В.В. Подбельский

Использованы иллюстрации из книги Daniel Solis, Illustrated C#

Иллюстрации к курсу лекций по дисциплине «Программирование на C#»

08. Часть 1

ref local. ref return

Основы ООП.

Классы

Благодарность Д.Д. Сагалову за помощь в переработке презентаций

ref local – Ссылочные Локальные Переменные

Локальные ссылочные переменные позволяют создавать *псевдонимы* для других переменных/полей.

Синтаксис:
ref <Тип Переменной> <Идентификатор> = ref <Выражение>;

Важно: Локальные ссылочные переменные должны быть сразу инициализированы в момент их объявления. До C# 7.3 переприсваивание ref local переменных не допускалось.

Допускается присваивание локальным переменным/полям ссылочных возвращаемых значений или значений ссылочных локальных переменных, однако, в таком случае будет совершено копирование значения.

ref local: Пример

```
using System;

class RefLocalDemo {
    static void Main() {
        int x = 2, z = 3;
        ref int y = ref x;
        Console.WriteLine($"x = {x}, y = {y}");
        x = 5;
        Console.WriteLine($"x = {x}, y = {y}");
        y = 6;      // y – псевдоним x.
        Console.WriteLine($"x = {x}, y = {y}, z = {z}");
        y = ref z;  // C# 7.3 и новее позволяет менять ссылку.
        y = 10;
        Console.WriteLine($"x = {x}, y = {y}, z = {z}");
    }
}
```

Вывод:

x = 2, y = 2
x = 5, y = 5
x = 6, y = 6, z = 3
x = 6, y = 10, z = 10

ref return – Ссылки, как Возвращаемые Значения (C# 7)

ref return позволяет вернуть ссылку на переменную вместо её значения в качестве результата работы метода. Таким образом, вызывающий код сможет использовать результат работы метода как по ссылке, так и по значению.

Как вернуть ссылку на значение в C#:

- В сигнатуре метода перед типом возвращаемого значения указывается ключевое слово **ref**;
- В теле метода после каждого **return** добавляется **ref**;
- При вызове метода перед именем метода необходимо явно добавлять **ref**.

Ограничения ref return

Для возвращаемых по ссылке (ref return) значений существует ряд ограничений:

- Область видимости переменной, возвращаемой по ссылке, должна превышать область видимости метода. Иными словами, нельзя вернуть ссылку на локальную переменную этого же метода.

- Запрещено возвращать ссылку на this в структурах;

- Запрещается возвращать ссылки на: **null**, **константы**, **члены перечислений**, **свойства**, **readonly-поля**.

ref local и return ref: Пример

```
using System;
using System.Numerics;

BigInteger num1 = new BigInteger(long.MaxValue);
BigInteger num2 = new BigInteger(long.MinValue);
// Для возвращения значения по ссылке тоже нужно писать ref.
ref BigInteger num1_ref = ref LightWeightSwap(ref num1, ref num2);
--num1_ref; // ref-переменная служит псевдонимом для num1.
Console.WriteLine($"num1: {num1}; \t num2: {num2}");

static ref BigInteger LightWeightSwap(ref BigInteger lhs, ref BigInteger rhs) {
    lhs ^= rhs;
    rhs ^= lhs;
    lhs ^= rhs;
    return ref lhs; // После return требуется ref.
}
```

Вывод:
num1: -9223372036854775809
num2: 9223372036854775807

Объектно-Ориентированная Парадигма

Идея: представлять программу как множество объектов, взаимодействующих друг с другом. Кроме того, одни объекты могут быть подтипами других.

В рамках объектно-ориентированного программирования в C# следует рассматривать две неразрывно связанные сущности:

- **Типы данных** *(class, struct, record, enum)***;**
- **Объекты.**

Объектно-ориентированная парадигма предоставляет сразу несколько преимуществ:

- Представление сущностей как объектов довольно естественно для человека;
- Объекты могут иметь внутреннее устройство, скрытое от внешнего мира;
- Возникает возможность представлять одни типы как подтипы других (стол является видом мебели);
- Подтип всегда можно использовать там, где ожидается его базовый тип.

