

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.

2.1 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.

2.2 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.

2.2.1 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 вычисления 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.

2.2.2 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}");  
    }  
}
```

2.2.3 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() { }  
}
```

2.2.4 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.

2.2.5 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.

2.2.6 Создание объекта класса

Создание экземпляра (объекта) класса происходит следующим образом:

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

2.2.7 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  
}
```

2.3 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:

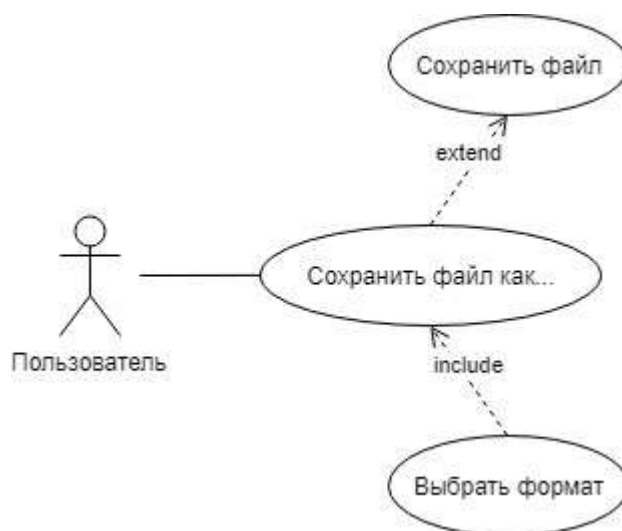


Для описания функций используются действующее лицо (обозначенное человекоподобной фигурой), прецеденты (обозначенные овалами) и ассоциативные связи.

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.



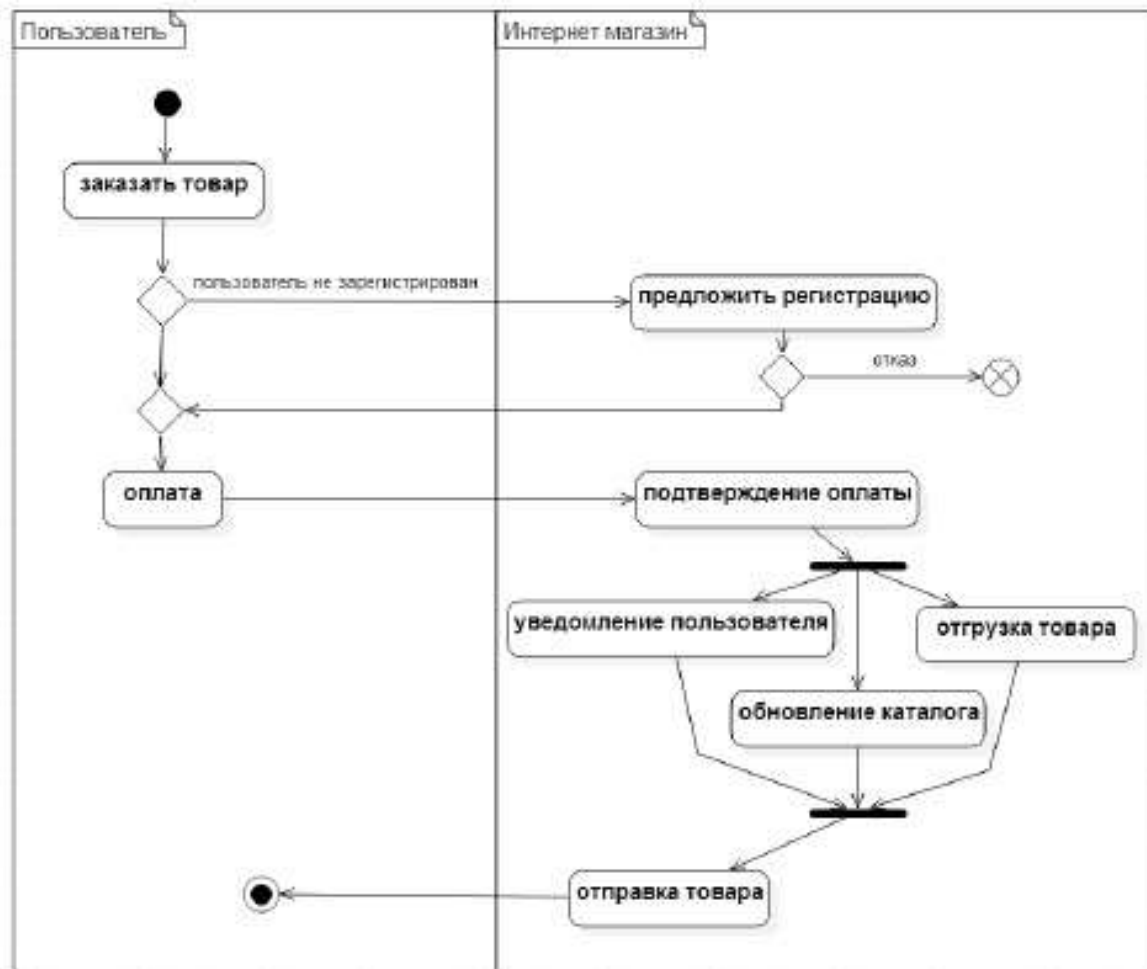
В данном примере описан прецедент расширенной функции «сохранить файл». Для указания того, что сохранение файла с выбором формата содержит в себе выбор формата и является расширением стандартной функции сохранения файла, используется отношение зависимости.



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.

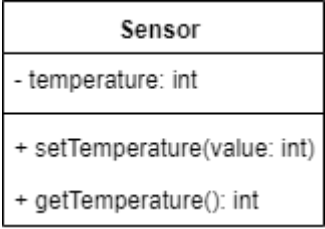
