# Лекция 1. Основа языка C#. Создание приложений WPF

## Типы данных в C#

Под типом данных понимается описание данных определенного вида, для которых известен их способ представления в памяти (формат), следующие из него размерность и диапазон значений, а также определен набор операций. Тип данных определяет внутреннее представление данных, множество значений, которые может принимать объект.

В языке C# есть следующие базовые типы данных:

* **bool**: хранит значение true или false (логические литералы).

Представлен системным типом System.Boolean

|  |
| --- |
| bool needShow = true;  bool isFinish = false; |

* **byte**: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte

|  |
| --- |
| byte bit1 = 241;  byte bit2 = 3; |

* **sbyte**: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte

|  |
| --- |
| sbyte bit1 = 92;  sbyte bit2 = -75; |

* **short**: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16

|  |
| --- |
| short sh1 = -8;  short sh2 = 6102; |

* **ushort**: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16

|  |
| --- |
| ushort sh1 = 3671;  ushort sh2 = 7; |

* **int**: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:

|  |
| --- |
| int count = 10; |

* **uint**: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32

|  |
| --- |
| uint row = 10; |

* **long**: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом System.Int64

|  |
| --- |
| long n = -1983850; |

* **ulong**: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64

|  |
| --- |
| ulong b = 746421710; |

* **float**: хранит число с плавающей точкой от -3.4\*1038 до 3.4\*1038 и занимает 4 байта. Представлен системным типом System.Single

|  |
| --- |
| float position = 4.56; |

* **double**: хранит число с плавающей точкой от ±5.0 \* 10324  до ±1.7\*10308 и занимает 8 байта. Представлен системным типом System.Double

|  |
| --- |
| Double pos = 75.2; |

* **char**: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом System.Char. Этому типу соответствуют символьные литералы:

|  |
| --- |
| char a = 'A';  char b = '\x5A';  char c = '\u0420'; |

* string: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.

|  |
| --- |
| string hello = "Hello";  string word = "world"; |

* **object**: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом System.Object, который является базовым для всех других типов и классов .NET.

|  |
| --- |
| object a = 78;  object b = 3.14;  object c = "name"; |

При объявлении переменных мы явно указывали ее тип, например, int a = 10; Компилятор при запуске уже знает, что a хранит целочисленное значение. Однако мы можем использовать и неявную типизацию:

|  |
| --- |
| var name = "Mike";  var age = 18; |

Для неявной типизации вместо названия типа данных используется ключевое слово var. Затем уже при компиляции компилятор сам выводит тип данных исходя из присвоенного значения. Так как по умолчанию все целочисленные значения рассматриваются как значения типа int, то поэтому в итоге переменная age будет иметь тип int. Аналогично переменной name присваивается строка, поэтому эта переменная будет иметь тип string

Эти переменные подобны обычным, однако они имеют некоторые ограничения.

Во-первых, мы не можем сначала объявить неявно типизируемую переменную, а затем инициализировать:

|  |
| --- |
| // этот код работает  int a;  a = 34;    // этот код не работает  var c;  c = 68; |

Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной null:

|  |
| --- |
| // этот код не работает  var c = null; |

Так как значение null, то компилятор не сможет вывести тип данных.

Все типы данных можно разделить на типы значений, еще называемые **значимыми** типами, (value types) и **ссылочные** типы (reference types). Важно понимать между ними различия.

Типы значений:

* Целочисленные типы (byte, sbyte, short, ushort, int, uint, long, ulong)
* Типы с плавающей запятой (float, double)
* Тип decimal
* Тип bool
* Тип char
* Перечисления enum
* Структуры (struct)

Ссылочные типы:

* Тип object
* Тип string
* Классы (class)
* Интерфейсы (interface)
* Делегаты (delegate)

Чтобы понять их различия, необходимо знать организацию памяти в .NET.

## Организация памяти

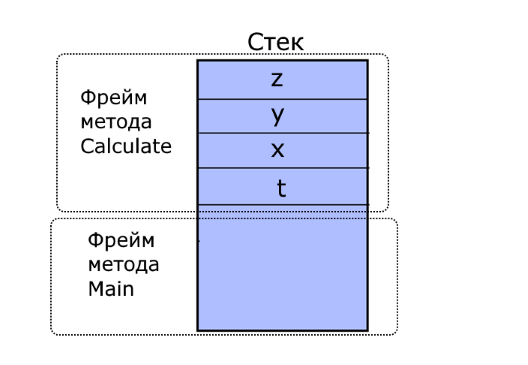
Память делится на два типа: стек и куча (heap). Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу вверх: каждый новый добавляемый элемент помещается поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве.

Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.

Например:

|  |
| --- |
| class Program  {      static void Main(string[] args)      {          Calculate(5);      }        static void Calculate(int t)      {          int x = 6;          int y = 7;          int z = y + t;      }  } |

Пи запуске такой программы в стеке будут определяться два фрейма - для метода Main (так как он вызывается при запуске программы) и для метода Calculate:



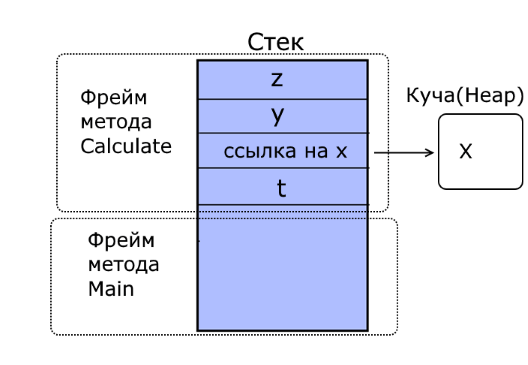
При вызове этого метода Calculate в его фрейм в стеке будут помещаться значения t, x, y и z. Они определяются в контексте данного метода. Когда метод отработает, область памяти, которая выделялась под стек, впоследствии может быть использована другими методами. Причем если параметр или переменная метода представляет тип значений, то в стеке будет храниться непосредсвенное значение этого параметра или переменной. Например, в данном случае переменные и параметр метода Calculate представляют значимый тип - тип int, поэтому в стеке будут храниться их числовые значения.

Ссылочные типы хранятся в куче или хипе, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.

При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, условно удаляет этот объект и очищает память - фактически помечает, что данный сегмент памяти может быть использован для хранения других данных.

Изменим метод Calculate следующим образом:

|  |
| --- |
| static void Calculate(int t)  {      object x = 9;      int y = 5;      int z = y + t;  } |

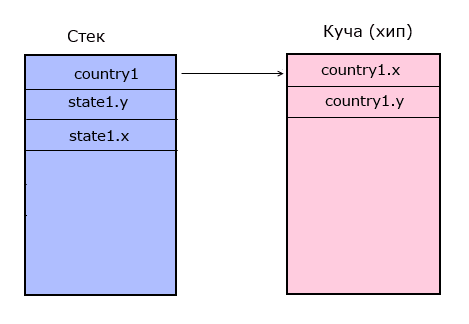


## Составные типы данных

Теперь рассмотрим ситуацию, когда тип значений и ссылочный тип представляют составные типы - структуру и класс:

|  |
| --- |
| // State - структура, ее данные размещены в стеке  State state1 = new State();  // Country - класс, в стек помещается ссылка на адрес в хипе  // а в хипе располагаются все данные объекта country1  Country country1 = new Country();  struct State  {      public int x;      public int y;  }  class Country  {      public int x;      public int y;  } |

Здесь в методе Main в стеке выделяется память для объекта state1. Далее в стеке создается ссылка для объекта country1 (Country country1), а с помощью вызова конструктора с ключевым словом new выделяется место в хипе (new Country()). Ссылка в стеке для объекта country1 будет представлять адрес на место в хипе, по которому размещен данный объект.

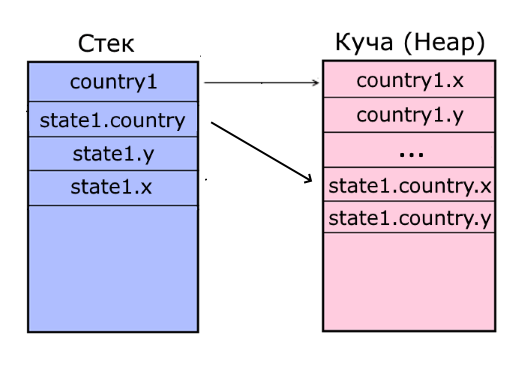


Таким образом, в стеке окажутся все поля структуры state1 и ссылка на объект country1 в хипе.

