Введение в язык программирования C#

Описание языка C#

Язык программирования C# является объектно-ориентированным языком программирования высокого уровня. Автором языка является Андерс Хейлсберг, который был архитектором Turbo Pascal и Borland Delphi.
Язык C# относится к C-подобным языкам программирования по синтаксису.
Изначально язык C# был разработкой для работы на прикладном уровне с CLR (Common Language Runtime) – среда, компилирующая программы, которые написаны на языках .NET (Visual C#, Visual Basic, Visual F#, Visual C++ и др.)
C# стандартизирован в ECMA (ECMA-334) и ISO (ISO/ IEC 23270).

Типы данных языка C#

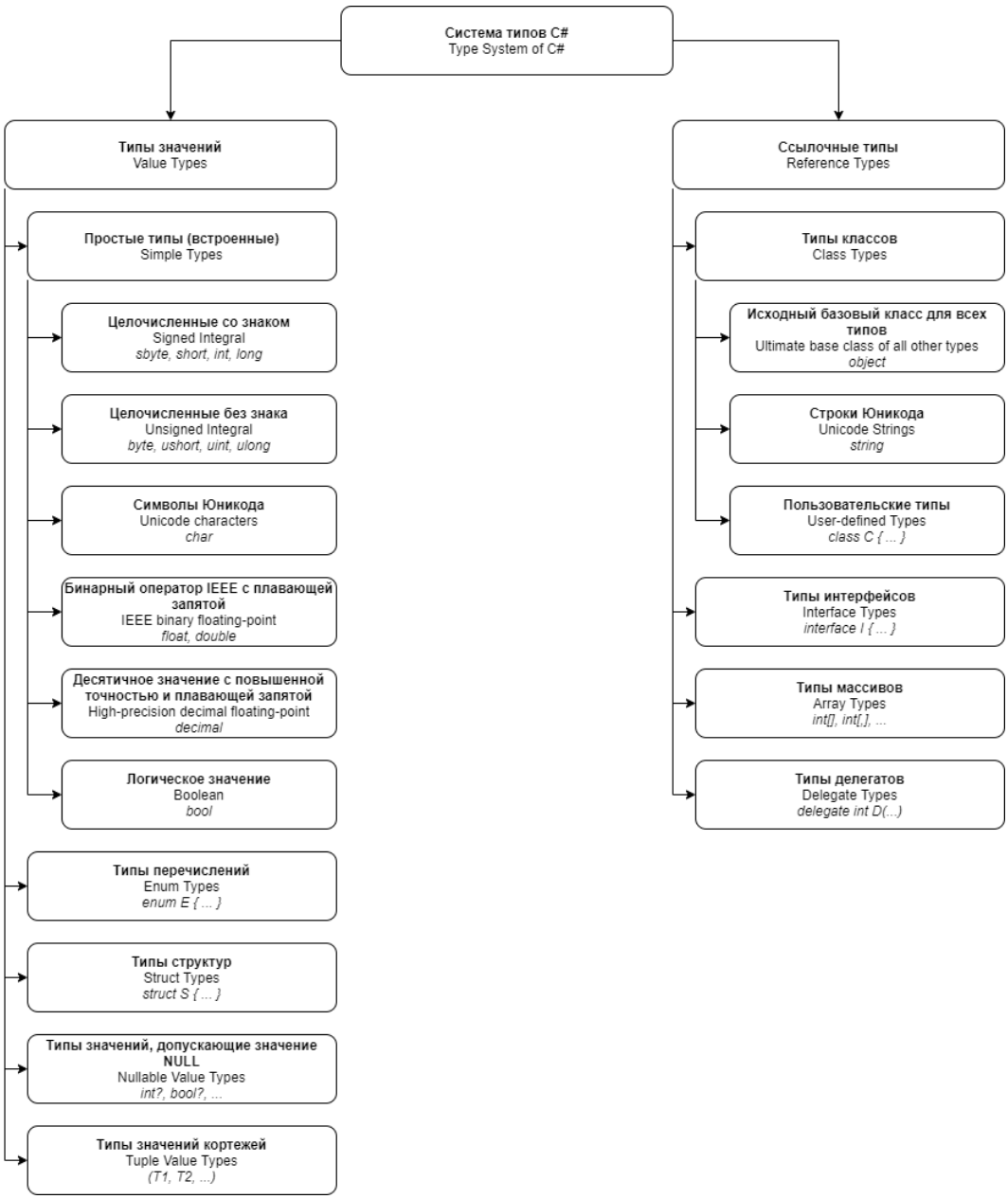

Рисунок 1 - Система типов в языке C#

Типы данных языка программирования C# разделяются на две разновидности: типы значений [знаковые] (переменные таких типов хранят в себе значения) и ссылочные типы (в переменной такого типа будет храниться ссылка на объекты [данные]).

