(Не)краткий справочный материал

Класс и объект

Класс

Класс — это тип данных, созданный пользователем. Можно думать о классе как о "чертеже" или шаблоне. В этом чертеже мы описываем, из чего состоит будущий объект: какие данные (поля) он хранит и что он умеет делать (методы).

Задается с помощью ключевого слова `class`.

Пример класса:

```
public class Person
{
    // 1. Поля (внутреннее состояние)
    // Поля принято делать приватными и называть с маленькой буквы.
    // Если модификатор доступа не указан, по умолчанию он private.
    private string name;
    private int age;
    private bool isMale;

    // 2. Методы (поведение)
    public void SayHello()
    {
        Console.WriteLine($"Привет, я {name}!");
    }
}
```

Объект

Объект — это конкретный экземпляр, созданный по "чертежу" (классу). Если `Person` — это чертёж человека, то конкретные люди Иван, Дмитрий и Анастасия — это объекты. Каждый из них существует в памяти компьютера как самостоятельная копия.

Для создания объекта используется ключевое слово `new`.

Пример создания объектов:

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

Сейчас мы создали три объекта, но их поля (`name`, `age`, `isMale`) имеют значения по умолчанию (`null`, `0`, `false`). Чтобы задать им начальные значения, нужен конструктор.

Конструктор класса

Конструктор — это специальный метод, который вызывается в момент создания объекта (`new Person()`). Его главная задача — инициализировать поля, то есть задать объекту начальное состояние.

- У конструктора нет возвращаемого типа, и его имя всегда совпадает с именем класса.
- Если вы не написали ни одного конструктора, компилятор сам создаст "невидимый" публичный конструктор без параметров.
- Конструкторы можно перегружать, создавая несколько версий с разным набором аргументов.

Пример конструктора:

```
public class Person
{
    private string name;
    private int age;
    private bool isMale;

    // А вот и конструктор
```

```
    // Он принимает аргументы для инициализации полей
    public Person(string name, int age, bool isMale)
    {
        // Ключевое слово "this" указывает на текущий
экземпляр объекта.
        // Оно нужно, чтобы отличить поле класса
(this.name)
        // от параметра конструктора (name).
        this.name = name;
        this.age = age;
        this.isMale = isMale;
    }

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

Теперь при создании объекта мы *обязаны* передать в него аргументы.

Пример создания объекта с использованием конструктора:

```
static void Main(string[] args)
{
    Person ivan = new Person("Иван", 30, true);
    Person nastya = new Person("Анастасия", 25, false);

    ivan.SayHello(); // Выведет: Привет, я Иван!
    nastya.SayHello(); // Выведет: Привет, я Анастасия!
}
```

Модификаторы доступа и Свойства

Мы скрыли наши поля модификатором `private`. Это называется **инкапсуляция** — сокрытие внутренней реализации от внешнего

мира. Теперь мы не можем напрямую изменить имя или возраст объекта снаружи: `ivan.name = "Другое имя";` вызовет ошибку компиляции.

Как же получить контролируемый доступ к этим данным? Для этого существуют **свойства**.

Поле — хранит состояние объекта.
Свойство — обеспечивает контролируемый доступ к состоянию.

Свойство выглядит как поле, но под капотом имеет методы-аксессоры `get` (для чтения) и `set` (для записи).

Пример свойств:

```
public class Person
{
    private string name;
    private int age;

    // 1. Авто-свойство для имени.
    // Компилятор сам создаст скрытое приватное поле.
    public string Name { get; set; }

    // 2. Полное свойство для возраста с логикой
проверки.
    // Оно работает с нашим приватным полем age.
    public int Age
    {
        // get (геттер) - срабатывает при чтении значения
        get { return age; }

        // set (сеттер) - срабатывает при присваивании
значения.
        // Новое значение доступно через ключевое слово
"value".
        set
        {
            if (value ⩾ 0 && value < 120) // Возраст
должен быть в разумных пределах
```

```
            {
                age = value;
            }
            else
            {
                // Если значение некорректно, можно
ничего не делать или...
                // ...выбросить исключение (об этом
ниже).
                Console.WriteLine("Некорректный
возраст!");
            }
        }
    }

    // ... конструктор и другие методы ...
}
```

Теперь доступ к данным осуществляется через публичные свойства с большой буквы:

