Defining Classes

Classes, Fields, Constructors, Methods, Properties

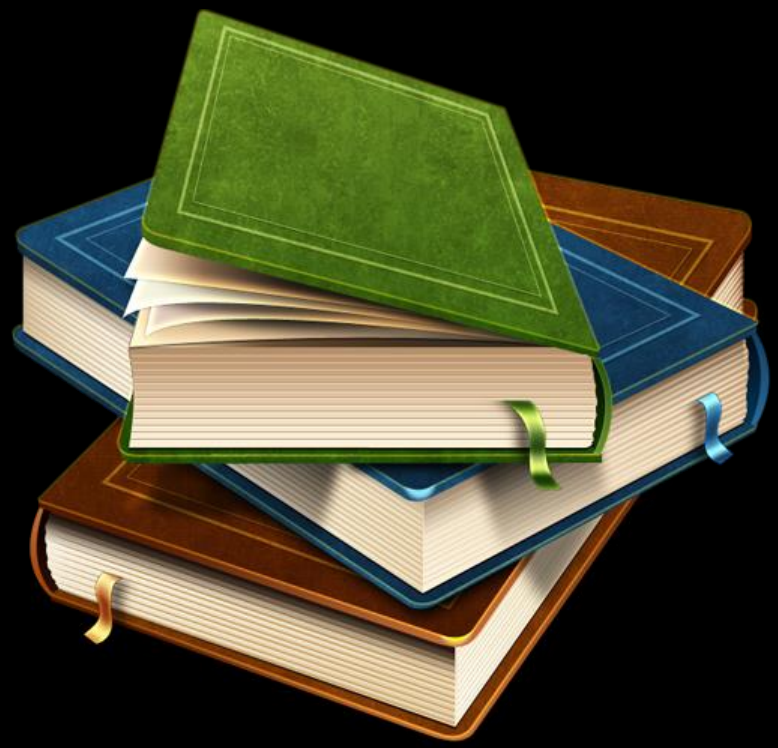
SoftUni Team

Technical Trainers

Software University

http://softuni.bg

Table of Contents



SOFTWARE UNIVERSITY FOUNDATION

Defining Simple Classes

Classes in OOP

- Classes model real-world objects and define:
  - Attributes (state, properties, fields)
  - Behavior (methods, operations)
- Classes describe the 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 in C# can have 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 (per instance)

Simple Class Definition


```
public class Cat : Animal
{
    private string name;
    private string owner;

    public Cat(string name, string owner)
    {
        this.Name = name;
        this.Owner = owner;
    }

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

Beginning of class definition

Inherited (base) class

Fields

Constructor

Property

Simple Class Definition (2)

```
    public string Owner
    {
        get { return this.owner; }
        set { this.owner = value; }
    }

    public void SayMiau()
    {
        Console.WriteLine("Miauuuuuuuu!");
    }
}
```

Method

End of class definition

Class Definition and Members

- Class definition consists of:
  - Class declaration
  - Inherited class and implemented interfaces
  - Fields (static or not)
  - Constructors (static or not)
  - Properties (static or not)
  - Methods (static or not)
  - Events, inner types, etc.

SOFTWARE UNIVERSITY FOUNDATION

Fields

Defining and Using Data Fields

Fields

- Fields are data members defined inside a class
  - Fields hold the internal object state
  - Can be **`static`** or per instance
  - Can be **`private`** / **`public`** / **`protected`** / …

```
class Dog
{
    private string name;
    private string breed;
    private int age;
    protected Color color;
}
```

Field declarations

Constant Fields

- Constant fields are of two types:
  - Compile-time constants – **`const`**
    - Replaced by their value during the compilation
  - Runtime constants – **`readonly`**
    - Assigned once only at object creation

```
class Math
{
    public const float PI = 3.14159;
    public readonly Color = Color.FromRGBA(25, 33, 74, 128);
}
```

Constant Fields – Example

SOFTWARE UNIVERSITY FOUNDATION

```
public class Constants
{
  public const double PI = 3.14159265358979323385;
  public readonly double Size;

  public Constants(int size)
  {
    this.Size = size; // Cannot be further modified!
  }

  static void Main()
  {
    Console.WriteLine(Constants.PI);
    Constants c = new Constants(5);
    Console.WriteLine(c.Size);

    c.Size = 10; // Compilation error: readonly field
    Console.WriteLine(Constants.Size); // Compilation error: non-static field
  }
}
```

SOFTWARE UNIVERSITY FOUNDATION

