Class Object. Type Declarations. Class

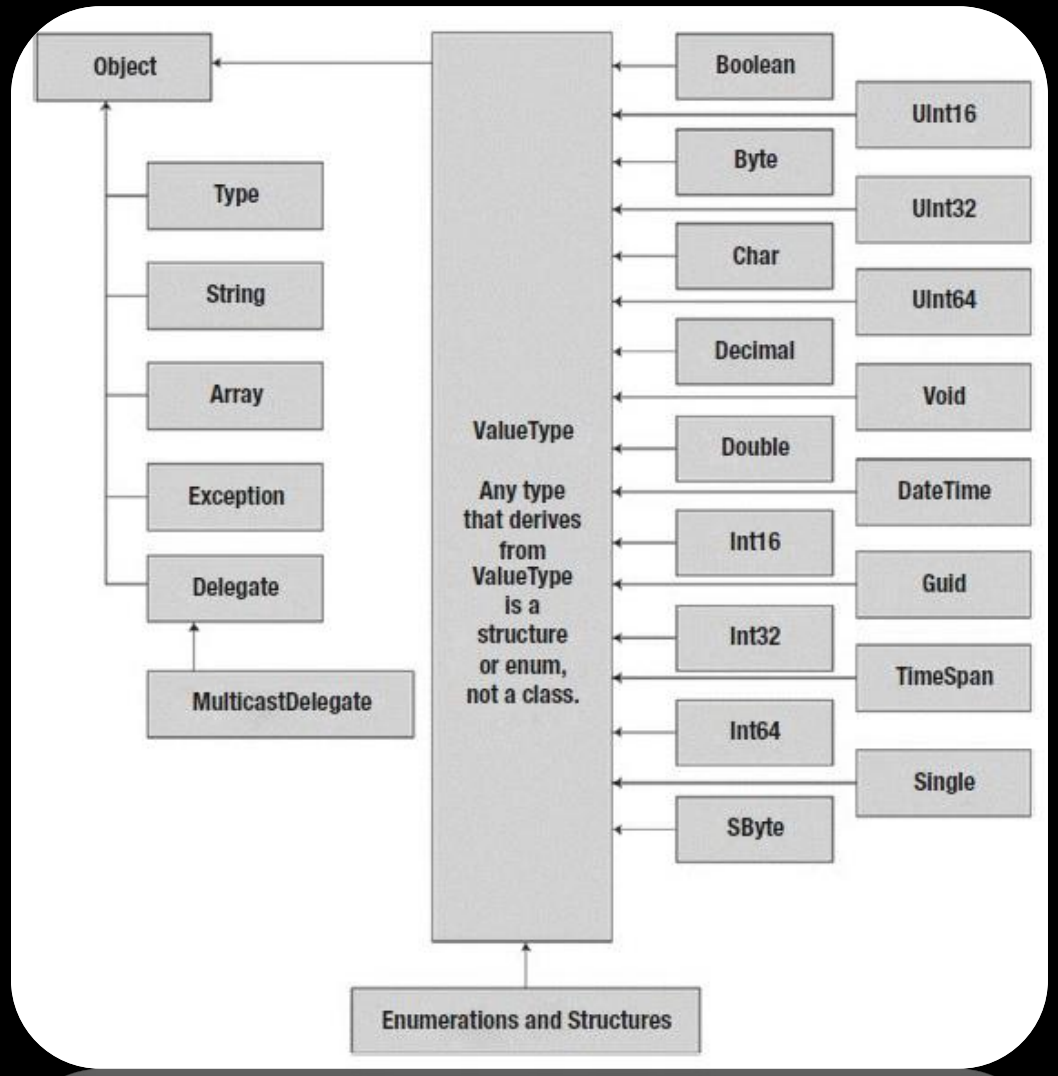
By Ira Zavushchak

softserve

Agenda

- Data Type Class Hierarchy
- Base class Object
- Class Declaration
- Value and Reference Types

The Data Type Class Hierarchy



softserve

System. Object class

- **ToString -** method is used to get a string representation of this object. For base types, their string value will simply be displayed:

```
int i = 5;
Console.WriteLine(i.ToString()); // число 5

double d = 3.5;
Console.WriteLine(d.ToString()); // число 3,5
```

- **GetHashCode -** method allows to return some numeric value that will correspond to a given object or its hash code. By this number, for example, you can compare objects. You can define a variety of algorithms for generating a similar number or take the implementation of the basic type:

```
class Person
{
    public string Name { get; set; }

    public override int GetHashCode()
    {
        return Name.GetHashCode();
    }
}
```