Но, допустим, в структуре State также определена переменная ссылочного типа Country. Где она будет хранить свое значение, если она определена в типе значений?

|  |
| --- |
| State state1 = new State();  Country country1 = new Country();    struct State  {      public int x;      public int y;      public Country country;      public State()      {          x = 0;          y = 0;          country = new Country();      }  }  class Country  {      public int x;      public int y;  } |

Значение переменной state1.country также будет храниться в куче, так как эта переменная представляет ссылочный тип:



## Копирование значений

Тип данных надо учитывать при копировании значений. При присвоении данных объекту значимого типа он получает копию данных. При присвоении данных объекту ссылочного типа он получает не копию объекта, а ссылку на этот объект в хипе. Например:

|  |
| --- |
| State state1 = new State(); // Структура State  State state2 = new State();  state2.x = 1;  state2.y = 2;  state1 = state2;  state2.x = 5; // state1.x=1 по-прежнему  Console.WriteLine(state1.x); // 1  Console.WriteLine(state2.x); // 5    Country country1 = new Country(); // Класс Country  Country country2 = new Country();  country2.x = 1;  country2.y = 4;  country1 = country2;  country2.x = 7; // теперь и country1.x = 7, так как обе ссылки и //country1 и country2 указывают на один объект в хипе  Console.WriteLine(country1.x); // 7  Console.WriteLine(country2.x); // 7 |

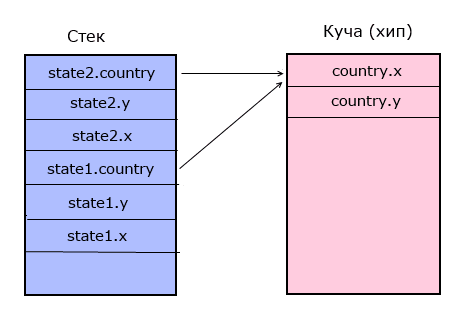
Так как state1 - структура, то при присвоении state1 = state2 она получает копию структуры state2. А объект класса country1 при присвоении country1 = country2; получает ссылку на тот же объект, на который указывает country2. Поэтому с изменением country2, так же будет меняться и country1.

## Ссылочные типы внутри типов значений

Теперь рассмотрим более изощренный пример, когда внутри структуры у нас может быть переменная ссылочного типа, например, какого-нибудь класса:

|  |
| --- |
| State state1 = new State();  State state2 = new State();    state2.country = new Country();  state2.country.x = 5;  state1 = state2;  state2.country.x = 8; // теперь и state1.country.x=8, так как  //state1.country и state2.country указывают на  //один объект в хипе  Console.WriteLine(state1.country.x); // 8  Console.WriteLine(state2.country.x); // 8    struct State  {      public int x;      public int y;      public Country country;      public State()      {          x = 0;          y = 0;          country = new Country(); // выделение памяти для объекта  //Country      }  }  class Country  {      public int x;      public int y;  } |

Переменные ссылочных типов в структурах также сохраняют в стеке ссылку на объект в хипе. И при присвоении двух структур state1 = state2; структура state1 также получит ссылку на объект country в хипе. Поэтому изменение state2.country повлечет за собой также изменение state1.country.



## Опасность ссылочных типов

* В переменных ссылочных типов хранятся ссылки на их данные (объекты). Две переменные ссылочного типа могут ссылаться на один и тот же объект, поэтому операции над одной переменной могут затрагивать объект, на который ссылается другая переменная.
* Присваивание переменных ссылочного типа влечет копирование не смого объекта, а его ссылки.

## Методы

Если переменные хранят некоторые значения, то методы содержат собой набор инструкций, которые выполняют определенные действия. По сути метод - это именованный блок кода, который выполняет некоторые действия.

Общее определение методов выглядит следующим образом:

|  |
| --- |
| [модификаторы] тип\_возвращаемого\_значения название\_метода ([параметры])  {      // тело метода  } |

Модификаторы и параметры необязательны.

|  |
| --- |
| void SayHello()  {      Console.WriteLine("Hello");  } |

Здесь определен метод SayHello, который выводит некоторое сообщение. К названиям методов предъявляются в принципе те же требования, что и к названиям переменных. Однако, как правило, названия методов начинаются с большой буквы. Перед названием метода идет возвращаемый тип данных. Здесь это тип void, который указывает, что фактически ничего не возвращает, он просто производит некоторые действия. После названия метода в скобках идет перечисление параметров. Но в данном случае скобки пустые, что означает, что метод не принимает никаких параметров. После списка параметров в круглых скобках идет блок кода, который представляет набор выполняемых методом инструкций. В данном случае блок метода SayHello содержит только одну инструкцию, которая выводит строку в консоль.

Вызов метода осуществляется следующим образом:

|  |
| --- |
| class Program  {      static void Main(string[] args)      {          SayHello(); // вызов метода      }        void SayHello()  {  Console.WriteLine("Hello");  }  } |

### Параметры метода

Параметры позволяют передать в метод некоторые входные данные. Параметры определяются через запятую в скобках после названия метода в виде:

|  |
| --- |
| тип\_метода имя\_метода (тип\_параметра1 параметр1, тип\_параметра2 параметр2, ...)  {  // действия метода  } |

Определение параметра состоит из двух частей: сначала идет тип параметра и затем его имя.

|  |
| --- |
| void PrintMessage(string text)  {      Console.WriteLine(text);  }  string str = "Hello my friend!";  PrintMessage("This is a Message");  PrintMessage(str); |

### Возвращаемое значение метода

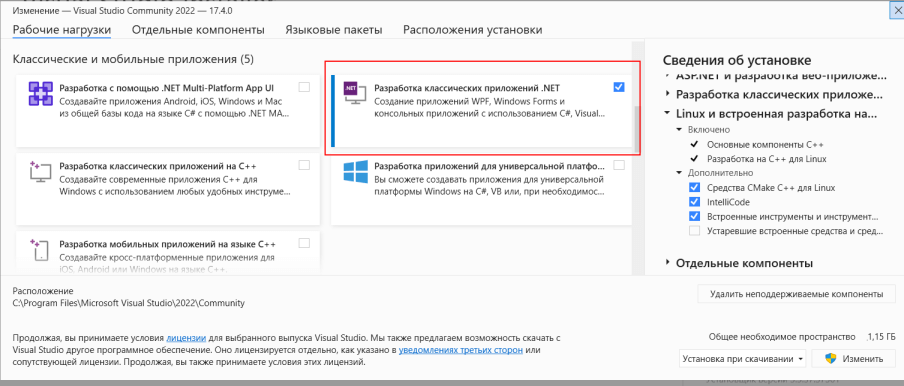
Метод может возвращать значение, какой-либо результат. Если перед названием метода стоит тип void, то такой метод не возвращает никакого значения. Он просто выполняет некоторые действия.

Для того чтобы метод вернул какое-то значение используется оператор **return**, после которого идет возвращаемое значение:

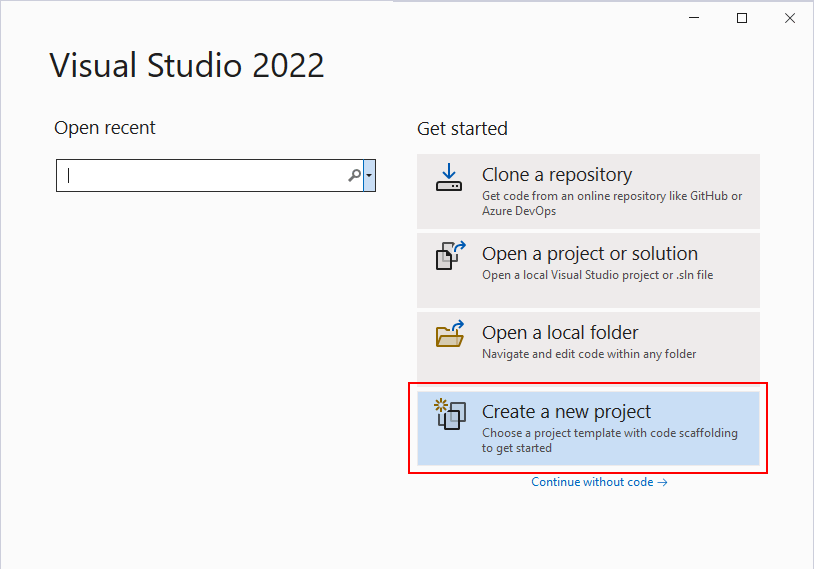
|  |
| --- |
| // метод возвращает сумму двух чисел  int GetSum()  {  int a = 5;  int b = 7;  return a + b;  } |

## Создание приложений WPF

Для создания приложений с помощью технологии WPF можно использовать среду разработки Visual Studio. Чтобы добавить в Visual Studio поддержку проектов для WPF и C# в программе установки среди рабочих нагрузок нужно выбрать только пункт Разработка классических приложений .NET. Можно выбрать и больше опций или вообще все опции, однако стоит учитывать свободный размер на жестком диске - чем больше опций будет выбрано, соответственно тем больше места на диске будет занято.