```
struct group_info init_groups = { .usage = ATOMIC_INIT(2) };
struct group_info *groups_alloc(int gidsetsize){
    struct group_info *group_info;
    int nblocks;
    int i;

    nblocks = (gidsetsize + NGROUPS_PER_BLOCK - 1) / NGROUPS_PER_BLOCK;
    /* Make sure we always allocate at least one indirect block pointer */
    nblocks = nblocks ? : 1;
    group_info = kmalloc(sizeof(*group_info) + nblocks*sizeof(gid_t *), GFP_USER);
    if
    gro
    gro
    ato

    if (gidsetsize <= NGROUPS_SMALL)
        group_info->blocks[0] = group_info->small_block;
    else {
        for (i = 0; i < nblocks; i++) {
            gid_t *b;
            b = (void *)__get_free_page(GFP_USER);
            if (!b)
                goto out_undo_partial_alloc;
            group_info->blocks[i] = b;
```

ACCESS DENIED

Access Modifiers

public, private, protected, internal

Access Modifiers

- Class members can have access modifiers
  - Used to restrict the access from the other classes
  - Supports the OOP principle "encapsulation"
- Class members can be:
  - **`public`** – accessible from any class
  - **`protected`** – accessible from the class itself and all its descendent classes
  - **`private`** – accessible from the class itself only
  - **`internal`** (default) – accessible from the current assembly, i.e. the current Visual Studio project

The "this" Keyword

- The keyword `this` points to the current instance of the class
- Example:

```
class Dog
{
    private string name;

    public void PrintName()
    {
        Console.WriteLine(this.name);
        // The same like Console.WriteLine(name);
    }
}
```

Defining Simple Classes

Example

Task: Define a Class "Dog"

- Our task is to define a simple class "*dog*"
  - Represents information about a dog
  - The dog should have name and breed
    - Optional fields (could be **`null`**)
  - The class allows to view and modify the name and the breed at any time
  - The dog should be able to bark

Defining Class Dog – Example

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

    public Dog()
    {
    }

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

(the example continues)

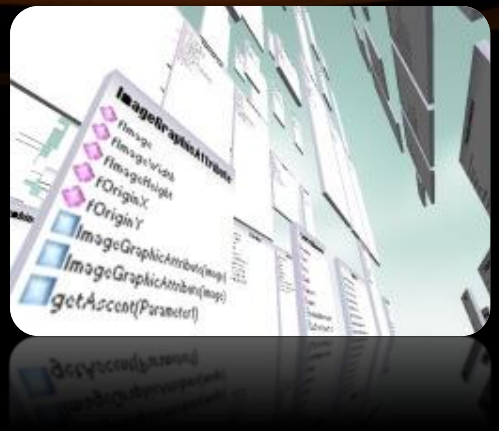
Defining Class Dog – Example (2)

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

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

    public void SayBau()
    {
        Console.WriteLine("{0} said: Bauuuuuu!",
            this.name ?? "[unnamed dog]");
    }
}
```

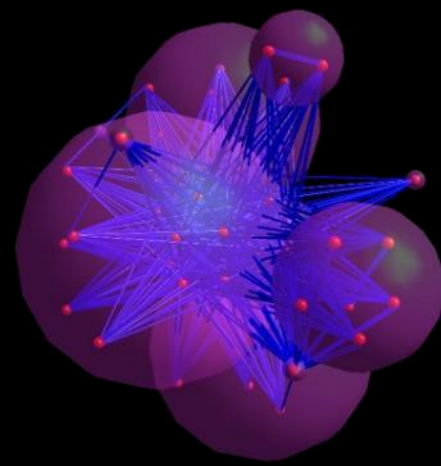
SOFTWARE UNIVERSITY FOUNDATION

Using Classes and Objects

How to Use Classes (Non-Static)?



