**Классы и объекты**

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

В принципе ранее уже использовались классы. Например, тип **string**, который представляет строку, фактически является классом. Или, например, класс **Console**, у которого метод WriteLine() выводит на консоль некоторую информацию. Теперь же посмотрим, как мы можем определять свои собственные классы.

По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова **сlass**:

class название\_класса

{

// содержимое класса

}

После слова **class** идет имя класса и далее в фигурных скобках идет собственно содержимое класса. Например, определим в файле **Program.cs** класс Person, который будет представлять человека:

class Person

{

}

Однако такой класс не особо показателен, поэтому добавим в него некоторую функциональность.

**Поля и методы класса**

Класс может хранить некоторые данные. Для хранения данных в классе применяются **поля**. По сути **поля класса** - это переменные, определенные на уровне класса.

Кроме того, класс может определять некоторое поведение или выполняемые действия. Для определения поведения в классе применяются методы.

Итак, добавим в класс Person поля и методы:

class Person

{

public string name = "Undefined"; // имя

public int age; // возраст

public void Print()

{

Console.WriteLine($"Имя: {name} Возраст: {age}");

}

}

В данном случае в классе Person определено поле name, которое хранит имя, и поле age, которое хранит возраст человека. В отличие от переменных, определенных в методах, поля класса могут иметь модификаторы, которые указываются перед полем. Так, в данном случае, чтобы все поля были доступны вне класса Person поля определены с модификатором **public**.

При определении полей мы можем присвоить им некоторые значения, как в примере выше в случае переменной name. Если поля класса не инициализированы, то они получают значения по умолчанию. Для переменных числовых типов это число 0.

Также в классе Person определен метод Print(). Методы класса имеют доступ к его поля, и в данном случае обращаемся к полям класса name и age для вывода их значения на консоль. И чтобы этот метод был виден вне класса, он также определен с модификатором **public**.

**Создание объекта класса**

После определения класса мы можем создавать его объекты. Для создания объекта применяются **конструкторы**. По сути конструкторы представляют специальные методы, которые называются так же как и класс, и которые вызываются при создании нового объекта класса и выполняют инициализацию объекта. Общий синтаксис вызова конструктора:

new конструктор\_класса(параметры\_конструктора);

Сначала идет оператор **new**, который выделяет память для объекта, а после него идет вызов **конструктора**.

**Конструктор по умолчанию**

Если в классе не определено ни одного конструктора (как в случае с нашим классом Person), то для этого класса автоматически создается пустой конструктор по умолчанию, который не принимает никаких параметров.

Теперь создадим объект класса Person:

Person tom = new Person(); // создание объекта класса Person

// определение класса Person

class Person

{

public string name = "Undefined";

public int age;

public void Print()

{

Console.WriteLine($"Имя: {name} Возраст: {age}");

}

}

Для создания объекта Person используется выражение new Person(). В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект, и через эту переменную мы можем использовать данный объект и обращаться к его функциональности.

**Обращение к функциональности класса**

Для обращения к функциональности класса - полям, методам (а также другим элементам класса) применяется точечная нотация точки - после объекта класса ставится точка, а затем элемент класса:

объект.поле\_класса

объект.метод\_класса(параметры\_метода)

Например, обратимся к полям и методам объекта Person:

Person tom = new Person(); // создание объекта класса Person

// Получаем значение полей в переменные

string personName = tom.name;

int personAge = tom.age;

Console.WriteLine($"Имя: {personName} Возраст {personAge}"); // Имя: Undefined Возраст: 0

// устанавливаем новые значения полей

tom.name = "Tom";

tom.age = 37;

// обращаемся к методу Print

tom.Print(); // Имя: Tom Возраст: 37

class Person

{

public string name = "Undefined";

public int age;

public void Print()

{

Console.WriteLine($"Имя: {name} Возраст: {age}");

}

}

Консольный вывод данной программы:

Имя: Undefined Возраст: 0

Имя: Tom Возраст: 37

**Константы классы**

Кроме полей класс может определять для хранения данных константы. В отличие от полей из значение устанавливается один раз непосредственно при их объявлении и впоследствии не может быть изменено. Кроме того, константы хранят некоторые данные, которые относятся не к одному объекту, а ко всему классу в целом. И для обращения к константам применяется не имя объекта, а имя класса:

Person tom = new Person();

tom.name = "Tom";

tom.age = 37;

tom.Print(); // Person: Tom - 37

Console.WriteLine(Person.type); // Person

// Person.type = "User"; // !Ошибка: изменить константу нельзя

class Person

{

public const string type = "Person";

public string name = "Undefined";

public int age;

public void Print() => Console.WriteLine($"{type}: {name} - {age}");

}

Здесь в классе Person определена константа type, которая хранит название класса:

public const string type = "Person";

Название класса не зависит от объекта. Мы можем создать много объектов Person, но название класса от этого не должно измениться - оно относится ко всем объектам Person и не должно меняться. Поэтому название типа можно сохранить в виде константы.