Структура программы

```
using System; // Подключение библиотек
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using static System.Console; // Доступ к статическим членам и вложенным типам

namespace _01_Structure // Пространство имен (Название проекта)
{
    class Program // Описание класса
    {
        static void Main(string[] args) // Метод Main
        {
            WriteLine("Hi, STANKIN!");
            ReadLine();

            Console.WriteLine("Программисты тут все чтоли?");
            Console.ReadLine();

            string lines = "Строка 1 \nСтрока 2\nСтрока 3";

            using (var reader = new StringReader(lines)) // Оператор using (statement)
            {
                var item = reader.ReadLine();
                while (item != null)
                {
                    Console.WriteLine(item);
                    Thread.Sleep(3000);
                    item = reader.ReadLine();
                }
            }
            Console.ReadLine();
        }
    }
}
```

Using. Инструкция и директива

Ключевое слово using используется, как правило, в трех случаях:

- Необходимо подключить библиотеку (directive using). Действует до конца файла.
- Необходимо получить доступ к статическим членам и вложенным типам без указания их имен (directive using static). Действует до конца файла.
- Необходимо получить один или несколько ресурсов, выполнить инструкции и уничтожить ресурс (using statement) – оператор using. Действует до закрывающейся фигурной скобки (до версии языка C# 8.0)

Директива #region

В случае необходимости создания удобочитаемости кода в виде сворачивания и разворачивания его элементов в редакторе кода, можно использовать директиву #region.

```
#region NameOfRegion
// Код
#endregion
```

Тип данных Object. Упаковка и распаковка [Boxing-Unboxing]

Все типы данных в языке C# (в том числе встроенные и написанные самостоятельно) являются неявными наследниками класса System.Object. Именно поэтому, все классы, написанные самостоятельно уже имеют методы, которые определены в классе Object (ToString(), GetHashCode(), …)

Упаковка [Boxing] – процесс неявного преобразования типа значения в тип класса Object.

```
int mark = 25; // в переменной mark лежит значение 25
object markObject = mark;
// неявное преобразование типа значения int в ссылочный тип object
```

Распаковка [Unboxing] – процесс извлечения типа значения из объекта (явное преобразование). При данной процедуре необходимо проверить экземпляр объекта на то, что он был упакован значением заданного типа данных, а уже после применяется процедура копирования значения в переменную типа значения.

```
object valueObject = 54;
int value = (int)valueObject;
// явное преобразование в тип значения int из ссылочного типа object
```

Процедура упаковки-распаковки может потребоваться в том случае, если необходимо избежать повторяющегося кода или смешать объекты разных типов в одной коллекции. Стоит отметить, что Boxing-Unboxing – это довольно медленные операции.

```
object[] trash = new object[3];
trash[0] = 24; // упаковка int
trash[1] = -54.01m; // упаковка decimal
trash[2] = "Когда экзамен?";
// в массиве типа object могут быть и объекты ссылочного типа
```

Тип данных Struct

Тип структуры является типом значения (переменная такого типа содержит экземпляр этого типа). В структуре могут содержаться элементы-данные и элементы-функции.

```
public struct Person
{
    string _name; // private field (приватное поле)
    string _lastName; // private field (приватное поле)

    /// <summary>
    /// Constructror. Конструктор
    /// </summary>
    /// <param name="name">Name of Person. Имя</param>
    /// <param name="lastName">Last Name of Person. Фамилия</param>
    public Person(string name, string lastName)
    {
        _name = name;
        _lastName = lastName;
    }



    /// <summary>
    /// Get Full Name. Получить полное имя
    /// </summary>
```

```
    /// <returns>Full Name (Name + Last Name). Полное имя</returns>
    public string GetFullName()
    {
        return $"Name: {_name}; Last Name: {_lastName}";
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person lecturer = new Person("Nikita", "Kashurkin"); // instance (представитель
структуры)
        Console.WriteLine(lecturer.GetFullName());  // method call (вызов метода)
    }
}
```

Все элементы, инкапсулированные внутри структуры, по умолчанию имеют модификатор доступа private. Поэтому нет необходимости писать этот модификатор доступа перед объявлением локальной переменной.

Наследование для структур

Так как структура относится к типу значений, она не может выступать членом наследования (ни базовым, ни наследуемым), но она может реализовывать интерфейс.

Поля структуры

При описании поля в структуре нельзя его одновременно инициализировать, если оно не является статическим.

Конструктор структуры

Если в структуре не описан конструктор, будет автоматически создан конструктор по умолчанию без параметров. При описании структуры нет возможности самостоятельно реализовать конструктор без параметров. Более того, в конструкторе должны инициализироваться все поля структуры.

Деструктор структуры

При реализации структуры нельзя объявить деструктор.

Что лучше использовать: структуру или класс?

https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/choosing-between-class-and-struct

Массивы

В языке C# массив представляет собой указатель на непрерывный участок памяти. Другими словами, на этом языке имеются только динамические массивы.

Массив рассматривается, как класс, что позволяет пользоваться его методами и свойствами.

Объявление одномерного массива

```
DataType[] name; // Тип_данных[] наименование_массива;
```

Объявление многомерного массива

```
DataType[,] name; // Тип_данных[,] наименование_массива;
```

Перед использованием массива он должен быть инициализирован, то есть под него должна быть выделена память.