1. Create an instance
   - Initialize its properties / fields
2. Manipulate the instance
   - Read / modify its properties
   - Invoke methods
   - Handle events
3. Release the occupied resources
   - Performed automatically in most cases



Dog Meeting – Example

```
static void Main()
{
    Console.Write("Enter first dog's name: ");
    string dogName = Console.ReadLine();
    Console.Write("Enter first dog's breed: ");
    string dogBreed = Console.ReadLine();

    // Use the Dog constructor to assign name and breed
    Dog firstDog = new Dog(dogName, dogBreed);

    // Use Dog's parameterless constructor
    Dog secondDog = new Dog();

    // Use properties to assign name and breed
    Console.Write("Enter second dog's name: ");
    secondDog.Name = Console.ReadLine();
    Console.Write("Enter second dog's breed: ");
    secondDog.Breed = Console.ReadLine();    (the example continues)
```

Dog Meeting – Example (2)

```
    // Create a Dog with no name and breed
    Dog thirdDog = new Dog();

    // Save the dogs in an array
    Dog[] dogs = new Dog[] { firstDog, secondDog, thirdDog };

    // Ask each of the dogs to bark
    foreach(Dog dog in dogs)
    {
        dog.SayBau();
    }
}
```

Dog Meeting

Live Demo

SOFTWARE UNIVERSITY FOUNDATION

Constructors

Defining and Using Class Constructors

What is Constructor?

- Constructors are special methods
  - Invoked at the time of creating a new instance of an object
  - Used to initialize the fields of the instance
- Constructors has the same name as the class
  - Have no return type
  - Can have parameters
  - Can be **`private`**, **`protected`**, **`internal`**, **`public`**

Defining Constructors

- Class **Point** with parameterless constructor:

```
public class Point
{
    private int xCoord;
    private int yCoord;

    // Simple parameterless constructor
    public Point()
    {
        this.xCoord = 0;
        this.yCoord = 0;
    }
    // More code …
}
```

Defining Constructors (2)

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

  // Parameterless constructor
  public Person()
  {
    this.name = null;
    this.age = 0;
  }

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

  // More code …

}
```

As a rule constructors should initialize all class fields

Constructors and Initialization

- Pay attention when using inline initialization!

```
public class AlarmClock
{
   private int hours = 9; // Inline initialization
   private int minutes = 0; // Inline initialization

   // Parameterless constructor (intentionally left empty)
   public AlarmClock()
   { }

   // Constructor with parameters
   public AlarmClock(int hours, int minutes)
   {
      this.hours = hours;      // Invoked after the inline
      this.minutes = minutes;  // initialization!
   }

   // More code …
}
```

Chaining Constructors Calls

- Reusing the constructors (chaining)

```
public class Point
{
    private int xCoord;
    private int yCoord;

    public Point() : this(0, 0) // Reuse the constructor
    {
    }

    public Point(int xCoord, int yCoord)
    {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }
    // More code …
}
```

SOFTWARE UNIVERSITY FOUNDATION

Constructors

Live Demo

SOFTWARE UNIVERSITY FOUNDATION

```
public double CalcDistance(Point p)
{
    return Math.Sqrt(
      (p.xCoord - this.xCoord) * (p.xCoord - this.xCoord) +
      (p.yCoord - this.yCoord) * (p.yCoord - this.yCoord));
}
```

Methods

Defining and Invoking Methods

Methods

- Methods execute some action (some code / some algorithm)
  - Could be **`static`** / per instance
  - Could be **`public`** / **`private`** / **`protected`** / …

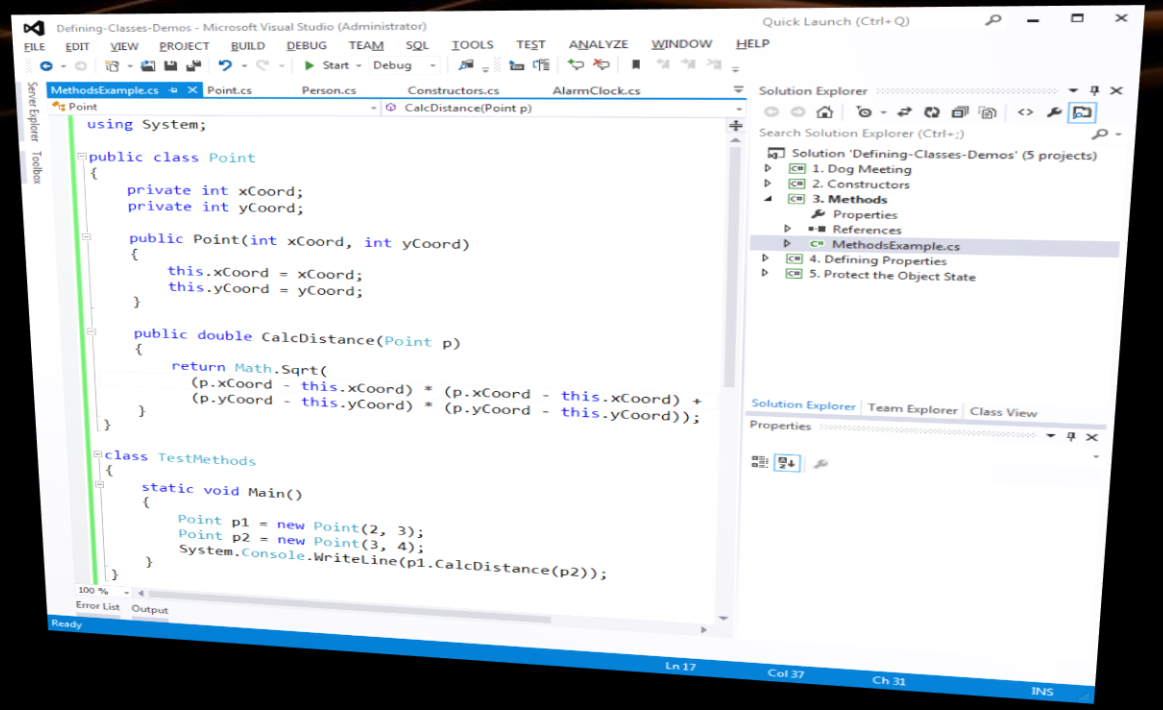
```
public class Point
{
  private int xCoord;
  private int yCoord;