Стоит отметить, что константе сразу при ее определении необходимо присвоить значение.

Подобно обычным полям мы можем обращаться к константам класса внутри этого класса. Например, в методе Print значение константы выводится на консоль.

Однако если мы хотим обратиться к константе вне ее класса, то для обращения необходимо использовались имя класса:

Console.WriteLine(Person.type); // Person

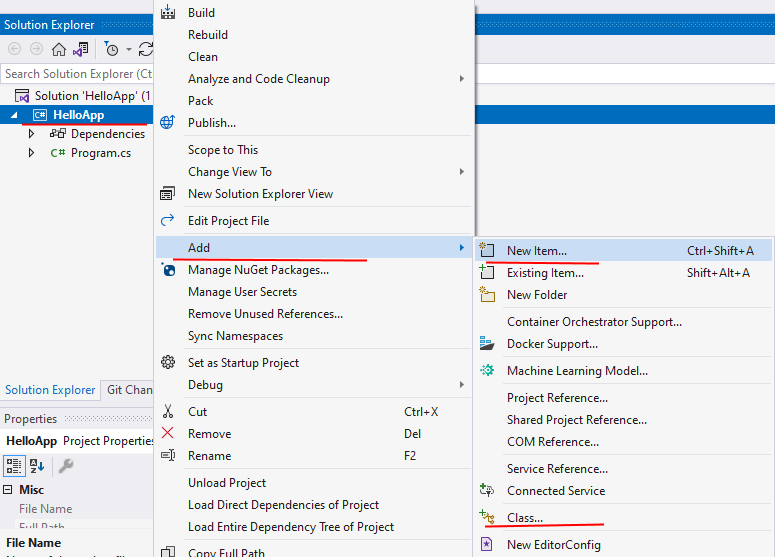
Таким образом, если необходимо хранить данные, которые относятся ко всему классу в целом

**Добавление класса в Visual Studio**

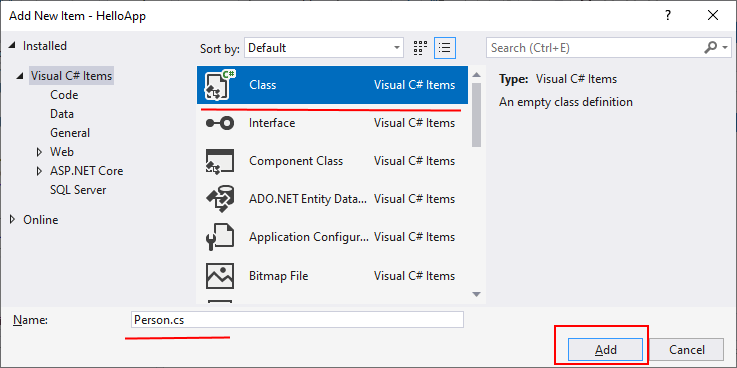
Обычно классы помещаются в отдельные файлы. Нередко для одного класса предназначен один файл. И Visual Studio предоставляет по умолчанию встроенные шаблоны для добвления класса.

Для добавления класса нажмем в Visual Studio правой кнопкой мыши на название проекта:

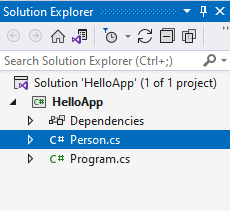
В появившемся контекстном меню выберем пункт **Add -> New Item...** (или **Add -> Class...**)



В открывшемся окне добавления нового элемента убедимся, что в центральной части с шаблонами элементов у нас выбран пункт **Class**. А внизу окна в поле **Name** введем название добавляемого класса - пусть он будет назваться **Person**:



В качестве названия класса можно вводить как Person, так и Person.cs. И после нажатия на кнопку добавления в проект будет добавлен новый класс:



В файле **Person.cs** определим следующий код:

class Person

{

public string name = "Undefined";

public void Print()

{

Console.WriteLine($"Person {name}");

}

}

Здесь определен класс Person с одним полем name и методом Print.

