Classes and Structures in C#

Table of Contents

- Defining Classes
- Constructors
- Fields, Constants and Properties
- Static Members
- Structures
- Enumerations
- Interfaces

Școala
informală
de IT

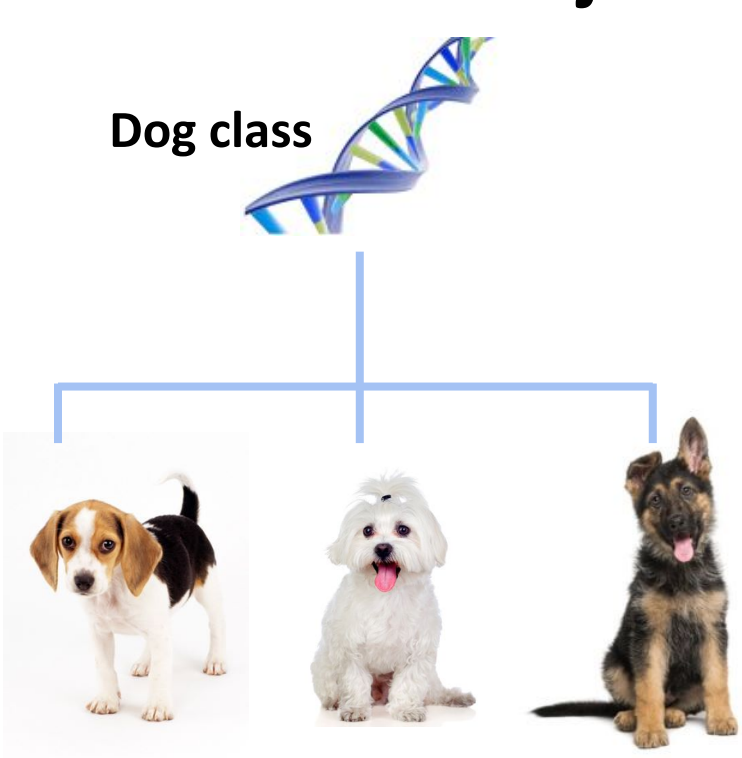
Defining Classes

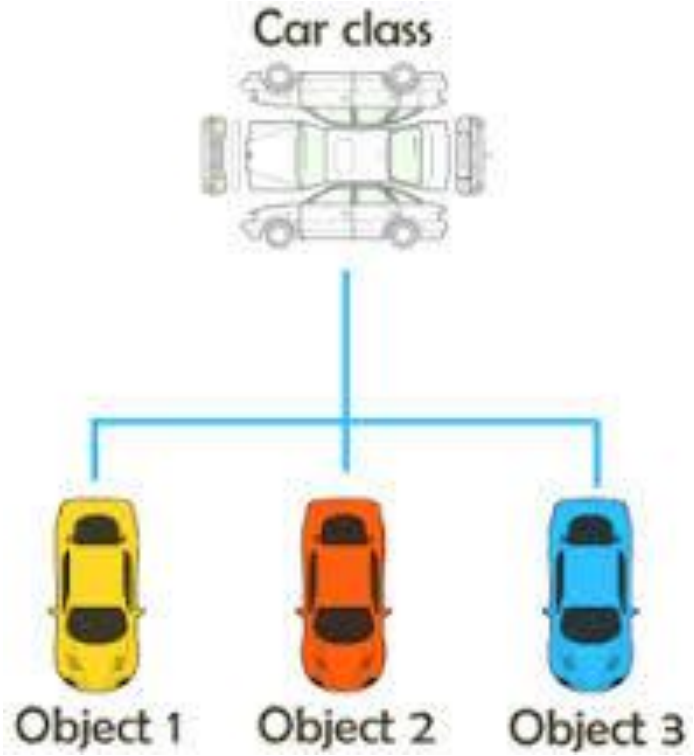
Classes in C#

- Classes model real-world objects and define
  - Attributes (state, properties, fields)
  - Behavior (methods, operations)
- Classes describe structure of objects
  - Objects describe particular instance of a class
- Properties hold information about the modeled object relevant to the problem
- Operations implement object behavior

Classes in C#

- **Classes ≠ objects!**





Școala informală de IT

Classes in C#

- Classes in C# could have following members:
  - Fields, constants, methods, properties, indexers, events, operators, constructors, destructors
  - Inner types (inner classes, structures, interfaces, delegates, …)
- Members can have access modifiers (scope)
  - `public`, `private`, `protected`, `internal`
- Members can be
  - `static` (common) or specific for a given object

Școala informală de IT

Defining Class Dog

Begin of class definition

```
public class Dog
{
    private string name;
    private string breed;

    public Dog()
    {
        this.name = "Spot";
        this.breed = "Beagle";
    }

    public Dog(string name, string breed)
    {
        this.name = name;
        this.breed = breed;
    }
```

Fields

Parameterless Constructor

Constructor

(continued)

Defining Class Dog

```
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public string Breed
    {
        get { return breed; }
        set { breed = value; }
    }

    public void Bark()
    {
        Console.WriteLine("{0} barks...", name);
    }
}
```

Property

Property

Method

Using Class Dog

```
static void Main()
{
    // Using the parameterless constructor
    Dog beast = new Dog();

    // Using properties to set name and breed
    beast.Name = "Lucky";
    beast.Breed = "Bichon";

    // Using the constructor with params to set name and breed
    Dog puppy = new Dog("Rex", "Beagle");
}
```

New Instance

New Instance

Școala
informală
de IT

Constructors

Defining and Using Class Constructors

Școala informală de IT

What is Constructor?