  public double CalcDistance(Point p)
  {
    return Math.Sqrt(
      (p.xCoord - this.xCoord) * (p.xCoord - this.xCoord) +
      (p.yCoord - this.yCoord) * (p.yCoord - this.yCoord));
  }
}
```

Using Methods

- Invoking instance methods is done through the object (through the class instance):

```
class TestMethods
{
  static void Main()
  {
    Point p1 = new Point(2, 3);
    Point p2 = new Point(3, 4);
    System.Console.WriteLine(p1.CalcDistance(p2));
  }
}
```

SOFTWARE UNIVERSITY FOUNDATION



Methods

Live Demo

SOFTWARE UNIVERSITY FOUNDATION

Properties

Defining and Using Properties

Why We Need Properties?

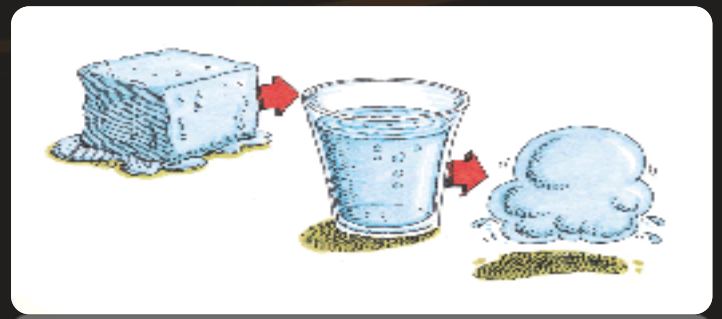
- Properties expose object's data to the world
  - Control how the data is accessed / manipulated
    - Ensure the internal object state is correct
    - E.g. price should always be kept positive
- Properties can be:
  - Read-only
  - Write-only
  - Read and write

Defining Properties

- Properties work as a pair of get / set methods
  - Getter and setter
- Properties should have:
  - Access modifier (`public`, `protected`, etc.)
  - Return type
  - Unique name
  - Get and / or Set part
  - Can process data in specific way, e.g. apply validation

Defining Properties – Example

```
public class Point
{
    private int xCoord;
    private int yCoord;

    public int XCoord
    {
        get { return this.xCoord; }
        set { this.xCoord = value; }
    }

    public int YCoord
    {
        get { return this.yCoord; }
        set { this.yCoord = value; }
    }

    // More code …
}
```



Dynamic Properties

- Properties are not always bound to a class field
  - Can be dynamically calculated:

```
public class Rectangle
{
    private double width;
    private double height;

    public double Area
    {
        get
        {
            return width * height;
        }
    }
}
```

Automatic Properties

- Properties could be defined without an underlying field
  - They are automatically created by the compiler (auto properties)

```
class UserProfile
{
    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
…
UserProfile profile = new UserProfile() {
    FirstName = "Steve",
    LastName = "Balmer",
    UserId = 91112
};
```

Properties

Live Demo

SOFTWARE UNIVERSITY FOUNDATION

Keeping the Object State Correct

Keep the Object State Correct

- Constructors and properties can keep the object's state correct
  - This is known as **encapsulation** in OOP
  - Can force validation when creating / modifying the object's internal state
  - Constructors define which properties are mandatory and which are optional
  - Property setters should validate the new value before saving it in the object field
  - Invalid values should cause an exception

Keep the Object State – Example