```
Person ivan = new Person("Иван", 30, true);
Console.WriteLine(ivan.Name); // Читаем значение через
get

ivan.Age = 31; // Записываем значение через set
ivan.Age = -5; // Выведет: "Некорректный возраст!" и
значение не изменится
```

Инициализатор объектов

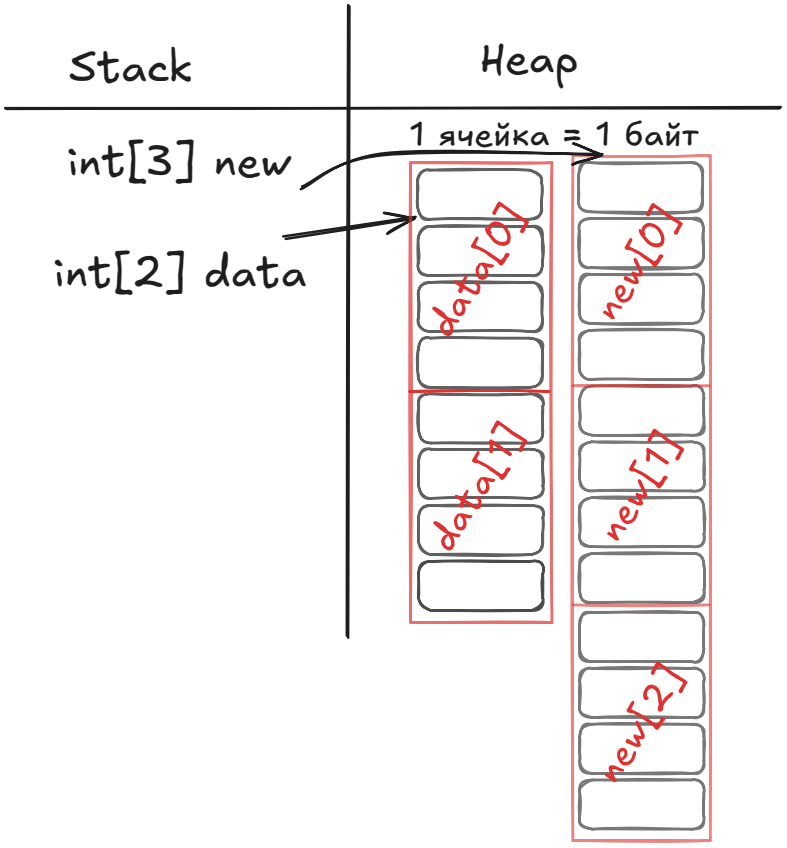
Если у класса есть публичный конструктор без параметров, можно использовать более короткий синтаксис для создания и инициализации объектов через их публичные свойства.

```
// Для этого в классе Person должен быть конструктор
public Person() {}
```

```
Person ivan = new Person { Name = "Иван", Age = 31 };
```

Напоминалка про ссылочные типы

Массивы как ссылочные типы: 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

Индексатор — это специальное свойство, которое именуется `this` и позволяет обращаться к объекту класса так, как будто он является массивом (используя квадратные скобки `[]`). Это удобно, если ваш класс внутри себя содержит коллекцию или массив.

```
public Тип_Возвращаемых_Данных this[int index]
{
    get
    {
        // логика для получения элемента по index
        return someArray[index];
    }
    set
    {
        // логика для установки элемента по index
        someArray[index] = value;
    }
}
```

Исключения

Что делать, если в программе происходит ошибка? Например, пользователь вводит некорректный возраст, или мы пытаемся обратиться к элементу массива по несуществующему индексу.

Исключения (`Exception`) — это стандартизированный механизм для обработки таких нештатных ситуаций. Вместо того чтобы возвращать код ошибки (`-1`), метод "выбрасывает" объект-исключение, сигнализируя о проблеме.

1. Выброс исключения (`throw`)

Улучшим сеттер нашего свойства `Age`. Вместо вывода в консоль, мы будем выбрасывать исключение, если возраст некорректен.

```
public int Age
{
    get { return age; }
    set
    {
        if (value ⩾ 0 && value < 120)
        {
            age = value;
```

```
        }
        else
        {
            // Выбрасываем исключение. Программа
немедленно прекратит
            // выполнение в этом месте и начнет искать
обработчик.
            throw new
ArgumentOutOfRangeException("Возраст должен быть в
диапазоне от 0 до 120.");
        }
    }
}
```

2. Обработка исключений (`try...catch`)

Код, который потенциально может выбросить исключение, помещается в блок `try`. Если ошибка действительно происходит, выполнение переходит в блок `catch`, где мы можем эту ошибку обработать.

```
static void Main(string[] args)
{
    Person person = new Person();

    try
    {
        Console.WriteLine("Пытаемся установить
некорректный возраст...");
        person.Age = -10; // Эта строка вызовет
исключение

        // Этот код не выполнится
        Console.WriteLine("Возраст успешно установлен.");
    }
    catch (ArgumentOutOfRangeException ex)
    {
        // Блок catch сработает, так как мы "поймали"
исключение нужного типа.
```

```
        Console.WriteLine("Произошла ошибка при установке
возраста!");
        Console.WriteLine($"Сообщение ошибки:
{ex.Message}");
    }
    finally
    {
        // Блок finally выполняется ВСЕГДА: и после try
(если ошибки не было),
        // и после catch (если ошибка была поймана).
        // Обычно используется для освобождения ресурсов.
        Console.WriteLine("Блок finally выполнен.");
    }

    Console.WriteLine("Программа продолжает свою
работу.");
}
```

Этот подход гораздо надежнее, так как он не позволяет проигнорировать ошибку и заставляет программиста явно её обрабатывать.