Типы Данных

Тип данных – описание того, из чего он состоит объект (данные) и того, что он умеет делать (функционал).

Когда мы определяем тип данных в программе, мы описываем, как будут выглядеть объекты данного типа:

```
public class NamedVector3D
{
    public string name;
    public int x;
    public int y;
    public int z;

    public void PrintCoords() => Console.WriteLine($"[{x}, {y}, {z}]");
}
```

Ключевое слово class + имя типа.

Данные, которые будут хранить в себе объекты данного типа.

Функционал, которым обладают все объекты данного типа.

На обсуждение: чего ещё стоит добавить в класс? Что не так с данным описанием?

Классы

Классы – ссылочные типы данных.

Классы могут быть объявлены в пространстве имён (включая глобальное, если иное не указано явно) или внутри других типов.

Важно: классы (как и другие типы), объявленные внутри пространств имён, по умолчанию *неявно* имеют **модификатор internal**.
При этом все члены классов по умолчанию имеют неявно **модификатор private**.

```
ключевое слово          имя класса
      ↓                     ↓
class MyLittleClass
{
    // <определение членов класса>…
}
```

По умолчанию имеет модификатор internal.

Объекты

Объекты – конкретные представители определённого типа, созданные по описанию класса. Часто объекты также называют **экземплярами** типа.

Для создания объектов в C# используется специальный синтаксис:

<Тип> <Идентификатор ссылки> = new <Тип>([Параметры]);

<Тип> <Идентификатор ссылки> = new([Параметры]); (*начиная с C# 9.0*)

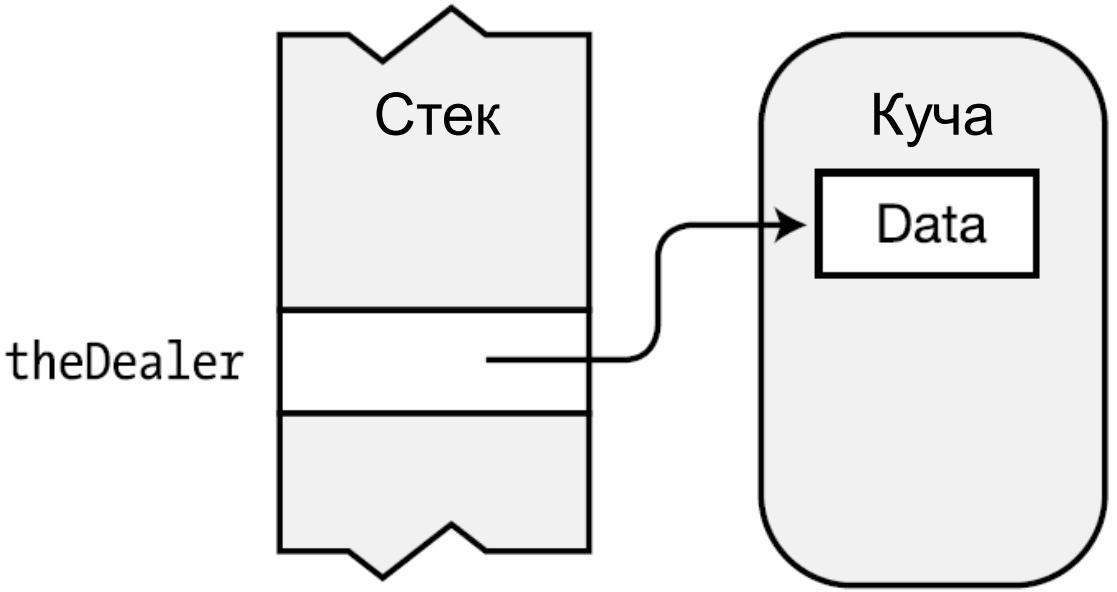
```
class Program
{
    static void Main(string[] args)
    {
        NamedVector3D v1 = new NamedVector3D();
        // C# 9.0: без явного указания типа после new:
        NamedVector3D v2 = new();
    }
}
```