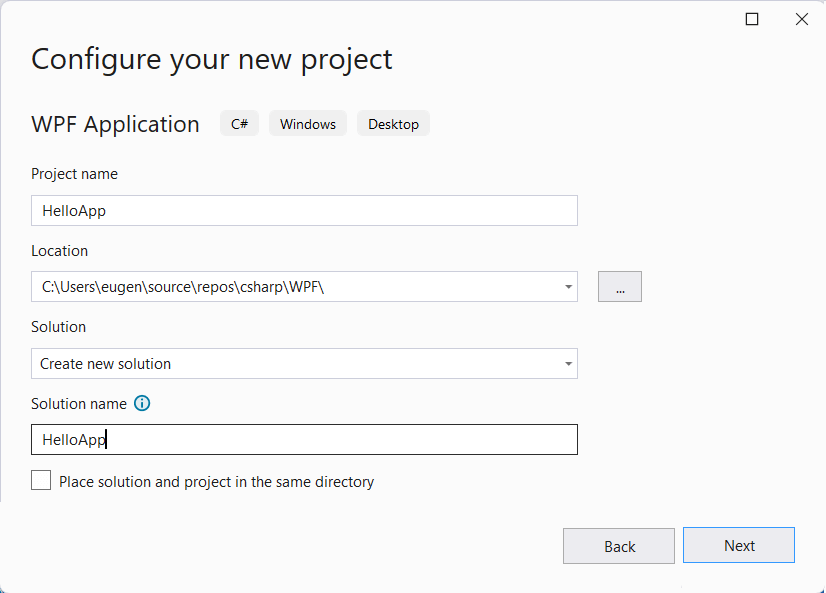
После установки среды и всех ее компонентов, запустим Visual Studio и создадим проект графического приложения. На стартовом экране выберем **Create a new project** (Создать новый проект).



На следующем окне в качестве типа проекта выберем WPF Application:

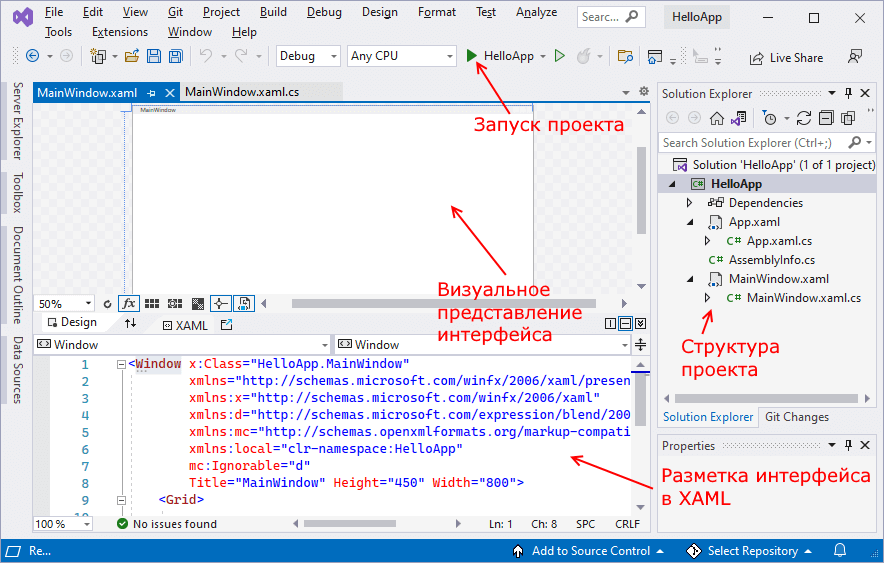


Далее на следующем этапе нам будет предложено указать имя проекта и каталог, где будет располагаться проект.



В поле Project Name дадим проекту какое-либо название. В данном случае это HelloApp.

После этого Visual Studio откроет наш проект с созданными по умолчанию файлами:



### Структура проекта

Справа находится окно Solution Explorer, в котором можно увидеть структуру нашего проекта. В данном случае у нас сгенерированная по умолчанию структура, которая аналогична той, что создается с помощью .NET CLI:

* **Dependencies** - это узел содержит сборки dll, которые добавлены в проект по умолчанию. Эти сборки как раз содержат классы библиотеки .NET, которые будет использовать C#
* **App.xaml** задает ресурсы приложения и ряд конфигурационных настроек в виде кода XAML. В частности, в файле App.xaml задается файл окна программы, которое будет открываться при запуске приложения. Если вы откроете этот файл, то можете найти в нем строку StartupUri="MainWindow.xaml" - то есть в данном случае, когда мы запустим приложение, будет создаваться интерфейс из файла MainWindow.xaml.
* **App.xaml.cs** - это файл кода на C#, связанный с файл App.xaml, который также позволяет задать ряд общих ресурсов и общую логику для приложения, но в вид кода C#.
* **AssemblyInfo.cs** содержит информацию о создаваемой в процессе компиляции сборке
* **MainWindow.xaml** представляет визуальный интерфейс окна приложения в виде кода XAML.
* **MainWindow.xaml.cs** - это файл логики кода на C#, связанный с окном MainWindow.xaml.

По умолчанию эти файлы открыты в текстовом редакторе Visual Studio. Причем файл MainWindow.xaml имеет два представления: визуальное - в режиме WYSIWIG отображает весь графический интерфейс данного окна приложения, и под ним декларативное объявление интерфейса в XAML. Если мы изменим декларативную разметку, например, определим там кнопку, то эти изменения отображаться в визуальном представлении. Таким образом, мы сможем сразу же получить представление об интерфейсе окна приложения.

### Запуск проекта

Чтобы запустить приложение в режиме отладки, нажмем на клавишу F5 или на зеленую стрелочку на панели Visual Studio. И после этого запустится пустое окно по умолчанию. После запуска приложения студия компилирует его в файл с расширением exe. Найти данный файл можно, зайдя в папку проекта и далее в каталог \bin\Debug\net7.0-windows

# Лекция 2. Введение в ООП

Методология объектно-ориентированного программирования пришла на смену процедурной или алгоритмической организации структуры программного кода, когда стало очевидно, что традиционные методы процедурного программирования не способны справиться ни с растущей сложностью программ и их разработки, ни с повышением их надежности. Во второй половине 80-х годов возникла настоятельная потребность в новой методологии программирования, которая бы позволила решить весь этот комплекс проблем. Такой методологией стало объектно-ориентированное программирование (ООП).

**Объектно-ориентированное программирование** (Object Oriented Programming) - совокупность принципов, технологий, а также инструментальных средств для создания программных систем на основе архитектуры взаимодействия объектов.

## Основные понятия

**Объект** – это некоторая конкретная сущность (предмет, явление). Каждый объект обладает состоянием, поведением и уникальностью.

**Класс** – это описание объекта, его структуры и поведения. Каждый объект обязательно принадлежит некоторому классу. Если объект принадлежит некоторому классу, то говорят, что он является экземпляром класса.

**Атрибуты** (поля класса) - это переменные, определенные на уровне класса.

**Состояние** – это набор значений характеристик объекта в данный момент времени. Состояние объектов задается при помощи атрибутов. Состояние может меняться под внешним воздействием, либо сам объект может менять свое состояние.

**Поведение** – это действия, которые может совершать объект и как объект может реагировать на воздействие со стороны других объектов. Поведение объекта задается при помощи методов.

## Классы

Класс – модель для создания объектов определённого типа, описывающая их структуру (набор полей и их начальное состояние) и определяющая алгоритмы (функции или методы) для работы с этими объектами. Класс является одним из ключевых понятий в ООП. На практике ООП сводится к созданию некоторого количества классов, включая интерфейс и реализацию, и последующему их использованию.

### Определение класса

Для объявления класса в программе используется ключевое слово **class**. Имя класса указывается с большой буквы.

|  |
| --- |
| class Point2D  {  // члены класса: поля и методы  } |

Каждый класс в C# может содержать поля (переменные, атрибуты) и методы (функции). Поля определяют структуру класса, а методы – поведение класса.

Ниже приведен пример класса двумерной точки (Point2D), содержащий координату x и y, конструктор и метод вычисление расстояния до другой точки GetDistance.

|  |
| --- |
| class Point2D  {  private double x;  private double y;  public Point2D(double x, double y)  {  this.x = x;  this.y = y;  }  public double GetDistance(Point2D otherPoint)  {  return Math.Sqrt(Math.Pow(x - otherPoint.x, 2) + Math.Pow(y - otherPoint.y, 2));  }  } |

x и y - это поля класса имеющие вещественный тип (double).

public Point2D(double x, double y) - конструктор класса.

public double GetDistance(Point2D otherPoint) - метод класса, который возвращает значение типа double. Метод GetDistance принимает один параметр типа Point2D.