Одномерные массивы

```
int[] myArray; // Объявление массива
myArray = new int[5]; // Выделение памяти под массив на 5 элементов
```

```
// Заполнение массива
for (var i = 0; i < myArray.Length; i++)
{
    Console.Write($"myArray[{i}] = ");
    myArray[i] = Convert.ToInt32(Console.ReadLine());
}

int sum = 0;
foreach (var item in myArray) // Оператор foreach
{
    sum += item;
}

// Вывод массива
for (var i = 0; i < myArray.GetLength(0); i++)
{
    Console.WriteLine($"myArray[{i}] = {myArray[i]}");
}

Console.WriteLine($"Sum = {sum}");
Console.ReadLine();
```

Массив является ссылочными типом данных.

```
// Копирование массивов
int[] secondArray = myArray;
secondArray[1] = myArray[1] - 200;

Console.WriteLine("myArray:");
PrintArray(myArray);
Console.WriteLine("secondArray:");
PrintArray(secondArray);
Console.ReadLine();

int[] thirdArray = new int[secondArray.Length];
secondArray.CopyTo(thirdArray, 0); // Корректное копирование массивов

thirdArray[1] = secondArray[1] - 200;

Console.WriteLine("thirdArray:");
PrintArray(thirdArray);
Console.ReadLine();
```

Многомерные массивы:

```
decimal[,] array = new decimal[4, 5];
Random random = new Random();

for (var i = 0; i < array.GetLength(0); i++)
    for (var j = 0; j < array.GetLength(1); j++)
    {
        array[i, j] = Math.Round((random.Next(1050, 3000) / 3.0m), 2);
    }

for (var i = 0; i < array.GetLength(0); i++)
{
    for (var j = 0; j < array.GetLength(1); j++)
    {
        Console.Write(array[i, j] + "\t");
    }
    Console.WriteLine();
}
Console.ReadLine();
```

Рваные (ступенчатые) массивы.

Массив массивов (Jagged Array). Вид массива, элементами которого могут являться массивы разных размерностей. Такие массивы также называют массивами массивов.

```
int[][] array = new int[3][]; // declaration

array[0] = new int[3];
array[0][0] = 11; // initialization
array[0][1] = 12;
array[0][2] = 13;

array[1] = new int[5] { 21, 22, 23, 24, 25 }; // initialization

array[2] = new int[] { 31, 32 }; // initialization

for (var i = 0; i < array.Length; i++)
{
    for(var j = 0; j< array[i].Length; j++)
    {
        Console.Write($"[{i}][{j}] = {array[i][j]}\t");
    }
    Console.WriteLine();
}
Console.WriteLine();

for (var i = 0; i < array.Length; i++)
{
    foreach(var item in array[i])
    {
        Console.Write($"{item}\t");
    }
    Console.WriteLine();
}
Console.ReadLine();
```

Коллекции

Для эффективного управления и обработки данных рекомендуется организовывать коллекции. Типы коллекций – это виды коллекций данных, таких как хэш-таблицы, очереди, стеки, контейнеры, словари и списки.

Список List<T>

Тип данных «Список» (List<T>), реализованный в C#, представляет собой строго типизированный список объектов, доступных по индексу. Поддерживает методы для поиска по списку, выполнения сортировки и других операций со списками.
Полный перечень методов для работы со списками можно получить, если ввести название списка и поставить точку. Среда программирования сама выведет данный перечень доступных методов.
Списки, закрытые встроенным типом данных

```
List<int> myList = new List<int>(); // Список, закрытый встроенным типом данных int
Random random = new Random();

// Наполнение списка
for (var i = 0; i < 10; i++)
{
    myList.Add(random.Next(-100, 100)); // Добавление элемента в конец списка
}

Console.WriteLine("Заполненный список:");
foreach (var item in myList)
{
    Console.WriteLine(item);
```

```
}

Console.WriteLine($"Максимальный элемент списка равен: {myList.Max()}");
Console.WriteLine("Отсортированный список:");
foreach (var item in myList.OrderByDescending(x => x))
{
    Console.WriteLine(item);
}
Console.WriteLine($"Сумма элементов списка: {myList.Sum()}");

Console.WriteLine("Неотрицательные элементы списка:");
foreach (var item in myList.Where(el => el >= 0))
{
    Console.WriteLine(item);
}

Console.ReadLine();
```

Списки, основанные на написанных программистом типах

```
List<Person> students = new List<Person>(); // Произвольный тип Person

var student1 = new Person("Владислав", "Моисеев");
students.Add(student1);

students.Add(new Person("Денис", "Комарков"));
students.Add(new Person() { Name = "Никита", LastName = "Паничев" });

var student4 = new Person();
Console.Write("Введи фамилию:");
student4.LastName = Console.ReadLine();
Console.Write("Введи имя:");
student4.Name = Console.ReadLine();

students.Add(student4);

Console.WriteLine("Студенты:");
students.ForEach(s => Console.WriteLine(s.GetFullName()));

Console.ReadLine();
```