- **GetType** - method allows to get the type of object:

```
Person person = new Person { Name = "Tom" };
Console.WriteLine(person.GetType());    // Person
```

- **Equals** - method allows to compare two objects for equality:

```
Person person1 = new Person { Name = "Tom" };
Person person2 = new Person { Name = "Bob" };
Person person3 = new Person { Name = "Tom" };
bool p1Ep2 = person1.Equals(person2);   // false
bool p1Ep3 = person1.Equals(person3);   // true
```

Class Declaration

```
<access specifier> class ClassName
 {
       // fields
       <access specifier> <data type> variable1;

       // member methods
       <access specifier> <return type> Method1(parameter_list)
       { // method body }

      // Methods, properties, fields, events, delegates
      // and nested classes go here.
}
```

```
1  using System;
2
3  namespace HelloApp
4  {
5      class Person
6      {
7
8      }
9      class Program
10     {
11         static void Main(string[] args)
12         {
13
14         }
15     }
16 }
```

Class Declaration. Access specifier:

public: a public class or member of a class. Such a class member is accessible from anywhere in the code, as well as from other programs and assemblies.

private: the private class or member of the class. Represents the exact opposite of the public modif ss or contex

```
1  public class State
2  {
3      int a; // рівносильно private int a
4      private int b; // поле доступне тільки із поточного класу
5      protected int c; //доступне з поточного і похідних класів
6      internal int d; // доступне в будь-якому місці програми
7      protected internal int e; // доступне в будь-якому місці програми та із класів-наслідників
8      public int f; //доступне в будь-якому місці програми, а також для інших програм і збірок
9      protected private int g; //доступне з поточного класу і похідних класів, які визначені в цьому ж проекті
10
```

prote rived classe

interr code in the with the pu

protected internal: combines the functionality of two modifiers. Classes and class members with this modifier are accessible from the current assembly and from derived classes.

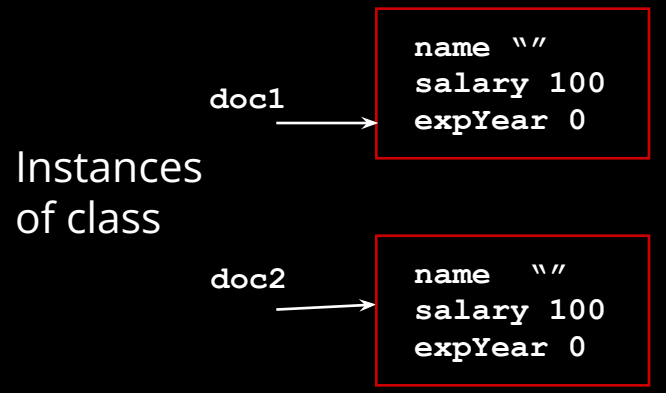
private protected: this class member is accessible from anywhere in the current class or in derived classes that are defined in the same assembly.

Class declaration. Fields

- A *field* is a variable of any type that is declared directly in a class or struct.
- Use *fields* only for variables that have ***private*** or ***protected*** accessibility
- A *field* can be initialized in declaration.

```
public class Doctor
{
 private string name;
 private double salary =100;
 private int    expYear;
   ...
}
```

```
Doctor doc1 = new Doctor();
Doctor doc2 = new Doctor();
```



Static, readonly and constants fields

- *static field* and *constant* belong to the class itself, and are shared among all instances of that class.
- only C# built-in types (and `string` or `enum`) may be declared as const.
- constants must be initialized as they are declared and do not change for the life of the program
- readonly field is const for instance of the class, can be initialized in declaration or in constructor only

```
1  class Account
2  {
3      public static decimal bonus = 100;
4      public decimal totalSum;
5      public Account(decimal sum)
6      {
7          totalSum = sum + bonus;
8      }
9  }
```

```
10 class Program
11 {
12     static void Main(string[] args)
13     {
14         Console.WriteLine(Account.bonus);      // 100
15         Account.bonus += 200;
16
17         Account account1 = new Account(150);
18         Console.WriteLine(account1.totalSum);   // 450
19
20
21         Account account2 = new Account(1000);
22         Console.WriteLine(account2.totalSum);   // 1300
23
24         Console.ReadKey();
25     }
26 }
```