### Добавление класса

Обычно классы помещаются в отдельные файлы. Нередко для одного класса предназначен один файл. Например, добавим новый файл, который назовем Person.cs и в котором определим следующий код:

|  |
| --- |
| class Person  {      public string name = "Undefined";      public void Print()      {          Console.WriteLine($"Person {name}");      }  } |

### Конструкторы класса

**Конструктор** – специальный метод, который позволяет создать и инициализировать экземпляр класса.

Он называется также как сам класс. Конструктор нельзя вызвать явно, он вызывается автоматически при создании объекта при помощи оператора **new**.

При объявлении метода-конструктора не указывается возвращаемый тип. Конструктор ничего не возвращает. Также конструктор может как не иметь аргументов, так и принимать несколько аргументов. Если при объявлении класса вообще не создавать конструктор, то компилятор сам генерирует конструктор по умолчанию (он без аргументов). Если в классе создать конструктор с аргументами, то компилятор не создает конструктор по умолчанию. Класс может иметь несколько конструкторов.

Ниже приведен пример класса Point2D с двумя конструкторами:

|  |
| --- |
| class Point2D  {  private double x;  private double y;  public Point2D(double x, double y)  {  this.x = x;  this.y = y;  }  public Point2D() { }  } |

### Ключевое слово this

Ключевое слово **this** представляет ссылку на текущий экземпляр класса.

|  |
| --- |
| public Point2D(double x, double y) {  this.x = x;  this.y = y;  } |

В классе Point2D объявлены два поля x и y типа double. Конструктор класса также имеет два аргумента x и y типа double. Чтобы компилятор понял, к какому x мы обращаемся или какой x мы имеем в виду, мы можем использовать ключевое слово this. this.x означает в данном случае, что мы имеем в виду поле x класса Point2D.

### Инициализаторы объектов и их особенности

Для инициализации объектов классов можно применять инициализаторы. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

|  |
| --- |
| public MainWindow()  {  Point2D point = new Point2D { x = 4, y = 6};  } |

Можно выделить следующие особенности инициализаторов:

* С помощью инициализатора мы можем установить значения только доступных извне класса полей и свойств объекта. Поля должны иметь модификатор доступа public, чтобы они были доступны из любой части программы.
* Инициализатор выполняется после конструктора, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, заменяются значениями из инициализатора.
* Инициализаторы удобно применять, когда поле или свойство класса представляет другой класс.

### Создание объекта класса

Создание экземпляра (объекта) класса происходит следующим образом:

|  |
| --- |
| public MainWindow()  {  Point2D point = new Point2D(1, 10); // создали объект point классса Point2D  Person tom = new Person(); // создали объект tom класса Person  tom.name = "Tom";  tom.Print();  } |

### Обращение к полям и методам классов

Обращение к полям и методам объекта осуществляется через оператор «точка». При обращении к методу внутри скобок перечисляются параметры метода (или оставляются пустые скобки).

|  |
| --- |
| public MainWindow()  {  Point2D point = new Point2D(1, 10);  point.ShiftX(3); // обращение к методу ShiftX класса Point2D  } |

Для членов класса могут иметься разные права доступа. Они задаются при объявлении класса при помощи модификаторов видимости, например, public и private. Если прав недостаточно, то обращение к члену класса приведет к ошибке компиляции.

|  |
| --- |
| public MainWindow()  {  Point2D point = new Point2D(1, 10);  point.ShiftX(3); // допустимый вызов метода  point.x = 7; // недопустимое обращение к полю  } |

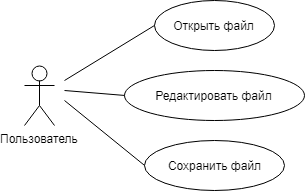
## UML диаграммы

**UML** (англ. Unified Modeling Language — унифицированный язык моделирования) — язык графического описания для объектного моделирования в области разработки программного обеспечения, для моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

В UML используются следующие виды диаграмм:

* **Диаграмма прецедентов** представляет собой простой способ визуального представления основных возможностей разрабатываемого программного обеспечения или процесса.

Пример диаграммы прецедентов, описывающий функции текстового редактора:

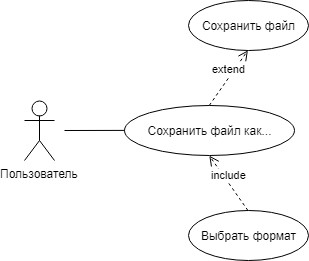


Для описания функций используются действующее лицо (обозначенное человекоподобной фигурой), прецеденты (обозначенные овалами) и ассоциативные связи.

Следует отметить, что действующим лицом может являться не только пользователь, но и программа, программист, специалист службы технической поддержки и т.д. А на одной диаграмме может присутствовать множество действующих лиц.

Помимо ассоциативных связей существуют:

* направленные ассоциации (линия со стрелкой) – в явной форме указывают характер отношений между прецедентами;
* зависимости – указывают на зависимости между прецедентами;
* обобщения – указывают на вхождение частного прецедента в более общий.

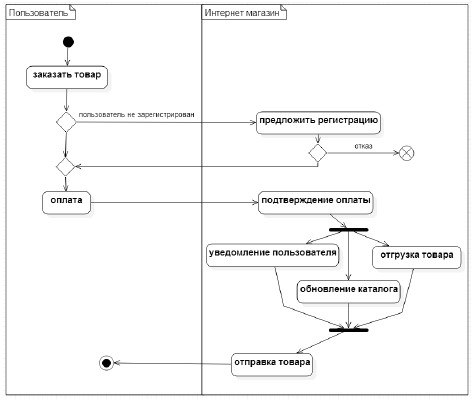


В данном примере описан прецедент расширенной функции «сохранить файл». Для указания того, что сохранение файла с выбором формата содержит в себе выбор формата и является расширением стандартной функции сохранения файла, используется отношение зависимости.



На данной диаграмме показано, что ввод текста является частным случаем редактирования файла, а ввод связан с выбором шрифта (без указания типа связи). Следует помнить, что диаграмма прецедентов должна описывать общие функциональные возможности разрабатываемой программы или системы, а также фундаментальные зависимости между прецедентами, не акцентируя внимания на деталях реализации процессов и действующих лиц.

* **Диаграмма деятельности** является альтернативой представлению процессов в виде блок-схем и используется для описания последовательности действий и выборов. Состоит из следующих элементов:



* начало процесса – обозначает старт описываемого процесса, может не совпадать с началом работы программы или глобального процесса;
* действие – содержит в себе описание действий на текущем этапе выполнения алгоритма;
* решение – как и на блок-схемах обозначается ромбом, однако не содержит в себе текста. Текст условий ветвления указывается на исходящих из решения управляющих потоках;
* управляющий поток – указывает последовательность выполнения действий;
* разделение – начало блока независимых операций;
* соединение – завершение блока независимых операций;
* завершение процесса – окончание описываемого процесса, может не совпадать с окончанием работы программы или глобального процесса.

Выше изображен пример описания процесса заказа товара через интернет-магазин при помощи диаграммы деятельности. На представленной диаграмме действия расположены в двух областях, обозначающих действующих лиц, участвующих в процессе заказа. Подобное представление не является обязательным при составлении диаграммы деятельности, однако, в дальнейшем, может упростить создание диаграммы последовательности. Предполагается что операции по уведомлению пользователя, обновлению каталога и отгрузки товара, могут осуществляться одновременно, поскольку независимы друг от друга. Действие «отправка товара» является обобщённым и может быть представлено в виде отдельной диаграммы, если это необходимо для понимания моделируемых процессов. Следует помнить, что диаграмма деятельности должна описывать последовательность действий и выборов в процессе выполнения некоего процесса, не акцентируя внимания на классах, полях и методах.

* **Диаграмма классов** - диаграмма языка моделирования UML, демонстрирующая общую структуру иерархии классов системы, их коопераций, атрибутов, методов, интерфейсов и взаимосвязей между ними. Широко применяется для документирования и визуализации информационных систем.

Основной структурной сущностью в диаграммах классов является класс. **Класс** – это описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой. Графически класс изображается в виде прямоугольника, разделенного на 3 блока горизонтальными линиями:

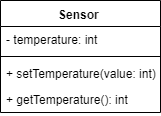
* имя класса;
* атрибуты (свойства) класса;
* операции (методы) класса.

Каждый класс должен обладать именем, отличающим его от других классов. Обычно используются краткие имена, отражающие описываемую сущность. Каждое слово в имени класса традиционно пишут с заглавной буквы, например Sensor или Датчик.

**Атрибут (свойство)** представляет некоторое свойство моделируемой сущности, которым обладают все объекты данного класса. Класс может иметь любое число атрибутов или не иметь ни одного. В последнем случае блок атрибутов оставляют пустым.

