(Не)краткий справочный материал

Класс и объект

Класс

Класс - тип данных, созданный пользователем. В классе задаются поля, свойства и поведение объекта в виде полей данных и функций для работы с ними. *Класс - "чертёж" или шаблон, по которому могут создаваться **объекты** этого класса.*

В прошлой лабе вы работали с готовым классом `Image` и создавали свой класс `Filters` (или как вы его назвали) для реализации внутри фильтров обработки изображений.

Задается с помощью ключевого слова `class`

Пример класса

```
public class Person
{
    string name;
    int age;
    bool gender;

    void SayHello()
    {
        Console.WriteLine($"Привет, я {name}!")
    }
}
```

Объект

Объект это экземпляр класса, созданный по этому "чертежу", или его копия, которая находится в памяти компьютера.

Для создания объекта класса (переменной этого класса, типа) мы объявляем переменную нужного типа (в нашем случае `Person`), далее указываем имя, далее `=` и используется ключевое слово `new`, после идет название класса.

Пример создания объекта

```
static void Main(string[] args)
{
    Person ivan = new Person();
    Person dimka = new Person();
    Person nastya = new Person();
}
```

В примере выше во всех трех наших объектах (переменных) значения внутри (`name`, `age` и `gender`) никак не определенны.

Конструктор класса

Конструктор класса - специальный метод, который вызывается при создании объекта класса (когда мы пишем `new ClassName()`. Его задача — инициализировать поля объекта, например, задать начальные значения переменных.

Сейчас в нашем классе нет никакого конструктора, но, он как бы есть. Стандартный "невидимый" конструктор есть в каждом классе, даже если он не написан явно.

Конструктор класса также поддерживает перегрузку (разные аргументы).

Пример конструктора

```
public class Person
{
    string name;
    int age;
    bool gender;
```

```
    // А вот и конструктор
    // Он может принимать аргументы
    public Person(string name, int age, bool gender)
    {
        // обратите внимание: аргументы у конструктора
называются также как
        // переменные в самом классе.
        // чтобы явно указать что куда мы хотим
присвоить, мы можем использовать
        // ключевое слово this.
        this.name = name;
        this.age = age;
        this gender = gender;
    }

    void SayHello()
    {
        Console.WriteLine($"Привет, я {name}!")
    }
}
```

Ключевое слово `this` означает использование конкретного, этого экземпляра в котором мы находимся (аналог `self` в Python). Получается, что `this.name` это `name` из класса, а просто `name` это аргумент из конструктора.

Теперь при создании объекта класса мы должны передать аргументы, но куда?
В `... = new Person(вот сюда)`.

Пример создания объекта

```
static void Main(string[] args)
{
    Person ivan = new Person("ivan", 20, false);
    Person dimka = new Person("dimka", 21, false);
    Person nastya = new Person("nastya", 20, true);
```

```
}
```

Оператор доступа

Для доступа к члену типа используется оператор `.` точка. Для получения доступа к полю `name` например, у конкретного объекта нужно написать:

```
Person ivan = new Person("ivan", 20, false);

ivan.name = "Ivan";
```

Таким же образом осуществляется доступ к методам и свойствам.

Сейчас этот код скомпилируется, так как наш `name` - приватный. **См ниже про "Модификаторы доступа"**

Инициализатор

Перед прочтением этого раздела рекомендуется сначала прочитать раздел `|Модификаторы доступа`, он чуть ниже.

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

```
Person ivan = new Person { name = "ivan", age = 31 };
```

При этом поля или свойства к которым мы хотим обратиться должны быть `public` и в классе должен присутствовать "пустой" конструктор.

Модификаторы доступа

Определение

Модификаторы доступа - ключевые слова, которые задают допустимую область видимости для компонентов класса (полей, методов, свойств и т. д.). Для нас сейчас модификатор доступа значит "*а можем ли мы использовать эту переменную или нет когда создаем объект?*"

Модификаторов больше чем 2, для нас сейчас это основа:

`private` **(приватный):** означает, что поле или метод доступны **только внутри самого класса**. Это основной принцип **инкапсуляции** — сокрытия внутренней реализации от внешнего мира (привет `Grow()` из списка).

`public` **(публичный):** делает член класса доступным из любого места программы. Это "публичный интерфейс" нашего класса.

Если модификатор не указан явно, то стандартно он будет `private`. Это значит, что все наши переменные и методы внутри `Person` сейчас имеют модификатор `private`. И мы не можем обратиться к ним напрямую, только через конструктор.

Пример исправления

```
public class Person
{
    public string name;
    public int age;
    public bool gender;

    // А вот и конструктор
    // Он может принимать аргументы
    public Person(string name, int age, bool gender)
    {
        // обратите внимание: аргументы у конструктора
называются также как
        // переменные в самом классе.
        // чтобы явно указать что куда мы хотим
присвоить, мы можем использовать
        // ключевое слово this.
        this.name = name;
```

```
        this.age = age;
        this gender = gender;
    }

    public void SayHello()
    {
        Console.WriteLine($"Привет, я {name}!")
    }
}
```

Теперь мы можем не только создать объекты, но и изменить значения полей или вызвать метод

```
static void Main(string[] args)
{
    Person ivan = new Person("ivan", 20, false);
    Person dimka = new Person("dimka", 21, false);

    ivan.SayHello(); // Выведет в консоль "Привет, я
ivan!"

    ivan.name = "Ivan";

    ivan.SayHello(); // Выведет в консоль "Привет, я
Ivan!"

}
```

Поля и свойства

Есть разница между **полем** и **свойством**.