Каждый класс должен обладать именем, отличающим его от других классов. Обычно используются краткие имена, отражающие описываемую сущность. Каждое слово в имени класса традиционно пишут с заглавной буквы, например Sensor или Датчик.

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.


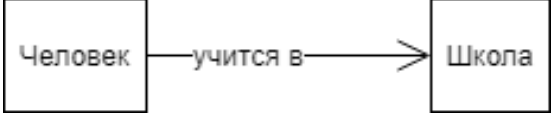


Операция (метод) – это реализация метода класса. Класс может иметь любое число операций либо не иметь ни одной. Часто вызов операции объекта изменяет его атрибуты. Класс может иметь любое число операций либо не иметь ни одной. Графически операции представлены в нижнем блоке описания класса. Можно специфицировать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

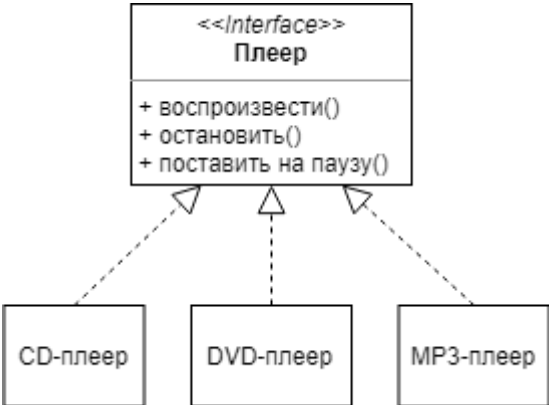
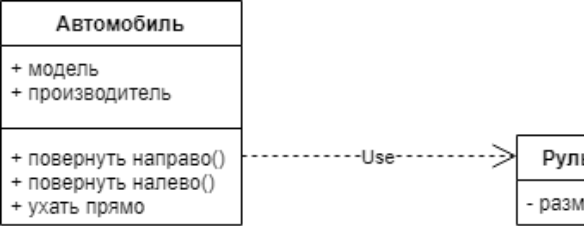
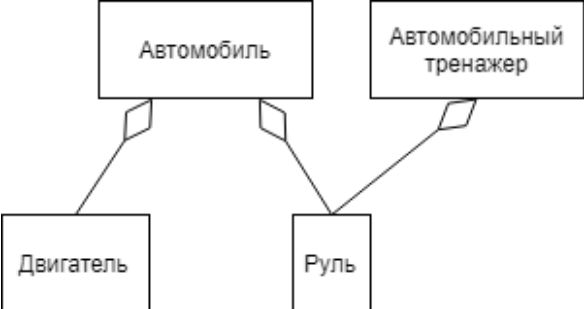
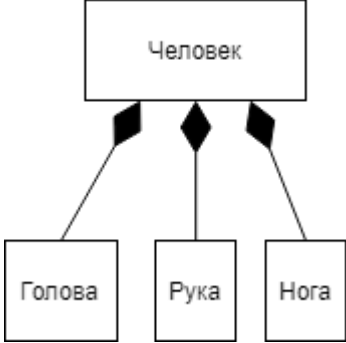
Изображая класс, не обязательно показывать сразу все его атрибуты и операции. Для конкретного представления, как правило, существенна только

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 даёте это понять, завершая каждый список многоточием.



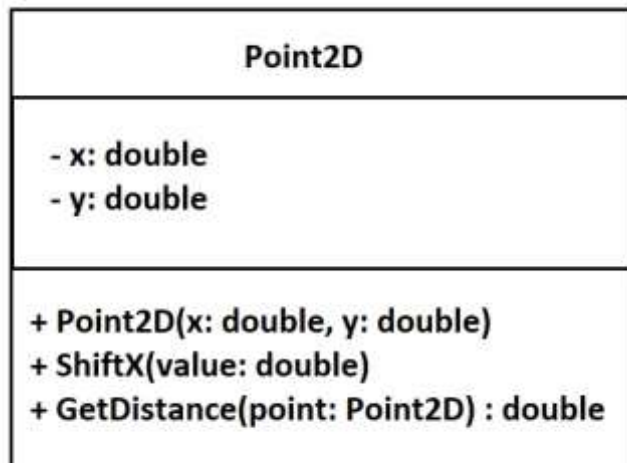
Взаимосвязь (отношение) - это особый тип логических отношений между сущностями, показанных на диаграммах классов и объектов. В UML представлены следующие виды отношений:

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	 <pre>graph LR A[Человек] -- "учится в" --> B[Школа]</pre>
Наследование 	показывает, что один из двух связанных классов (подтип) является частной формой другого (надтипа)	 <pre>graph BT A[ЧеловекИзУниверситета] B[Студент] -- > A C[Преподаватель] -- > A</pre>

<p>Реализация/ Имплементация</p> <p>-----></p>	<p>отношение между классами, в котором один из них (клиент) реализует поведение, заданное другим (поставщиком)</p>	
<p>Зависимость</p> <p>-----></p>	<p>связь использования, указывающая, что изменение спецификаций одной сущности может повлиять на другие сущности, которые используют её</p>	
<p>Агрегация</p> <p>—◇</p>	<p>особая разновидность ассоциации, представляющая структурную связь целого с его частями</p>	
<p>Композиция</p> <p>—◆</p>	<p>форма агрегации с четко выраженными отношениями владения и совпадением времени жизни частей и целого</p>	

2.3.1 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));
    }
}
```

2.4 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.

2.4.1 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.

2.4.2 Создание объекта структуры

2.4.3 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".

2.4.4 Непосредственная инициализация полей

Если все поля структуры доступны (как в случае с полями структуры Person, который имеет 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}");
    }
}

```

2.4.5 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.

2.4.6 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 умолчанию. 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: Том  Возраст: 37

struct Person
{
    public string name;
    public int age;

    public Person()
    {
        name = "Том";
        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.

2.4.7 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.