**Операция (метод)** – это реализация метода класса. Класс может иметь любое число операций либо не иметь ни одной. Часто вызов операции объекта изменяет его атрибуты. Класс может иметь любое число операций либо не иметь ни одной. Графически операции представлены в нижнем блоке описания класса. Можно специфицировать операцию, устанавливая ее сигнатуру, включающую имя, тип и значение по умолчанию всех параметров, а применительно к функциям – тип возвращаемого значения.

Изображая класс, не обязательно показывать сразу все его атрибуты и операции. Для конкретного представления, как правило, существенна только часть атрибутов и операций класса. В силу этих причин допускается упрощенное представление класса, то есть для графического представления выбираются только некоторые из его атрибутов. Если помимо указанных существуют другие атрибуты и операции, вы даете это понять, завершая каждый список многоточием.



**Взаимосвязь (отношение)** - это особый тип логических отношений между сущностями, показанных на диаграммах классов и объектов. В UML представлены следующие виды отношений:

|  |  |  |
| --- | --- | --- |
| Имя взаимосвязи и графическое обозначение | Описание | Пример |
| Ассоциация | показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов  одного класса к другому |  |
| Наследование | показывает, что один из двух связанных классов (подтип) является частной формой  другого (надтипа) |  |
| Реализация/ Имплементация | отношение между классами, в котором один из них (клиент) реализует поведение, заданное другим (поставщиком) |  |
| Зависимость | связь использования, указывающая, что изменение спецификаций одной сущности может повлиять на другие сущности, которые используют её |  |
| Агрегация | особая разновидность ассоциации, представляющая структурную связь целого с его частями |  |

Композиция

четко

с

агрегации

форма

отношениями

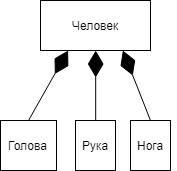
выраженными

и

совпадением

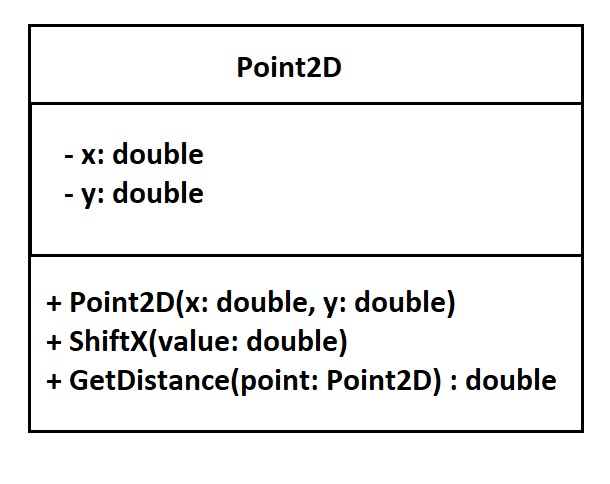
владения

времени жизни частей и целого



### Пример использования UML диаграммы классов

Ниже приведена UML диаграмма класса Point2D:



Данной диаграмме класса будет соответствовать определение класса Point2D в программе:

|  |
| --- |
| class Point2D  {  private double x;  private double y;  public Point2D(double x, double y)  {  this.x = x;  this.y = y;  }  public ShiftX(double value)  {  x += value;  }  public double GetDistance(Point2D otherPoint)  {  return Math.Sqrt(Math.Pow(x - otherPoint.x, 2) + Math.Pow(y - otherPoint.y, 2));  }  } |

## Структуры

Наряду с классами структуры представляют еще один способ создания собственных типов данных в C#. Более того многие примитивные типы, например, int, double и т.д., по сути являются структурами. Структуры относятся к типу значения, а классы к ссылочному типу.

### Определение структуры

Для определения структуры применяется ключевое слово **struct**:

|  |
| --- |
| struct имя\_структуры  {  // элементы структуры  } |

После слова struct идет название структуры и далее в фигурных скобках размещаются элементы структуры - поля, методы и т.д.

Например, определим структуру, которая будет называться Person и которая будет представлять человека:

|  |
| --- |
| struct Person  {  } |

Как и классы, структуры могут хранить состояние в виде полей (переменных) и определять поведение в виде методов. Например, добавим в структуру Person пару полей и метод:

|  |
| --- |
| struct Person  {  public string name;  public int age;    public void Print()  {  Console.WriteLine($"Имя: {name} Возраст: {age}");  }  } |

В данном случае определены две переменные - name и age для хранения соответственно имени и возраста человека и метод Print для вывода информации о человеке на консоль.

### Создание объекта структуры

### Инициализация с помощью конструктора

Для использования структуры ее необходимо инициализировать. Для инициализации создания объектов структуры, как и в случае с классами, применяется вызов конструктора с оператором **new**. Даже если в коде структуры не определено ни одного конструктора, тем не менее существует как минимум один конструктор - конструктор по умолчанию, который генерируется компилятором. Этот конструктор не принимает параметров и создает объект структуры со значениями по умолчанию.

Например, создадим объект структуры Person с помощью конструктора по умолчанию:

|  |
| --- |
| Person tom = new Person(); // вызов конструктора  tom.name = "Tom"; // изменяем значение по умолчанию в поле name    tom.Print(); // Имя: Tom Возраст: 0    struct Person  {  public string name;  public int age;    public void Print()  {  Console.WriteLine($"Имя: {name} Возраст: {age}");  }  } |

В данном случае создается объект tom. Для его создания вызывается конструктор по умолчанию, который устанавливает значения по умолчанию для его полей. Для числовых данных это значение 0, поэтому поле age будет иметь значение 0. Для строк это значение null, которое указывает на отсутствие значения. Но далее, если поля доступны (а в данном случае поскольку они имеют модификатор public они доступны), мы можем изменить их значения. Так, здесь полю name присваивается строка "Tom".

### Непосредственная инициализация полей

Если все поля структуры доступны (как в случае с полями структуры Person, который имеют модификатор **public**), то структуру можно инициализировать без вызова конструктора. В этом случае необходимо присвоить значения всем полям структуры перед получением значений полей и обращением к методам структуры.

|  |
| --- |
| Person tom; // не вызываем конструктор инициализация полей  tom.name = "Sam";  tom.age = 37;    tom.Print(); // Имя: Sam Возраст: 37    struct Person  {  public string name;  public int age;    public void Print()  {  Console.WriteLine($"Имя: {name} Возраст: {age}");  }  } |

### Инициализация полей по умолчанию

Стоит отметить, что начиная с версии C# 10, мы можем напрямую инициализировать поля структуры при их определении (до C# 10 это делать было нельзя):

|  |
| --- |
| Person tom = new Person();  tom.Print(); // Имя:Tom Возраст: 1    struct Person  {  // инициализация полей значениями по умолчанию - доступна с C#10  public string name = "Tom";  public int age = 1;  public Person() { }  public void Print() =>Console.WriteLine($"Имя: {name} Возраст: {age}");  } |

Однако даже в этом случае, несмотря на значения по умолчанию, необходимо явно определить и вызывать конструктор, если мы хотим использовать эти значения.

### Конструкторы структуры

Как и класс, структура может определять конструкторы. Например, добавим в структуру Person конструктор:

|  |
| --- |
| Person bob = new("Bob");  Person sam = new("Sam", 25);    tom.Print(); // !!!! Имя: Возраст: 0  bob.Print(); // Имя: Bob Возраст: 1  sam.Print(); // Имя: Sam Возраст: 25    struct Person  {  public string name;  public int age;    public Person(string name = "Tom", int age = 1)  {  this.name = name;  this.age = age;  }  public void Print() => Console.WriteLine($"Имя: {name} Возраст: {age}");  } |

В данном случае в структуре Person определен конструктор с двумя параметрами, для которых предоставлены значения по умолчания. Однако обратите внимание на создание первого объекта структуры:

|  |
| --- |
| Person tom = new(); // по прежнему используется конструктор без параметров по умолчанию  tom.Print(); // !!!! Имя: Возраст: 0 |

Здесь по-прежнему применяется конструктор по умолчанию, тогда как при инициализации остальных двух переменных структуры применяется явно определенный конструктор.

Однако начиная с версии C# 10 мы можем определить свой конструктор без параметров:

|  |
| --- |
| Person tom = new();    tom.Print(); // Имя: Tom Возраст: 37    struct Person  {  public string name;  public int age;    public Person()  {  name = "Tom";  age = 37;  }  public void Print() => Console.WriteLine($"Имя: {name} Возраст: {age}");  } |

Стоит отметить, что до версии C# 11 при определении конструктора структуру в нем необходимо было инициализировать все поля структуры, начиная с версии C# 11 это делать необязательно.