Экземпляр Класса в Памяти

Как и любые другие ссылочные типы, экземпляры классов хранятся в куче:

```
class Dealer {...}

class App
{
    static void Main()
    {
        Dealer theDealer;
        theDealer = new Dealer();
                ...
    }
}
```

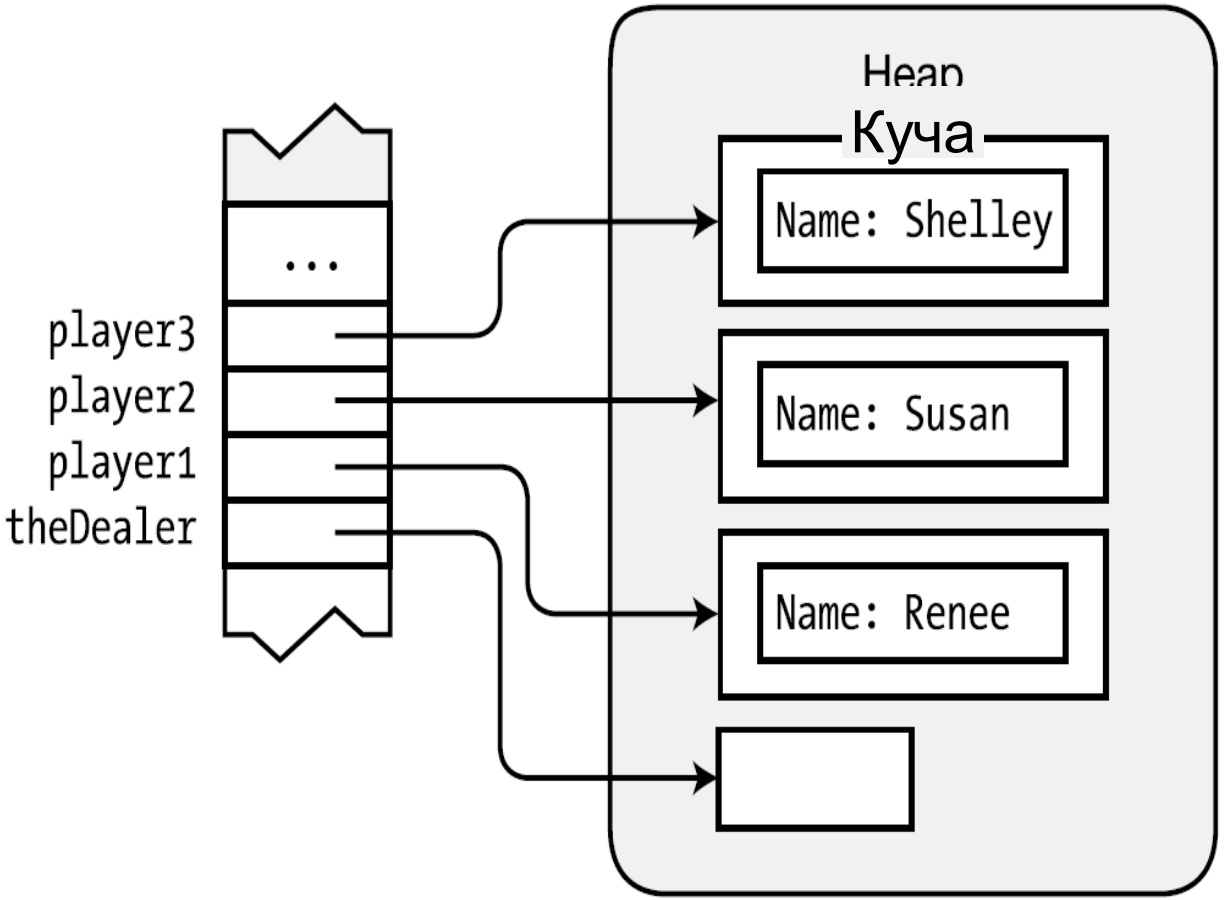


Несколько Экземпляров Классов в Памяти

```
// объявление класса
class Dealer { /*...*/ }

// объявление класса
class Player
{
    string Name; // поле
    // ...
}

class Program
{
    static void Main()
    {
        Dealer theDealer = new Dealer();
        Player player1 = new Player();
        Player player2 = new Player();
        Player player3 = new Player();
        // ...
    }
}
```