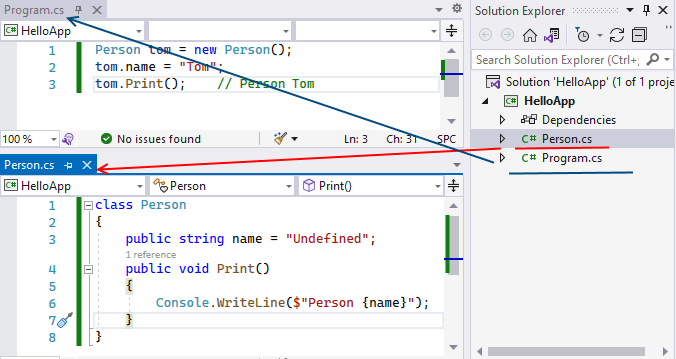
В файле **Program.cs**, который представляет основной файл программы используем класс Person:

Person tom = new Person();

tom.name = "Tom";

tom.Print(); // Person Tom

Таким образом, мы можем определять классы в отдельных файлах и использовать их в программе.



**Создание конструкторов**

В прошлой статье для создания объекта использовался конструктор по умолчанию. Однако мы сами можем определить свои конструкторы. Как правило, конструктор выполняет инициализацию объекта. При этом если в классе определяются свои конструкторы, то он лишается конструктора по умолчанию.

На уровне кода конструктор представляет метод, который называется по имени класса, который может иметь параметры, но для него не надо определять возвращаемый тип. Например, определим в классе Person простейший конструктор:

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

Итак, здесь определен конструктор, который выводит на консоль некоторое сообщение и инициализирует поля класса.

|  |
| --- |
| public Person()  {      Console.WriteLine("Создание объекта Person");      name = "Tom";      age = 37;  } |

Конструкторы могут иметь модификаторы, которые указываются перед именем конструктора. Так, в данном случае, чтобы конструктор был доступен вне класса Person, он определен с модификатором **public**.

Определив конструктор, мы можем вызвать его для создания объекта Person:

|  |
| --- |
| Person tom = new Person();  // Создание объекта Person |

В данном случае выражение Person() как раз представляет вызов определенного в классе конструктора (это больше не автоматический конструктор по умолчанию, которого у класса теперь нет). Соответственно при его выполнении на консоли будет выводиться строка "Создание объекта Person"

Подобным образом мы можем определять и другие конструкторы в классе. Например, изменим класс Person следующим образом:

|  |
| --- |
| Person tom = new Person();          // вызов 1-ого конструктора без параметров  Person bob = new Person("Bob");     //вызов 2-ого конструктора с одним параметром  Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами    tom.Print();          // Имя: Неизвестно  Возраст: 18  bob.Print();          // Имя: Bob  Возраст: 18  sam.Print();          // Имя: Sam  Возраст: 25    class Person  {      public string name;      public int age;      public Person() { name = "Неизвестно"; age = 18; }      // 1 конструктор      public Person(string n) { name = n; age = 18; }         // 2 конструктор      public Person(string n, int a) { name = n; age = a; }   // 3 конструктор        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. И мы можем вызвать один из этих конструкторов для создания объекта класса.

Консольный вывод данной программы:

Имя: Неизвестно Возраст: 18

Имя: Bob Возраст: 18

Имя: Sam Возраст: 25

Стоит отметить, что начиная с версии C# 9 мы можем сократить вызов конструктора, убрав из него название типа:

|  |
| --- |
| Person tom = new ();            // аналогично new Person();  Person bob = new ("Bob");       // аналогично new Person("Bob");  Person sam = new ("Sam", 25);   // аналогично new Person("Sam", 25); |

**Ключевое слово this**

Ключевое слово **this** представляет ссылку на текущий экземпляр/объект класса. В каких ситуациях оно нам может пригодиться?

|  |
| --- |
| Person sam = new("Sam", 25);  sam.Print();          // Имя: Sam  Возраст: 25    class Person  {      public string name;      public int age;      public Person() { name = "Неизвестно"; age = 18; }      public Person(string name) { this.name = name; age = 18; }      public Person(string name, int age)      {          this.name = name;          this.age = age;      }      public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

В примере выше во втором и третьем конструкторе параметры называются также, как и поля класса. И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово **this**. Так, в выражении

**this.name = name;**

первая часть - this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно. Также через ключевое слово **this** можно обращаться к любому полю или методу.