В случае если нам необходимо вызывать конструкторы с различным количеством параметров, то мы можем, как и в случае с классами, вызывать их по цепочке:

|  |
| --- |
| Person tom = new();  Person bob = new("Bob");  Person sam = new("Sam", 25);    tom.Print(); // Имя: Tom Возраст: 1  bob.Print(); // Имя: Bob Возраст: 1  sam.Print(); // Имя: Sam Возраст: 25    struct Person  {  public string name;  public int age;    public Person() : this("Tom")  { }  public Person(string name) : this(name, 1)  { }  public Person(string name, int age)  {  this.name = name;  this.age = age;  }  public void Print() => Console.WriteLine($"Имя: {name} Возраст: {age}");  } |

Конструкторы по-прежнему должны инициализировать значения всех полей, однако поскольку при вызове любого конструктора цепочка все равно закончится на последнем конструкторе, который выполняет инициализацию, то инициализацию полей в других конструкторах можно не делать.

### Копирование структуры с помощью with

Если нам необходимо скопировать в один объект структуры значения из другого с небольшими изменениями, то мы можем использовать оператор **with**:

|  |
| --- |
| Person tom = new Person { name = "Tom", age = 22 };  Person bob = tom with { name = "Bob" };  bob.Print(); // Имя: Bob Возраст: 22 |

В данном случае объект bob получает все значения объекта tom, а затем после оператора with в фигурных скобках указывается поля со значениями, которые мы хотим изменить.

# Лекция 3 - 4. Основные принципы ООП

Парадигма разработки – это набор правил и критериев, соблюдаемых разработчиками, чтобы выдержать конкретную стилистику и модель написания кода.

Единая парадигма помогает избегать ошибок, упрощает работу в команде и ускоряет разработку. Ориентируясь на одну парадигму, можно корректно структурировать код приложения, зная четкие правила, выбранные командой, которая работает над конкретным проектом.

Существуют различные типы парадигм, например, процедурный, ориентированный на работу с функциями, или логический, подразумевающий решение базовых логических задач в духе «если А = true, то и B = true». Но есть и более интересный подход к решению задач разработки, и это ООП.

ООП – это одна из парадигм разработки, подразумевающая организацию программного кода, ориентируясь на данные и объекты, а не на функции и логические структуры.

Обычно объекты в подобном коде представляют собой полноценные блоки с данными, которые имеют определенный набор характеристик и возможностей. Объект может олицетворять что угодно – от человека с именем, фамилией, номером телефона, паролем и другой информацией до мелкой утилиты с минимумом характеристик из реального мира, но увеличенным набором функций. Объекты могут взаимодействовать друг с другом, пользователем и любыми другими компонентами программы.

ООП заставляет разработчиков фокусироваться на объектах, которыми нужно манипулировать, а не на той логике, что позволяет изменять данные и как-то с ними взаимодействовать. Такой подход хорошо работает в случае с комплексными программными решениями, требующими постоянной поддержки со стороны большого числа программистов.

Объектно-ориентированное программирование исповедует ряд принципов, лежащих в основе правил создания и использования всех структурных элементов, включая классы, объекты, методы и прочие компоненты.

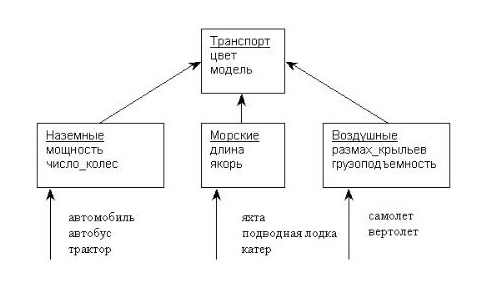
Ниже разберем основные принципы ООП.

## Наследование

**Наследование** - это отношение между классами, при котором класс использует структуру или поведение другого класса.

Наследование определяет взаимоотношений между классами и объектами. Чтобы не создавать кучу одинаковых объектов или классов, можно создать класс над классами с более общими характеристики и функциями, а потом постепенно наследовать от него те или иные возможности. Используя специальную конструкцию, программист может забрать из класса ряд атрибутов или методов, оставить их в прежнем виде и дополнить новыми или же слегка переосмыслить на свое усмотрение, а потом создать из них уникальный объект или подкласс для дальнейшего наследования опций.

Наследование вводит иерархию классов.



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:

|  |
| --- |
| 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().

### Особенности наследования

* У одного родительского класса может быть сколько угодно классов-наследников.
* В дочернем классе свойства при наследовании повторно не описываются.
* Объекты производных классов обычно являются более узкоспециализированными.
* При наследовании потомок сохраняет свойства и методы родительского класса.
* При наследовании потомок добавляет новые свойства и методы.
* При наследовании потомок может менять старые свойства и методы.

### Ограничения при наследовании

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.
* Нельзя унаследовать класс от статического класса.
* 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  {  } |

### 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);  }  } |

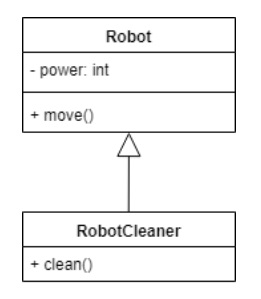
Этот код не сработает и выдаст ошибку, так как переменная \_name объявлена с модификатором private и поэтому к ней доступ имеет только класс Person. Но зато в классе Person определено общедоступное свойство Name, которое мы можем использовать, поэтому следующий код у нас будет работать нормально:

|  |
| --- |
| 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**.

### Наследование в терминах UML

Для обозначения наследования в UML применяется взаимосвязь «обобщение». От дочернего класса идет сплошная стрелка к родительскому классу, что означает дочерний класс является наследником родительского класса.



### Ключевое слово base

Добавим в классы Person и Employee, описанные выше, 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).

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

### Конструкторы при наследовании

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

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| public Employee(string company)  {  Company = company;  } |

Фактически был бы эквивалентен следующему конструктору:

|  |
| --- |
| public Employee(string company)  :base()  {  Company = company;  } |

### Слово sealed для классов

Можно запретить наследоваться от класса, добавив к его объявлению ключевое слово sealed:

|  |
| --- |
| public sealed class C  {  // код  }  public class D : C  {  // ошибка компиляции, нельзя наследоваться от sealed класса  } |

Запрет наследования используют из соображений безопасности – ведь наследники могут переопределять методы как хотят и обращаться к protected членам, а хочется запретить менять реализацию класса.

Запрет наследования улучшает производительность, так как sealed методы не могут быть переопределены, реализованы как не виртуальные, поэтому их вызов быстрее.

### Когда не нужно использовать наследование

* Если мы просто хотим использовать некоторые методы класса-родителя, чтобы выполнить свою работу.
* При этом наш класс логически не сильно связан с классом-родителем, либо большая часть методов класса-родителя ему вообще не нужна
* Допустим, есть класс «Окно операционной системы». Он очень сложно устроен и много чего умеет, например, отрисовываться на экране, получать события от пользователя о нажатиях мыши и клавиатуры и т.д. И еще у него есть ширина и высота и методы для работы с ними.
* Допустим, мы хотим создать свой класс для геометрических фигур, и нам тоже надо уметь работать с шириной и высотой. Отнаследовавшись от окна, мы бы получили реализацию этих методов, но тем самым мы:

– Получили много лишнего кода

– Наш класс может использоваться везде, где нужны окна, а это не нужно.

## Инкапсуляция

**Инкапсуляция** - это сокрытие отдельных деталей внутреннего устройства классов от внешних по отношению к нему объектов или пользователей.

Вся важная информация, необходимая для работы объекта, в нем же и хранится. И только определенные данные доступны для внешних функций и объектов. Данные конкретного класса хранятся в пределах этого класса. Вносить в них изменения, используя другие классы, нельзя. У окружения есть право только запрашивать «публичные» методы и атрибуты. Такой подход обеспечивает повышенный уровень безопасности, а также сокращает шансы на случайное повреждение данных внутри какого-то класса со стороны.

Также под инкапсуляцией понимают ограничение области видимости для переменных и функций классов.

### Основные модификаторы доступа

Все поля, методы и остальные компоненты класса имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для компонентов класса. То есть модификаторы доступа определяют контекст, в котором можно употреблять данную переменную или метод.

В языке C# применяются следующие модификаторы доступа:

* **private**: закрытый или приватный компонент класса или структуры. Приватный компонент доступен только в рамках своего класса или структуры.
* **private protected**: компонент класса доступен из любого места в своем классе или в производных классах, которые определены в той же сборке.
* **file**: добавлен в версии C# 11 и применяется к типам, например, классам и структурам. Класс или структура с такми модификатором доступны только из текущего файла кода.
* **protected**: такой компонент класса доступен из любого места в своем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
* **internal**: компоненты класса или структуры доступен из любого места кода в той же сборке, однако он недоступен для других программ и сборок.
* **protected internal**: совмещает функционал двух модификаторов protected и internal. Такой компонент класса доступен из любого места в текущей сборке и из производных классов, которые могут располагаться в других сборках.
* **public**: публичный, общедоступный компонент класса или структуры. Такой компонент доступен из любого места в коде, а также из других программ и сборок.