Readonly and constants fields

```
class MathLib
{
    public const double PI=3.141;
    public const double E = 2.81;
    public const double K;  // Error, константа не ініціалізована
}

class Program
{
    static void Main(string[] args)
    {
        MathLib.E=3.8; // Error, значення константи не можна змінити
    }
}
```

```
class MathLib
{
    public readonly double K = 23;  // Можна так

    public MathLib(double _k)
    {
        K = _k; // поле для читання може бути ініціалізовано в конструкторі після  компіляції
    }
    public void ChangeField()
    {
        // Так не можна!
        //K = 34;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        MathLib mathLib = new MathLib(3.8);
        Console.WriteLine(mathLib.K); // 3.8

        //mathLib.K = 7.6; // поле для читання не можна встановити за межами класу
        Console.ReadLine();

    }
}
```

Constructors

- In addition to the usual methods in classes, special methods are also used, which are called constructors. Constructors are called when creating a new object of this class. Designers perform object initialization.

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

    public Person() { name = " Невідомий "; age = 18; }      // 1 конструктор

    public Person(string n) { name = n; age = 18; }          // 2 конструктор

    public Person(string n, int a) { name = n; age = a; }    // 3 конструктор

    public void GetInfo()
    {
        Console.WriteLine($"Ім'я: {name} Вік: {age}");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Person tom = new Person();
        tom.GetInfo();      // Ім'я:  Вік: 0

        tom.name = "Tom";
        tom.age = 34;
        tom.GetInfo();  // Ім'я: Tom Вік: 34

        Console.Read();
    }
}
```

Keyword «this»

- The keyword this represents a link to the current instance of the class.

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

    public Person() : this("Невідомий" )
    {
    }
    public Person(string name) : this(name, 18)
    {
    }
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void GetInfo()
    {
        Console.WriteLine($"Ім'я: {name} Вік: {age}" );
    }
}
```

Properties

- In addition to the usual methods in the C # language, there are special access methods called *properties*. They provide easy access to the fields of the class, find out their meaning or install them.

```
class Person
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }
}
```

- Property is a member that provides a flexible mechanism to read, write, or compute the value of a private field.
- Properties can be used as if they are public data members, but they are actually special methods called ***accessors***.

Methods

- *Methods* are declared in a class or struct by specifying the access level, the return value, the name of the method, and any method parameters.

```
class Calculator
{
    public void Add(int a, int b)
    {
        int result = a + b;
        Console.WriteLine($"Result is {result}");
    }
    public void Add(int a, int b, int c)
    {
        int result = a + b + c;
        Console.WriteLine($"Result
    }
    public int Add(int a, int b, in
    {
        int result = a + b + c + d;
        Console.WriteLine($"Result is {result}");
        return result;
    }
    public void Add(double a, double b)
    {
        double result = a + b;
        Console.WriteLine($"Result is {result}");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        ator calc = new Calculator();
        dd(1, 2); // 3
        d(1, 2, 3); // 6
        (1, 2, 3, 4); // 10
        (1.4, 2.5); // 3.9

        Console.ReadKey();
    }
}
```

```
Result is 3
Result is 6
Result is 10
Result is 3.9
```

softserve

Parameter Modifiers

C# Parameter Modifiers

Parameter Modifier	Meaning in Life
(None)	If a parameter is not marked with a parameter modifier, it is assumed to be passed by value, meaning the called method receives a copy of the original data.
`out`	Output parameters must be assigned by the method being called, and therefore, are passed by reference. If the called method fails to assign output parameters, you are issued a compiler error.
`ref`	The value is initially assigned by the caller and may be optionally reassigned by the called method (as the data is also passed by reference). No compiler error is generated if the called method fails to assign a `ref` parameter.
`params`	This parameter modifier allows you to send in a variable number of arguments as a single logical parameter. A method can have only a single `params` modifier, and it must be the final parameter of the method. In reality, you might not need to use the `params` modifier all too often; however, be aware that numerous methods within the base class libraries do make use of this C# language feature.