Инкапсуляция

Инкапсуляция – один из основных принципов ООП. При работе с объектами возникает необходимость ограничения доступа к внутренним составляющим. При этом клиентский код должен иметь минимально необходимый доступ к используемым возможностям объектов.

При взаимодействии друг с другом объекты зачастую обращаются к необходимым *функциональным членам*, а не к данным друг друга напрямую.

Вы можете различными способами **ограничить доступ к данным** в C#:

- Модификаторы доступа;
- Модификаторы readonly/const для осуществления доступа только для чтения;
- С помощью методов, аксессоров свойств/индексаторов/событий.

Модификаторы Доступа

Fields

```
AccessModifier Type Identifier
```

Methods

```
AccessModifier ReturnType MethodName ()
{
   ...
}
```

Модификаторы доступа – набор ключевых слов, описывающих доступ к типам/их членам:

- **public** – полный доступ из любой части кода;
- **protected internal** – полный доступ внутри текущей сборки, а также для всех наследников;
- **protected** – доступ для наследников из любой сборки;
- **internal** – доступ для любого типа в рамках текущей сборки;
- **private protected** – доступ только для наследников в текущей сборке;
- **private** – доступ только в том типе, в котором элемент определён.

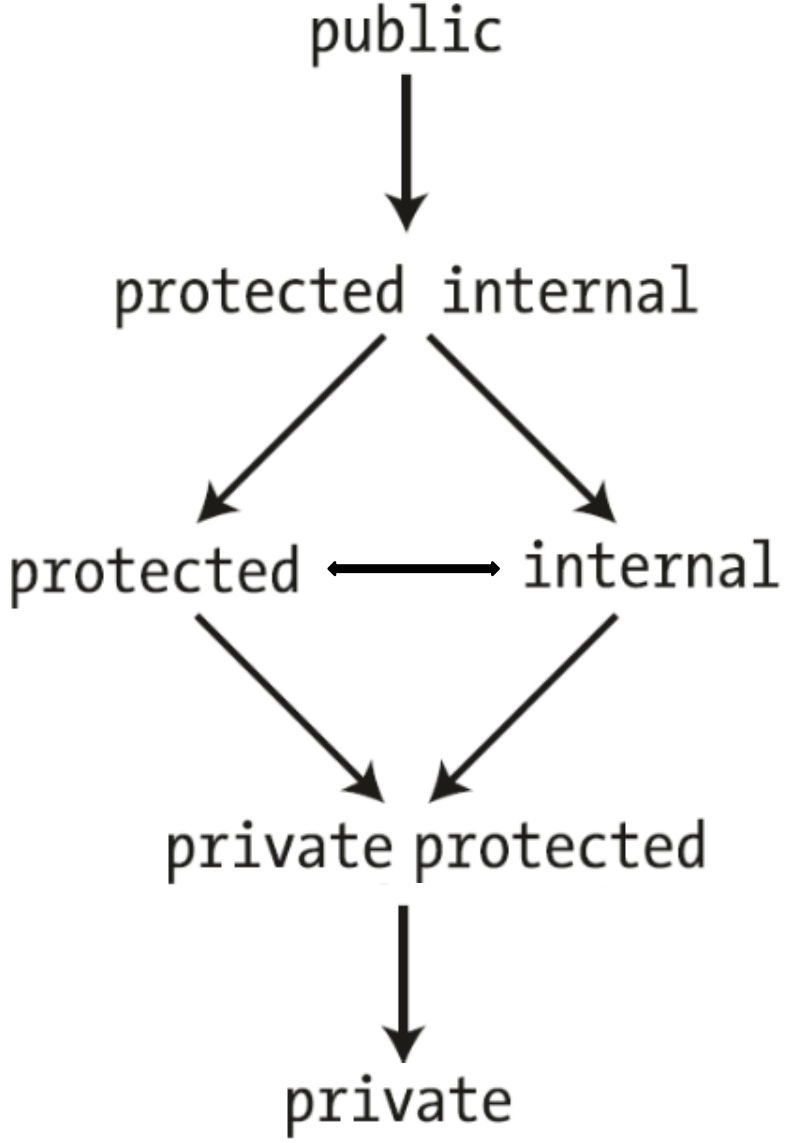
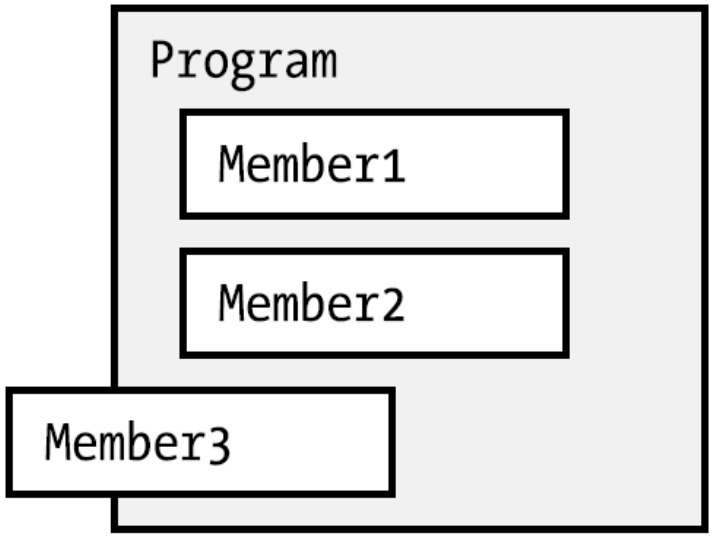