## Абстракция

### Абстрактные классы

**Абстрактный** **класс** – это класс, экземпляры которого нельзя создать.

Базовый класс объявляют абстрактным, если не имеет смысл создавать его экземпляры, а имеет смысл создавать только экземпляры его наследников.

Абстрактный класс помечается словом **abstract.**

Абстрактный класс может иметь абстрактные методы – методы без реализации. Их надо пометить словом abstract. Конструктор часто делают protected - все равно создать экземпляры нельзя. Абстрактный класс может иметь обычные поля и методы.

Пусть класс Shape – абстрактный:

|  |
| --- |
| public abstract class Shape  {  private Color color;  public abstract double GetWidth();  public abstract double GetHeight();  public abstract double GetArea();  protected Shape(Color color)  {  this.color = color;  }  protected Color GetColor()  {  return color;  }  } |

Экземпляр такого абстрактного класса создавать нельзя:

|  |
| --- |
| Shape shape = new Shape(); // будет ошибка компиляции  //нельзя создавать экземпляр абстрактного класса |

Если в классе есть хотя бы один абстрактный метод или от родителей достался нереализованный абстрактный метод, то класс обязан быть абстрактным.

|  |
| --- |
| public class TestClass  {  // ошибка компиляции – класс должен быть помечен как abstract  public abstract void TestMethod();  } |

Класс можно делать абстрактным, даже если в нем нет собственных или унаследованных от родителей абстрактных методов.

|  |
| --- |
| public abstract class TestClass  {  public void TestMethod()  {  Console.WriteLine("TestMethod");  }  } |

Если нет возможности или мы не хотим реализовывать абстрактный член, доставшийся от абстрактного базового класса, то данный класс тоже помечается как абстрактный.

|  |
| --- |
| abstract class Person  {  public abstract string Name { get; set; }  }  abstract class Manager : Person {  // абстрактный класс не обязан иметь реализацию абстрактных членов,  // доставшихся от базового класса  } |

Абстрактные классы нужны чтобы создавать базовые классы, которые реализуют некоторую общую логику, но при этом некоторые аспекты на данном уровне абстракции еще не известны.

### Абстрактные свойства

Абстрактные свойства помечаются словом abstract.

Ниже приведен пример абстрактного свойства Name в классе Person:

|  |
| --- |
| abstract class Person  {  public abstract string Name { get; set; }  } |

В классе наследнике необходимо переопределить это свойство:

|  |
| --- |
| class Client : Person  {  private string name;  public override string Name  {  get  {  return "Mr/Ms. " + name;  }  set  {  name = value;  }  }  } |

### Когда следует делать класс абстрактным

* Когда нет смысла создавать экземпляры этого класса.
* На текущем уровне абстракции непонятно, как реализовать какие-то методы, и нет какой-то разумной реализации по умолчанию.

|  |
| --- |
| public abstract class Shape  {  public abstract double GetWidth();  public abstract double GetHeight();  public abstract double GetArea();  } |

Тут непонятно, каковы размеры и площадь фигуры, потому что мы не знаем её тип, положение.

### Когда не следует делать класс абстрактным

* Объекты класса уже можно создавать и использовать

|  |
| --- |
| Shape s = new Square(Color.RED, 1);  Console.WriteLine(s.GetArea()) |

* Мы реализовали все абстрактные методы всех родителей, поэтому класс можно делать не абстрактным.
* Мы понимаем, как реализовать определенный метод.

|  |
| --- |
| public class Square : Shape  {  private double sideLength;  public Square(Color color, double sideLength) : base(color)  {  this.sideLength = sideLength;  }  public override double GetWidth()  {  return sideLength;  }  public override double GetHeight()  {  return sideLength;  }  public override double GetArea()  {  return sideLength \* sideLength;  }  } |

Для квадрата мы уже понимаем, как посчитать размеры и площадь.

### Интерфейсы в терминах ООП

**Интерфейс** обычно подобен абстрактному базовому классу, имеющему только абстрактные члены. В C# интерфейсы обозначаются при помощи ключевого слова **interface**. Для интерфейсов принято соглашение именования – начинать их с буквы I (от Interface)

|  |
| --- |
| public interface IShape  {  double GetWidth();  double GetHeight();  double GetArea();  } |

Интерфейсы в C# могут содержать:

* Методы
* Свойства
* События
* Индексаторы
* Константы
* Операторы
* Статический конструктор
* Вложенные типы
* Статические поля, методы, свойства, индексаторы и события

|  |
| --- |
| interface IMovable  {  void Move(); // Метод  string Name { get; set; } // Свойство  event MoveHandler MoveEvent; // Событие  string this[int index] { get; set; } // Индексаторы  } |

Интерфейс может иметь только **public** члены.

Модификаторы видимости указывать нельзя, они всегда подразумеваются public. Но также, начиная с версии C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса:

|  |
| --- |
| public interface IShape  {  private double GetWidth();  private double GetHeight();  protected double GetArea();  } |

### Реализация интерфейса

Так как интерфейсы в терминах ООП – абстрактные классы, то нельзя создавать их экземпляры.

От интерфейса можно наследоваться, как от обычного класса, указав двоеточие:

|  |
| --- |
| public class Square : IShape  {  // реализация методов GetHeight, GetArea, GetWidth  public void GetWidth()  {  // реализация метода GetWidth  }  } |

Вместо слова «наследуют», про интерфейсы говорят что их «реализуют». То есть класс Square реализует интерфейс IShape. Implement с англ. – реализовывать. Если класс реализует интерфейс, то он должен реализовывать все его методы, либо быть абстрактным.

### Реализация интерфейсов и наследование

Класс может реализовывать несколько интерфейсов, в этом отличие от классов.

|  |
| --- |
| public interface ITest1  {  void TestMethod1();  }  public interface ITest2  {  void TestMethod2();  }  public class TestClass : ITest1, ITest2  {  public void TestMethod1()  {  }  public void TestMethod2()  {  }  } |

Можно одновременно наследоваться от некоторого класса и реализовывать сколько угодно интерфейсов. В этом случае класс должен идти первым после «:»

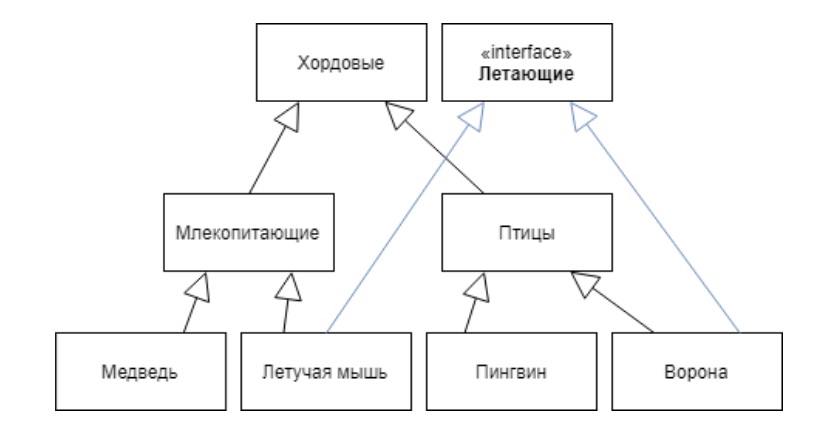
|  |
| --- |
| public abstract class Animal  {  public abstract void Move();  }  public interface IEating  {  void Eat();  }  public interface ISpeaking {  void Speak();  }  public class Cat : Animal, IEating, ISpeaking  {  public override void Move()  {  }  public void Eat()  {  }  public void Speak()  {  }  } |

Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию.

|  |
| --- |
| interface IMovable  {  void Move()  {  Console.WriteLine("Walking");  }  // реализация свойства по умолчанию только для чтения  int MaxSpeed  {  get  {  return 0;  }  }  } |

Интерфейсы полезны в следующих случаях:

* Чтобы указать для класса признак, который не вписывается в иерархию классов



* Когда хочется выделить некоторую абстракцию, но непонятно, как она будет реализована. Или реализации могут быть абсолютно несхожими между собой

|  |
| --- |
| interface ILogger  {  // сообщает о предупреждении  void Warning(string text);  // сообщает об ошибке  void Error(string text);  // информационное сообщение  void Info(string text);  } |

### Применение интерфейсов в классах и структурах

|  |
| --- |
| interface IMovable  {  void Move();  }  class Person : IMovable  {  public void Move()  {  Console.WriteLine("Человек идет");  }  }  struct Car : IMovable  {  public void Move()  {  Console.WriteLine("Машина едет");  }  } |

## Полиморфизм

**Полиморфизм** — возможность объектов с одинаковой спецификацией иметь различную реализацию.

Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

### Виды полиморфизма

* **Ad-hoc - полиморфизм**

Позволяет определять поведение в зависимости от входных аргументов.