**Цепочка вызова конструкторов**

В примере выше определены три конструктора. Все три конструктора выполняют однотипные действия - устанавливают значения полей name и age. Но этих повторяющихся действий могло быть больше. И мы можем не дублировать функциональность конструкторов, а просто обращаться из одного конструктора к другому также через ключевое слово **this**, передавая нужные значения для параметров:

|  |
| --- |
| class Person  {      public string name;      public int age;      public Person() : this("Неизвестно")    // первый конструктор      { }      public Person(string name) : this(name, 18) // второй конструктор      { }      public Person(string name, int age)     // третий конструктор      {          this.name = name;          this.age = age;      }      public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

В данном случае первый конструктор вызывает второй, а второй конструктор вызывает третий. По количеству и типу параметров компилятор узнает, какой именно конструктор вызывается. Например, во втором конструкторе:

|  |
| --- |
| public Person(string name) : this(name, 18)  { } |

идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

Стоит отметить, что в примере выше фактически все конструкторы не определяют каких-то других действий, кроме как передают третьему конструктору некоторые значения. Поэтому в реальности в данном случае проще оставить один конструктор, определив для его параметров значения по умолчанию:

|  |
| --- |
| Person tom = new();  Person bob = new("Bob");  Person sam = new("Sam", 25);    tom.Print();          // Имя: Неизвестно  Возраст: 18  bob.Print();          // Имя: Bob  Возраст: 18  sam.Print();          // Имя: Sam  Возраст: 25    class Person  {      public string name;      public int age;      public Person(string name = "Неизвестно", int age = 18)      {          this.name = name;          this.age = age;      }      public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

И если при вызове конструктора мы не передаем значение для какого-то параметра, то применяется значение по умолчанию.

**Инициализаторы объектов**

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

|  |
| --- |
| Person tom = new Person { name = "Tom", age = 31 };  // или так  // Person tom = new() { name = "Tom", age = 31 };  tom.Print();          // Имя: Tom  Возраст: 31 |

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания. При использовании инициализаторов следует учитывать следующие моменты:

* С помощью инициализатора мы можем установить значения только доступных из вне класса полей и свойств объекта. Например, в примере выше поля name и age имеют модификатор доступа public, поэтому они 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:

|  |
| --- |
| Person tom = new Person{ name = "Tom", company = { title = "Microsoft"} };  tom.Print();          // Имя: Tom  Компания: Microsoft   class Person  {      public string name;      public Company company;      public Person()      {          name = "Undefined";          company = new Company();      }      public void Print() => Console.WriteLine($"Имя: {name}  Компания: {company.title}");  }   class Company  {      public string title = "Unknown";  } |

Обратите внимание, как устанавливается поле company:

|  |
| --- |
| company = { title = "Microsoft"} |

**Деконструкторы**

Деконструкторы (не путать с деструкторами) позволяют выполнить декомпозицию объекта на отдельные части.

Например, пусть у нас есть следующий класс Person:

|  |
| --- |
| class Person  {      string name;      int age;      public Person(string name, int age)      {          this.name = name;          this.age = age;      }       public void Deconstruct(out string personName, out int personAge)      {          personName = name;          personAge = age;      }  } |

В этом случае мы могли бы выполнить декомпозицию объекта Person так:

|  |
| --- |
| Person person = new Person("Tom", 33);    (string name, int age) = person;    Console.WriteLine(name);    // Tom  Console.WriteLine(age);     // 33 |

Значения переменным из деконструктора передаюся по позиции. То есть первое возвращаемое значение в виде параметра personName передается первой переменной - name, второе возващаемое значение - переменной age.

По сути деконструкторы это не более,чем синтаксический сахар. Это все равно, что если бы мы написали:

|  |
| --- |
| Person person = new Person("Tom", 33);    string name; int age;  person.Deconstruct(out name, out age); |

При получении значений из декоструктора нам необходимо предоставить столько переменных, сколько деконструктор возвращает значений. Однако бывает, что не все эти значения нужны. И вместо возвращаемых значений мы можм использовать прочерк **\_**. Например, нам надо получить только возраст пользователя:

|  |
| --- |
| Person person = new Person("Tom", 33);    (\_, int age) = person;    Console.WriteLine(age);    // 33 |

Поскольку первое возвращаемое значение - это имя пользователя, которое не нужно, в в данном случае вместо переменной прочерк.