- Constructors are special methods
  - Invoked when creating a new instance of an object
  - Used to initialize the fields of the instance
  - If no constructor is defined a default parameterless constructor is provided
- Constructors has the same name as the class
  - Have no return type
  - Can have parameters
  - Can be `private`, `protected`, `internal`, `public`

Defining Constructors

- Class Dog with default parameterless constructor
  - (only) If no constructor is present the compiler provides a parameterless default constructor

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

    // Parameterless default constructor
    // public Dog()
    // {
    // }
}
```

Defining Constructors

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

    // Parameterless constructor
    public Dog()
    {
        name = "[no name]";
        age = 0;
    }

    // Constructor with parameters
    public Dog(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

As rule, constructors should initialize all own class fields.

Școala informală de IT

Constructors Initialization

- **Pay attention when using inline initialization!**

```
public class Dog
{
   private int age = 9;                // Inline initialization
   private string name = "[no name]"; // Inline initialization
   private string species;

   public Dog()                        // Default constructor
   {
      species = "Canis Canis";
   }
   public Dog(int age, string name)    // constructor with params
   {
      this.age = age;                  // Invoked after the inline
      this.name = name;                // initialization!
   }
}
```

Școala informală de IT

Constructors Initialization

- **Pay attention when using inline initialization!**

```
public class Dog
{
   private int age;                // Inline initialization
   private string name;            // Inline initialization
   private string species;

   public Dog()                          // Default constructor
   {
      species = "Canis Canis";
      this.age = 9;                      // Invoked after the inline
      this.name = "[no name]";           // initialization!
   }
   public Dog(int age, string name)   // constructor with params
   {
      species = "Canis Canis";
      this.age = age;                    // Invoked after the inline
      this.name = name;                 // initialization!
   }
}
```

Școala informală de IT

Chaining Constructors Calls

- **Reusing constructors**

```
public class Dog
{
    private int age;
    private string name;
    private string species;

    public Dog() : this(9, "[no name]") // Reuse constructor
    {
    }

    public Dog(int age, string name)
    {
        species = "Canis Canis";
        this.age = age;
        this.name = name;
    }
}
```

Școala
informală
de IT

Workshop Constructors

Școala
informală
de IT

Fields, Constants and Properties

Fields

- Fields contain data for the class instance (object state)
- Can be arbitrary type (value or reference type)
- Have given visibility (using access modifiers)
- Can be declared with a specific value

```
class Dog
{
    public string name;
    public string owner;
    public int age = 3;
    public string breed;
    public Trick[] tricks;
}
```

Constants

- Constant fields are defined like fields, but:
  - Defined with `const`
  - Must be initialized at compile-time
  - Their value can not be changed at runtime

```
public class MathConstants
{
    public const string PI_SYMBOL = "π";
    public const double PI = 3.14159265358979323385;
    public const double E = 2.71828182845904523354;
    public const double LN10 = 2.30258509299405;
}
```

Read-Only Fields

- Initialized at the definition or in the constructor
  - Can not be further modified
- Defined with the keyword `readonly`
- Represent runtime constants

```
public class Dog
{
    public readonly DateTime dateOfBirth;

    public Dog()
    {
      dateOfBirth = DateTime.Now; // cannot be further modified!
    }
}
```

The Role of Properties

- Expose object data (fields) to the outside world
- Control how sensitive data is manipulated (e.g. by validation)
- Properties can be:
  - Read-only
  - Write-only
  - Read-write

Școala informală de IT

Defining Properties in C#

- Properties should have:
  - Access modifier (`public`, `protected`, etc.)
  - Return type
  - Unique name
  - getter and / or setter
  - Can contain code processing data in specific way (e.g. validating, formatting)

Properties Example

```
public class Dog
{
    private string name;        // prevent access to the field
    private int age;

    public string Name          // the property is "embedding" the field
    {
        get { return name; }
        set { name = value; }
    }


    public int Age
    {
        get { return age; }
        set { age = (value >= 0) ? value : 0; } // validating the value
    }
}
```

Dynamic Properties

- **Properties are not obligatory bound to a class field – can be calculated dynamically:**