К Ad-hoc – полиморфизму относят:

- Перегрузка методов

- Методы с произвольным количеством аргументов

**-** Необязательные аргументы

* **Параметрический полиморфизм**

Позволяет создавать универсальные базовые типы и методы для них.

Повышает повторное использования кода.

К параметрическому полиморфизму относят:

- Обобщения (универсальные шаблоны)

* **Полиморфизм подтипов**

Упрощает разработку методов для разных типов данных.

К полиморфизму подтипов относят:

- Переопределение методов

### Перегрузка методов

В одном классе можно создавать методы с одинаковыми именами, но разной сигнатурой:

|  |
| --- |
| public class Summator  {  public double Sum(int a, int b)  {  return a + b;  }  public int Sum(int a, int b, int c)  {  return a + b + c;  }  } |

### Сигнатура метода для перегрузки

В сигнатуру входят: название метода, количество, тип, порядок и модификаторы аргументов.

В классе нельзя определить два метода с одинаковой сигнатурой.

В сигнатуру не входит возвращаемый тип!

|  |
| --- |
| abstract public double Sum(int a, int b);  abstract public double Sum(double a, double b);  abstract public int Sum(int a, int b, int c);  abstract public double Sum(int a, int b);// Конфликт именования методов  abstract public int Sum(int a, int b); // Конфликт именования методов |

### Методы с произвольным количеством аргументов

|  |
| --- |
| public static double Average(params double[] numbers)  {  double total = 0.0;  foreach (double d in numbers)  {  total += d;  }  return total / numbers.Length;  }  Average(5); // выведется 5  Average(2, 4); // выведется 3 |

Использовать **params** в списке параметров метода можно только один раз. При этом такой параметр обязательно должен быть **последним** в списке аргументов

|  |
| --- |
| public static double Example1(int a, params double[] numbers)  {  // OK  }  public static double Example2(params double[] numbers, int a)  {  // Ошибка компиляции, params double[] numbers  // должен быть последним параметром  } |

### Необязательные аргументы

Сигнатура метода может указывать, являются ли его параметры обязательными или нет

|  |
| --- |
| // Метод создания квадрата с заданным цветом  public static Square CreateSquare(int size, int color)  {  return new Square(size, color);  }  // Метод создания красного квадрата  public static Square CreateRedSquare(int size)  {  return new Square(size, Color.Red);  }  // Универсальный метод  public static Square CreateSquare(int size, int color = Color.Red)  {  return new Square(size, color);  }  // Пример вызова метода с необязательными аргументами  var redSquare = CreateSquare(5);  var square = CreateSquare(5, Color.BLUE); |

Определение каждого необязательного параметра содержит его значение по умолчанию.

Необязательные параметры (их может быть несколько) определяются в конце списка параметров после всех обязательных параметров.

### Обобщения (универсальные шаблоны)

Иногда нужно производить одинаковые операции над данными разного типа.

Пример: идентификатор банковского счета может быть любого типа (число, строка и др.)

|  |
| --- |
| class Account  {  public int Id { get; set; }  public int Sum { get; set; }  } |

**Решение 1:**

Обьявить идентификатор как object:

|  |
| --- |
| class Account  {  public object Id { get; set; }  public int Sum { get; set; }  }  Account acc1 = new Account { Id = 1 };  Account acc2 = new Account { Id = "705b58a" };  int id1 = (int)acc1.Id;  string id2 = (string)acc2.Id;  Console.WriteLine(id1);  Console.WriteLine(id2); |

В данном случае необходимо конвертировать тип идентификатора при каждом обращении, возникает путаница и есть вероятность ошибки.

**Решение 2:**

Использовать обобщения:

|  |
| --- |
| class Account<T>  {  public T Id { get; set; }  public int Sum { get; set; }  }  Account<int> acc1 = new Account<int> { Id = 1 };  Account<string> acc2 = new Account<string> { Id = "705b58a" };  int id1 = acc1.Id;  string id2 = acc2.Id;  Console.WriteLine(id1);  Console.WriteLine(id2); |

Тип идентификатора задается переменной (в данном случае T).

При инициализации объекта переменная T инициализируется типом (в данном случае int и string), конвертировать типы не нужно.

### Использование нескольких универсальных параметров

|  |
| --- |
| class Transaction<U, V>  {  public U FromAccount { get; set; } // с какого счета перевод  public U ToAccount { get; set; } // на какой счет перевод  public V Code { get; set; } // код операции  public int Sum { get; set; } // сумма перевода  } |

### Наследование обобщенных типов

|  |
| --- |
| // Обобщенный базовый класс  abstract class Account<T>  {  public T Id { get; private set; }  protected Account(T id)  {  Id = id;  }  }  // Необобщенный класс-наследник  class StringAccount : Account<string>  {  public StringAccount(string id) : base(id) { }  }  // Класса-наследник, который типизирован тем же типом, что и базовый  class UniversalAccount<T> : Account<T>  {  public UniversalAccount(T id) : base(id) { }  } |

### Ограничения обобщений

|  |
| --- |
| class NumbersCollection<T> where T : ICollection<int>, new()  {  public T Numbers { get; set; } = new T();  // добавление чисел в коллекцию  public void Add(int i)  {  Numbers.Add(i);  }  // вывод чисел  public void Print()  {  foreach (int v in Numbers)  {  Console.WriteLine(v);  }  }  } |

С помощью выражения where T : ICollection мы указываем, что используемый тип T обязательно должен реализовывать интерфейс ICollection и соответственно обращаться к Numbers как к коллекции.

Стандартное ограничение new() позволяет создавать новые экземпляры универсального типа T с помощью общедоступного (public) конструктора без параметров.

|  |
| --- |
| // создание коллекции на основе списка  var numbersListСollection = new NumbersCollection<List<int>>();  numbersListСollection.Add(5);  numbersListСollection.Add(1);  numbersListСollection.Add(2);  numbersListСollection.Print(); // выведется 5 1 2  // создание коллекции на основе отсортированного множества  var numbersSortedSetCollection = new NumbersCollection<SortedSet<int>>();  numbersSortedSetCollection.Add(5);  numbersSortedSetCollection.Add(1);  numbersSortedSetCollection.Add(2);  numbersSortedSetCollection.Print(); // выведется 1 2 3 |

### Обобщенные методы

|  |
| --- |
| static void Swap<T>(ref T lhs, ref T rhs)  {  T temp;  temp = lhs;  lhs = rhs;  rhs = temp;  }  int a = 1, b = 2;  Swap<int>(ref a, ref b); // или Swap(ref a, ref b);  string s1 = "hello", s2 = "world";  Swap<string>(ref s1, ref s2); // или Swap(ref s1, ref s2); |

### Обобщения в стандартных библиотеках C#

Обобщенные коллекции (System.Collections.Generic):

* List
* Dictionary
* LinkedList
* Queue
* и др.

Пример:

|  |
| --- |
| class Product  {  public Product(string title, double price, string description)  {  Title = title;  Price = price;  Description = description;  }  public string Title { get; private set; }  public double Price { get; private set; }  public string Description { get; private set; }  }  Dictionary<long, Product> products = new Dictionary<long, Product>();  // добавление элементов  products.Add(0, new Product("Хлеб", 30, "Бородинский"));  products.Add(1, new Product("Молоко", 50, "Простоквашино"));  products.Add(2, new Product("Стиральный порошок", 30, "Для цветной стирки"));  // проверка на наличие элемента по заданному ключу  if (products.ContainsKey(0))  {  // обращение к элементу по ключу  Console.WriteLine(products[0].Title);  // удаление элемента по ключу  products.Remove(0);  } |

### Переопределение методов

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

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

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

|  |
| --- |
| public class A  {  public virtual void F()  {  Console.WriteLine(1);  }  }  public class B : A  {  public override void F()  {  Console.WriteLine(2);  }  }  A a = new A();  a.F(); // 1  B b = new B();  b.F(); // 2  A c = new B();  c.F(); // 2 |

Для виртуальной функции вызывается та реализация, которая определена для фактического типа объекта, а не для типа ссылки.

Пример – геометрические фигуры

|  |
| --- |
| public class Shape  {  public virtual double GetArea()  {  return 0; // нет разумной реализации  }  }  public class Rectangle : Shape  {  // опущен код полей и конструктора  public override double GetArea()  {  return width \* height;  }  }  // И тогда эти примеры будут работать правильно  Shape s1 = new Rectangle(10, 2);  Console.WriteLine(s1.GetArea()); // 20 (вызывается реализация для  // Прямоугольника) |