Схема сужения доступа

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers

Открытые и Закрытые Члены: Пример 1

```
class Program
{
    int Member1;
    private int Member2;
    public int Member3;
}
```



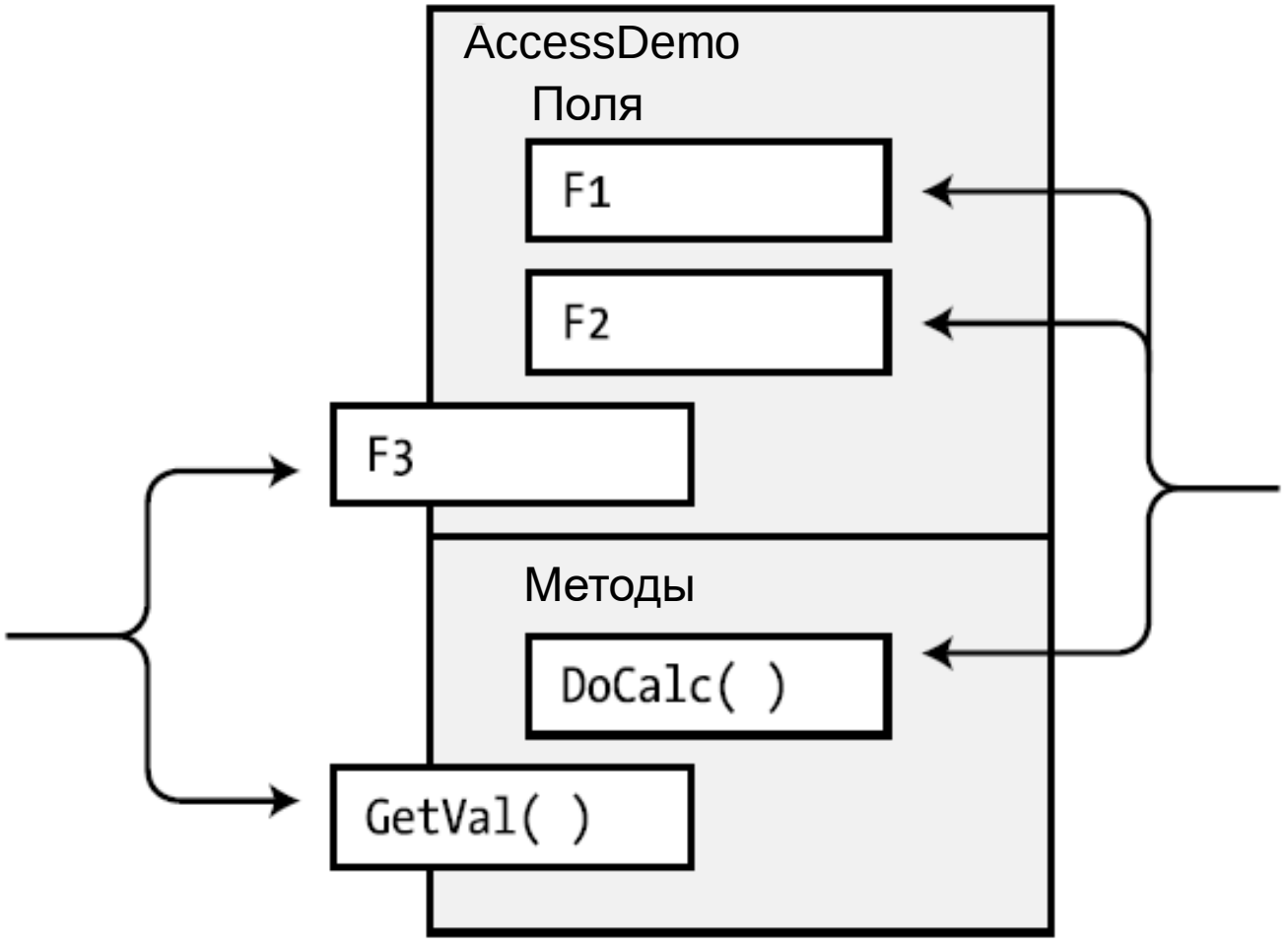
Открытые и Закрытые Члены: Пример 2

```
class AccessDemo
{
    int F1;             // Неявно закрытое поле.
    private int F2;     // Явно закрытое поле.
    public int F3;      // Открытое поле.

    void DoCalc()       // Неявно закрытый метод.
    {
        //...
    }

    public int GetVal() // Открытый метод.
    {
        //...
    }
}
```

Открытые и Закрытые Члены: Схема к Примеру 2



Доступ к открытым члены класса не ограничивается.

Закрытые члены класса доступны только внутри данного класса.

Что Входит в Типы Данных

Как уже упоминалось ранее, типы данных содержат в себе функциональные члены и данные.
Список всех возможных членов типов:

Данные	Функциональные Члены	
Поля*	***Методы****	Операции
Константы	Свойства	Индексаторы
	Конструкторы	События
	Деструкторы (Финализаторы)	

Обратите внимание: на уровне IL существуют только методы и поля! Остальные конструкции фактически существуют только для упрощения работы программиста.

Поля

Поля – переменные, определённые на уровне типов.
Важно: Глобальных переменных (на уровне пространства имен) в C# не существует!

Имеют набор важных отличий по сравнению с локальными переменными:

- Область видимости полей – весь тип;
- Поля никогда не бывают неинициализированными (без явного указания используется значение типа по умолчанию).