Поле - переменная любого типа, объявленного непосредственно в классе или структуре. **Отвечает за состояние объекта.**

Свойство - нужно для контроля доступа к полю (только для чтения, например), скрывая реализацию проверок значения или

реализуя дополнительную логику. **Отвечает за контролируемый доступ к состоянию объекта.**

- Поле имеет модификатор `private` и именуется с маленькой буквы (`age`). (самые внимательные поднимутся выше и исправят этот косяк в классе, убрав `public` у полей.)
- Свойство имеет модификатор `public` и дублирует название поля, только с большой буквы (`Age`).

Для свойства также указывается тип данных, и он даже может отличаться от типа поля.

Свойство может реализовывать методы-аксессоры `get` и `set` для свойства.

- `get` срабатывает когда мы хотим получить значение (`int temp = ivan.Age`) **(геттер)**
- `set` срабатывает когда мы хотим присвоить значение (`ivan.Age = 15`) **(сеттер)**

Значение, которое мы пытаемся присвоить свойству, внутри свойства будет называться `value`.

Пример свойства.

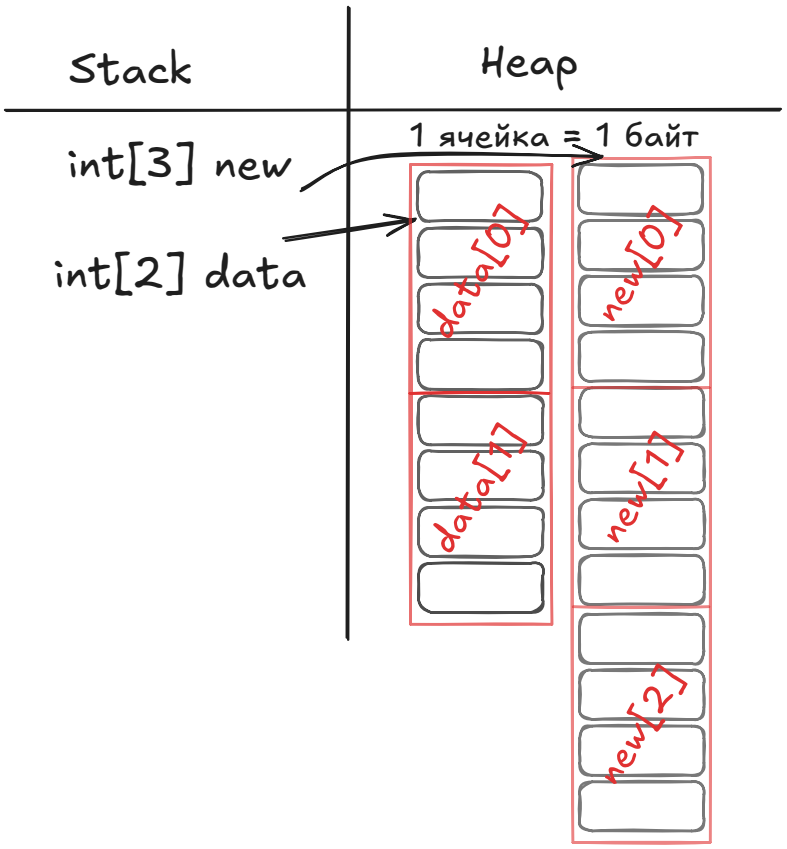
```
public int Age
{
    get { return age; } //просто возвращаем наш age
    set
    {
        if (value < 0) // возраст же не может быть
отрицательным
            age = value;
        else
            age = 0;
    }
}
```

Пример свойства, с отличающимся типом данных и доступное только для чтения.

```
public string Gender
{
    get
    {
        if (gender)
            return "Female"
        else
            return "Male"
    }
}
```

Напоминалка про ссылочные типы

Массивы как ссылочные типы: В C# переменная типа массив (например, `int[]`) хранит не сами данные, а **ссылку (адрес) на область в памяти**, где эти данные расположены. Операция `_data = newData;` не копирует содержимое массивов, а лишь изменяет эту ссылку, заставляя `_data` указывать на новый объект в памяти. Старый массив, на который больше нет ссылок, будет позже автоматически удален **сборщиком мусора (Garbage Collector, GC)**.



Индексатор

Специальное свойство, которое именуется `this` и позволяет обращаться к объекту класса с использованием синтаксиса доступа к элементам массива (квадратные скобки `[]`). В основном, нужен только если внутри есть массив и к нему нужен удобный доступ.

```
public Тип_данных_для_возврата this[int index]
{
    get
    {
        return someArray[index];
    }

    set
    {
        someArray[index] = value;
    }
}
```

Исключения

- **Исключения (`Exception`):** Стандартизированный механизм для обработки ошибок и нештатных ситуаций в программе. Оператор `throw new IndexOutOfRangeException();` **генерирует (выбрасывает)** исключение, а конструкция `try...catch` позволяет **перехватить (обработать)** его. Это более надежный и предсказуемый подход, чем возврат кодов ошибок (например, `-1`).