                  throw new PersonException("Лицам до 18 регистрация запрещена");              else                  age = value;          }      }  } |

Однако необязательно наследовать свой класс исключений именно от типа Exception, можно взять какой-нибудь другой производный тип. Например, в данном случае мы можем взять тип ArgumentException, который представляет исключение, генерируемое в результате передачи аргументу метода некорректного значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class PersonException : ArgumentException  {      public PersonException(string message)          : base(message)      { }  } |

Каждый тип исключений может определять какие-то свои свойства. Например, в данном случае мы можем определить в классе свойство для хранения устанавливаемого значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class PersonException : ArgumentException  {      public int Value { get;}      public PersonException(string message, int val)          : base(message)      {          Value = val;      }  } |

В конструкторе класса мы устанавливаем это свойство и при обработке исключения мы его можем получить:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | try  {      Person person = new Person { Name = "Tom", Age = 17 };  }  catch (PersonException ex)  {      Console.WriteLine($"Ошибка: {ex.Message}");      Console.WriteLine($"Некорректное значение: {ex.Value}");  }    class Person  {      private int age;      public string Name { get; set; } = "";      public int Age      {          get => age;          set          {              if (value < 18)                  throw new PersonException("Лицам до 18 регистрация запрещена", value);              else                  age = value;          }      }  } |

И в данном случае мы получим следующий консольный вывод:

## Поиск блока catch при обработке исключений

Если код, который вызывает исключение, не размещен в блоке try или помещен в конструкцию try..catch, которая не содержит соответствующего блока catch для обработки возникшего исключения, то система производит поиск соответствующего обработчика исключения в стеке вызовов.

Например, рассмотрим следующую программу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | try  {      TestClass.Method1();  }  catch (DivideByZeroException ex)  {      Console.WriteLine($"Catch в Main : {ex.Message}");  }  finally  {      Console.WriteLine("Блок finally в Main");  }  Console.WriteLine("Конец метода Main");    class TestClass  {      public static void Method1()      {          try          {              Method2();          }          catch (IndexOutOfRangeException ex)          {              Console.WriteLine($"Catch в Method1 : {ex.Message}");          }          finally          {              Console.WriteLine("Блок finally в Method1");          }          Console.WriteLine("Конец метода Method1");      }      static void Method2()      {          try          {              int x = 8;              int y = x / 0;          }          finally          {              Console.WriteLine("Блок finally в Method2");          }          Console.WriteLine("Конец метода Method2");      }  } |

В данном случае стек вызовов выглядит следующим образом: метод Main вызывает метод Method1, который, в свою очередь, вызывает метод Method2. И в методе Method2 генерируется исключение DivideByZeroException. Визуально стек вызовов можно представить следующим образом:

Внизу стека метод Main, с которого началось выполнение, и на самом верху метод Method2.

Что будет происходить в данном случае при генерации исключения?

1. Метод Main вызывает метод Method1, а тот вызывает метод Method2, в котором генерируется исключение DivideByZeroException.
2. Система видит, что код, который вызывал исключение, помещен в конструкцию try..catch

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | try  {      int x = 8;      int y = x / 0;  }  finally  {      Console.WriteLine("Блок finally в Method2");  } |

1. Система ищет в этой конструкции блок catch, который обрабатывает исключение DivideByZeroException. Однако такого блока catch нет.
2. Система опускается в стеке вызовов в метод Method1, который вызывал Method2. Здесь вызов Method2 помещен в конструкцию try..catch

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | try  {      Method2();  }  catch (IndexOutOfRangeException ex)  {      Console.WriteLine($"Catch в Method1 : {ex.Message}");  }  finally  {      Console.WriteLine("Блок finally в Method1");  } |

1. Система также ищет в этой конструкции блок catch, который обрабатывает исключение DivideByZeroException. Однако здесь также подобный блок catch отсутствует.
2. Система далее опускается в стеке вызовов в метод Main, который вызывал Method1. Здесь вызов Method1 помещен в конструкцию try..catch

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | try  {      TestClass.Method1();  }  catch (DivideByZeroException ex)  {      Console.WriteLine($"Catch в Main : {ex.Message}");  }  finally  {      Console.WriteLine("Блок finally в Main");  } |

1. Система снова ищет в этой конструкции блок catch, который обрабатывает исключение DivideByZeroException. И в данном случае такой блок найден.
2. Система наконец нашла нужный блок catch в методе Main, для обработки исключения, которое возникло в методе Method2 - то есть к начальному методу, где непосредственно возникло исключение. Но пока данный блок catch НЕ выполняется. Система поднимается обратно по стеку вызовов в самый верх в метод Method2 и выполняет в нем блок finally:

|  |  |
| --- | --- |
| 1  2  3  4 | finally  {      Console.WriteLine("Блок finally в Method2");  } |

1. Далее система возвращается по стеку вызовов вниз в метод Method1 и выполняет в нем блок finally:

|  |  |
| --- | --- |
| 1  2  3  4 | finally  {      Console.WriteLine("Блок finally в Method1");  } |

1. Затем система переходит по стеку вызовов вниз в метод Main и выполняет в нем найденный блок catch и последующий блок finally:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | catch (DivideByZeroException ex)  {      Console.WriteLine($"Catch в Main : {ex.Message}");  }  finally  {      Console.WriteLine("Блок finally в Main");  } |

1. Далее выполняется код, который идет в методе Main после конструкции try..catch:

|  |  |
| --- | --- |
| 1 | Console.WriteLine("Конец метода Main"); |

1. Стоит отметить, что код, который идет после конструкции try...catch в методах Method1 и Method2, не выполняется, потому что обработчик исключения найден именно в методе Main.

Консольный вывод программы:

Блок finally в Method2

Блок finally в Method1

Catch в Main: Attempted to divide by zero

Блок finally в Main

Конец метода Main