Тип MyType – internal. Program имеет доступ к value, т. к. они находятся в одной сборке.

```
class MyType {
    public int value;
}
class Program {
    static void Main(string[] args) {
        MyType mt = new();
        // Выведет 0 - значение int по умолчанию.
        System.Console.WriteLine(mt.value);
    }
}
```

Инициализация Полей

Допускается использование **инициализаторов** полей значениями по умолчанию – Вы можете явно после = указать некоторое выражение, результат которого будет использован при создании объекта:

```
class MyClass {
    int F1;                          // Инициализируется 0 - тип значения.
    string F2;                       // Инициализируется null - ссылочный тип.
    int F3 = 25;                     // Инициализируется 25.
    string F4 = "abcd";              // Инициализируется "abcd".
    Random rnd = new Random();       // Инициализируется объектом Random.
}

class MyClass2
{
    int F1, F3 = 25;
    string F2, F4 = "abcd";
}
```

Методы Классов (Повторение)

Как обсуждалось ранее, методы – один из основных способов определения функционала типов.

В отличие от C/C++ методы *не могут* объявляться вне типов данных (помним, что для top-level statements класс генерируется неявно).

Минимальный синтаксис объявления методов включает:

- Идентификатор метода;
- Тип возвращаемого значения;
- Список параметров в круглых скобках;
- Тело;

Методы Классов: Пример

```
using System;
class MethodClass {
```

тип возвращаемого значения — ↓ — список параметров ↓

```
    public void PrintNums() {
        Console.WriteLine("1");
        Console.WriteLine("2");
    }
}

class Program {
    static void Main() {
        MethodClass mc = new();
        mc.PrintNums();
    }
}
```

Создаём объект,
вызываем его метод.

Модификатор static

Модификатор **static** может применяться к:

- Классам (не к структурам!);
- Членам типов (как к функциональным, так и к данным).

Все члены, помеченные модификатором static принадлежат не к конкретным экземплярам, а к типу в целом.

Запомните: обращение к статическим членам осуществляется *по имени типа*, а не по ссылкам на конкретные экземпляры, и *не требует наличия* экземпляров соответствующего типа.

Фактически, статические члены инициализируются при первом обращении к типу; как и в случае с экземплярными полями, допускается указание инициализатора сразу после объявления.

Статические Классы

Статические классы могут содержать только статические члены.

Кроме этого, полностью запрещается создавать ссылки/экземпляры статических типов.

```
using System;

// 2 варианта ниже недопустимы и ведут к ошибке компиляции:
// Randomizer wrong1 = new();
// Randomizer wrong2;
public static class Randomizer
{
    private static Random _rnd = new Random();

    public static int GetIntInRange(int left = 0, int right = int.MaxValue) {
        return _rnd.Next(left, right);
    }
}
```

Сравнение: Статические Поля

```
using System;

DragonStatic dragon1 = new();
DragonStatic dragon2 = new();
Console.WriteLine($"Общее золото: {DragonStatic.gold}");
Console.WriteLine(dragon1.GetPersonalGold());
Console.WriteLine(dragon2.GetPersonalGold());
dragon1.SpendGold(80);
Console.WriteLine(dragon1.GetPersonalGold());
Console.WriteLine(dragon2.GetPersonalGold());

public class DragonStatic
{
    // Золото общее для всех объектов.
    public static int gold = 100;

    public int GetPersonalGold() { return gold; }
    public void SpendGold(int amount) { gold -= amount; }
}
```

2 независимых дракона.

Обращение по имени типа!

Тратится общее для всех золото

Вывод:

Общее золото: 100
100
100
20
20

Сравнение: Поля объекта (экземплярные)

```
using System;

Dragon dragon1 = new();
Dragon dragon2 = new();
// Console.WriteLine($"Общее золото: {Dragon.gold}");
Console.WriteLine(dragon1.GetPersonalGold());
Console.WriteLine(dragon2.GetPersonalGold());
dragon1.SpendGold(80);
Console.WriteLine(dragon1.GetPersonalGold());
Console.WriteLine(dragon2.GetPersonalGold());

public class Dragon
{
    // Золото своё у каждого дракона.
    public int gold = 100;

    public int GetPersonalGold() { return gold; }
    public void SpendGold(int amount) { gold -= amount; }
}
```

Данная строка не скомпилируется, золото есть у каждого экземпляра, а не на уровне типа.