```
public class Dog
{
    private string name;       // prevent access to the field
    private string owner;

    public string NameAndOwner // the property is "embedding" the field
    {
        get { return name + " " + owner; }
    }
}
```

Școala informală de IT

Automatic Properties

- Properties could be defined without an underlying field behind them
  - Behind field is automatically created by the C# compiler

```
class Dog
{
    public int Age { get; set; }
    public string Name { get; set; }
    public string Breed { get; set; }
}
```

Automatic Properties

- Create object with initial values for properties

```
class Dog
{
    public int Age { get; set; }
    public string Name { get; set; }
    public string Breed { get; set; }

}
```

```
Dog puppy = new Dog() {
    Age = 3,
    Name = "Rex",
    Breed = "Beagle"
};
```

Școala informală de IT

Readonly and Writeonly Properties

- Private or missing set accessor turns the property readonly
- Private or missing get accessor turns the property writeonly

```
class Dog
{
    public string Name { get; private set; }     // readonly
    public string Owner { get; }                 // readonly

    public string Species { private get; set; }  // writeonly
    public string Breed { set; }                 // writeonly
}
```

Școala
informală
de IT

Workshop Attributes

Școala
informală
de IT

Static Members

Școala informală de IT

Static Members

- Static (common) members are associated with a type rather than with an instance
  - Defined with the modifier `static`
  - Accessed by class name
- Static can be used for
  - Fields
  - Properties
  - Methods
  - Events
  - Constructors

Static vs. Non-Static

- Static:
  - Associated with a type, not with an instance
  - Initialized just before the type is used for the first time
  - `const` field behaves like `static`
- Non-Static:
  - The opposite, associated with an instance
  - Initialized when the constructor is called

Școala informală de IT

Static Members – Example

```
public class Dog
{
   private string name;              // non-static field
   private static int counter;       // declare static field

   private static Dog() {            // static constructor
      counter = 0;                   // init static field
      name = "";                     // cannot access non-static field here
   }

   public Dog() {
      counter++;                     // use static field
   }

   public static int GetTotal()      // static method
   {
      return counter;                // use static field
   }
}
```

Școala informală de IT

Static Members – Example

```
// The Main() method is always static
static void Main()
{
   // no need to instantiate the class
   Console.WriteLine(Dog.GetTotal()); // returns 0

   Dog puppy = new Dog();
   Dog beast = new Dog();

   Console.WriteLine(Dog.GetTotal()); // returns 2
}
```

Școala
informală
de IT

Workshop Static

Școala
informală
de IT

Structures

Structures

- Structures represent a combination of fields with data
  - Look like the classes, but they are value types
  - Structs content is stored on the stack (when possible)
  - Transmitted by value (a perfect clone is created)
  - Destroyed along with the stack which contains them, when go out of scope
- However classes are reference type and are placed always in the dynamic memory (heap)
  - Class creation and destruction is slower

Școala informală de IT

Structures Example

```
struct Point
{
    public int x, y;
}

struct Color
{
    public byte red;
    public byte green;
    public byte blue;
}

struct Square
{
    public Point location;
    public int size;
    public Color borderColor;
    public Color surfaceColor;
}
```

When to Use Structures?

Class	Structure
A Class is a reference type	A Structure is a value type
By default, the members of a Class are private	By default, the members of a Structure are public
Class supports Inheritance	Structure does not support Inheritance
Class can contain constructor/destructor	Structure does not require Constructor/Destructor
Variables of a Class can be assigned as null	Structure members can not have null values

Școala informală de IT

When to Use Structures?

- Use structures
  - To make your type behave as a primitive type or if you need to pass variables by value
  - If you create many instances and after that you free them – e.g. in a "for" cycle
  - If you have fewer fields (not recommended for more than 16 bytes in total)
  - When methods don't have complex logic inside
  - To minimize the number of method params

Școala informală de IT

When to Use Classes?

- Use classes
  - If you need to pass variables by reference
  - When you have complex scenarios and logic
  - If you prefer to work with reference types
  - When you often transmit your instances as method parameters

Școala
informală
de IT

Workshop Structs

Școala
informală
de IT

Enumerations

Enumerations

- Enumeration is a structure consisting only on constants.
- Enumeration can take values only from the constants listed in the type.
- Each constant in enumeration is being associated with a certain integer (by default is the zero-based index in the list).
- Each constant in enumeration is a textual representation (alias) of an integer.
- Use enumerations instead of set of constants.

Școala informală de IT

Enumerations Example

```
enum Breed
{
  Beagle, Bichon, GermanShepard, ShiTzu, Boxer, Bulldog
}
```

```
enum Day
{
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
}
```

```
enum Level
{
  Beginner = 100,
  Intermediate = 200,
  Advanced = 300
}
```

Școala informală de IT

Enumerations Example

```
static void Main()
{
  public Breed breed = Breed.Beagle;
  public Breed otherBreed = (Breed)1;  // convert to enum by index
}
```

Școala
informală
de IT

Workshop Enums