Example


```
static void Main ( string [ ] args )
{
    int arg;
    arg = 4;
    // Passing by value. The value of arg in Main is not changed.
    squareVal ( arg );
    Console.WriteLine ( arg ); // Output: 4

    arg = 4;
    // Passing by reference.  The value of  arg in Main is changed.
    squareRef ( ref arg );

    Console.WriteLine ( arg );       // Output: 16
}
```

```
static void squareVal ( int valParameter
)
   {
      valParameter *= valParameter;
   }
```

```
 static void squareRef( ref int refParameter
)
   {
      refParameter *= refParameter;
   }
```

Operator Overloading

```
public static rettype operator op(param1 [,param2])
{…}
```

- **Only some operators can be overloaded:**

  unary: `+ , - , !, ~, ++, --, true, false`
  binary: `+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=`

- **Some operators should be overloaded in pair:**

  `==` and `!=`
  `>` and `<`
  `>=` and `<=`
  `true` and `false`

Operator Overloading. Example

```
public class Doctor
    {
        private string name;
        private double salary = 100;
        private int expYear;
        public static bool operator == (Doctor first, Doctor second)
        {
            return first.name == second.name;
        }
        public static bool operator !=(Doctor first, Doctor second)
        {
            return !(first == second);

        }
    }
```

```
static void Main(string[] args)
        {
            Doctor a = new Doctor();
            Doctor b = new Doctor();
            if (a == b) Console.WriteLine("The same names");
            else Console.WriteLine("Not the sane names");

        }
```

Conversion Operators

- Classes or structs can be converted to and/or from other classes or structs, or basic types
- Conversions are defined like **operators** and are named for the type to which they convert.
- Conversions declared **as implicit** occur automatically when it is required.
- Conversions declared **as explicit** require a cast to be called.
- All conversions must be declared as static.

```
public static implicit operator conv-type-out (conv-type-in operand)

public static explicit operator conv-type-out (conv-type-in operand)
```

Conversion Operators. Example

```
//in Doctor class
public static explicit operator Doctor(string newName)
{
      Doctor temp = new Doctor();
      temp.name = newName;
      return temp;
}
 public static implicit operator string(Doctor doc)
{
      return doc.ToString();
}
```

```
static void Main(string[] args)
        {
            Doctor a = new Doctor();
            string doc = a;
            Doctor c = (Doctor)"Aibolit";

        }
```

Value and Reference Types

Intriguing Question	Value Type	Reference Type
Where are objects allocated?	Allocated on the stack.	Allocated on the managed heap.
How is a variable represented?	Value type variables are local copies.	Reference type variables are pointing to the memory occupied by the allocated instance.
What is the base type?	Implicitly extends `System.ValueType`.	Can derive from any other type (except `System.ValueType`), as long as that type is not "sealed" (more details on this in Chapter 6).
Can this type function as a base to other types?	No. Value types are always sealed and cannot be inherited from.	Yes. If the type is not sealed, it may function as a base to other types.
What is the default parameter passing behavior?	Variables are passed by value (i.e., a copy of the variable is passed into the called function).	For value types, the object is copied-by-value. For reference types, the reference is copied-by-value.
Can this type override `System.Object.Finalize()`?	No. Value types are never placed onto the heap and, therefore, do not need to be finalized.	Yes, indirectly (more details on this in Chapter 13).
Can I define constructors for this type?	Yes, but the default constructor is reserved (i.e., your custom constructors must all have arguments).	But, of course!
When do variables of this type die?	When they fall out of the defining scope.	When the object is garbage collected.

Task 4

- Define **class Car** with fields *name*, *color*, *price* and const field *CompanyName* Create two *constructors* - default and with parameters. Create a *property* to access the color field. Define methods: **Input ()** - to enter car data from the console, **Print ()** - to output the machine data to the console **ChangePrice (double x)** - to change the price by x%
- Enter data about 3 cars.
- Decrease their price by 10%, display info about the car.
- Enter a new color and paint the car with the color white in the specified color
- Overload the operator == for the class Car (cars - equal if the name and price are equal)
- Overload the method ToString () in the class Car, which returns a line with data about the car

Homework 4

1) Create class Person.
   Class Person should consists of
   a) two private fields: name and birthYear (the birthday year).As a type for this field you may use DataTime type.)
   b) two properties for access to these fields (only get)
   c) default constructor and constructor with 2 parameters
   d) methods: - Age() - to calculate the age of person
   -Input() - to input information about person
   -ChangeName() - to change the name of person
   -ToString()
   -Output() - to output information about person (call ToString())
   - operator== (equal by name)
   In the method Main() create 6 objects of Person type and input information about them. Then calculate and write to console the name and Age of each person;
Change the name of persons, which Age is less then 16, to "Very Young".
Output information about all persons.
Find and output information about Persons with the same names (use ==)

2. Learn next C# topics:
   a) Class, objects, fields, properties, constructors, methods
   b) Interfaces
   c) Collections C#