Только 1-ый дракон тратит золото

Вывод:
100
100
20
100

Конструкторы Классов – Введение

Конструктор – специальный функциональный член, позволяющий создавать объекты данного типа. Конструкторы позволяют описать обязательные параметры, необходимые для создания экземпляров.

Синтаксис описания конструктора похож на синтаксис описания других методов:

```
[Модификаторы] <Идентификатор Типа> ([Параметры]) {[Тело]}
```

Конструкторы имеют ряд особенностей:

- Их имя обязано совпадать с именем типа;
- Нет типа возвращаемого значения, они всегда возвращают ссылку на созданный объект;
- Могут быть перегружены, аналогично другим методам;
- При отсутствии явных определений конструктора, компилятор генерирует конструктор без параметров, инициализирующий поля значениями их типов по умолчанию. При наличии хотя бы одного явно определённого конструктора, конструктор по умолчанию не генерируется.

Конструкторы: Пример 1

```
Person person1 = new("Stevie");
Person person2 = new Person("Eloy", 23);
// Person incorrectPerson = new(); - ошибка: нет конструктора без параметров.

public class Person
{
    public string name;
    public int age = 1;

    public Person(string personName) {
        name = personName;
    }

    public Person(string personName, int personAge) {
        name = personName;
        age = personAge;
    }
}
```

Значение возраста по умолчанию – 1.

Конструктор с 1 параметром.

Конструкторы: Пример 2

Вывод:
2,75
10

```
TemperaturesAverage avg = new(1, 2, 3, 5);
System.Console.WriteLine(avg.CalculateAverage());
avg = new(5, 10, 15);
System.Console.WriteLine(avg.CalculateAverage());

public class TemperaturesAverage {
    private double[] _temperatures;

    public TemperaturesAverage(params double[] temperatures) {
        return _temperatures = temperatures ?? new double[0];
    }
    public double CalculateAverage() {
        double sum = 0;
        foreach (double temperature in _temperatures) {
            sum += temperature;
        }
        return sum / (_temperatures.Length == 0 ? 1 : _temperatures.Length);
    }
}
```

При явной передаче null в конструктор будет создан пустой массив.

Обработка сценария с потенциальным делением на 0.

Статический Конструктор

Статический конструктор – специальный вариант конструктора, который нужен для инициализации статических членов типа.

Синтаксис описания статического конструктора:

```
static <Идентификатор Типа>() {[Тело]}
```

Статический конструктор имеет ряд особенностей:

- Вызывается неявно ровно один раз за исполнение программы при первом обращении к статическим членам типа или при первом создании объекта. Не может быть вызван явно;
- Не может иметь модификаторов доступа;
- Не имеет параметров и не может быть перегружен – единственный в типе;
- Выполняется строго после инициализаторов всех статических полей (сами инициализаторы отрабатывают в порядке их объявления в файле).

Статический Конструктор: Пример

```
using System;

Console.WriteLine(StaticConstructorDemo.GetTimestamp());

public class StaticConstructorDemo {
    static int classId = 101;
    static DateTime referenceTime;

    static StaticConstructorDemo() {
        // OK - поле уже инициализировано.
        Console.WriteLine($"ID: {classId}");
        referenceTime = DateTime.Now;
    }

    public static string GetTimestamp() {
        return $"{referenceTime}-{classId}";
    }
}
```

Вариант вывода:
ID: 101
24.10.2021 15:14:10-101

Класс запоминает момент вызова статического конструктора.

ref return: Пример с Классом

```
using System;

Simple s = new Simple();
s.Display();
ref int v1Outside = ref s.RefToValue();
v1Outside = 10; // Изменяем значение по ссылке.
s.Display();    // Проверка изменений.

class Simple {
    private int Score = 5;

    // Осторожно: серьёзное нарушение инкапсуляции!
    public ref int RefToValue() { return ref Score; }

    public void Display() { Console.WriteLine($"Simple value: {Score}"); }
}
```

Вывод:
Simple value: 5
Simple value: 10