```
public class Person
{
    private string name;

    public Person(string name)
    {
        this.Name = name;
    }

    public string Name
    {
        get { return this.name; }
        set
        {
            if (String.IsNullOrEmpty(value))
                throw new ArgumentException("Invalid name!");
            this.name = value;
        }
    }
}
```

We have only one constructor, so we cannot create person without specifying a name

Incorrect name cannot be assigned

SOFTWARE UNIVERSITY FOUNDATION

Keeping the Object State Correct

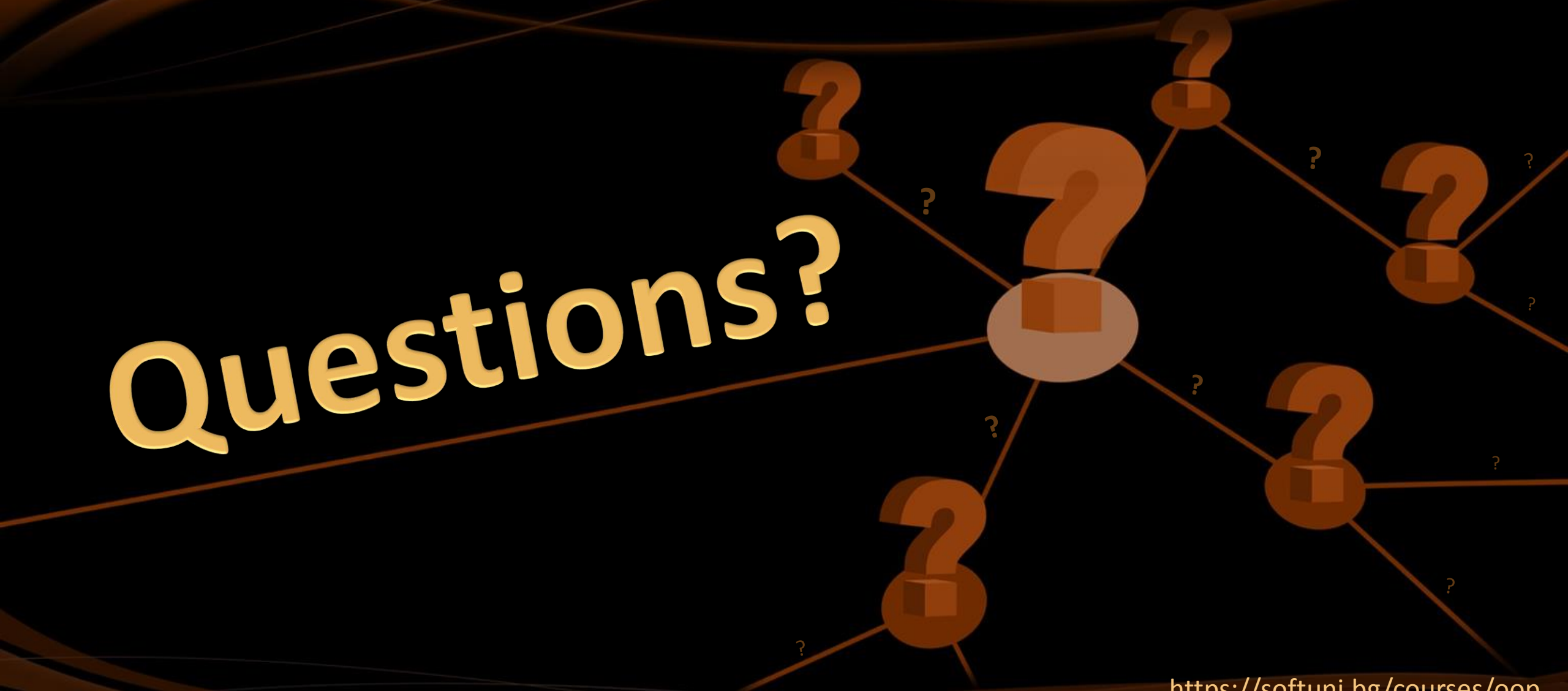
Live Demo

Summary

- Classes define specific structure for objects
  - Objects are particular instances of a class
- Classes define fields, methods, constructors, properties and other members
  - Access modifiers limit the access to class members
- Constructors are invoked when creating new class instances and initialize the object's internal state
- Properties expose the class data in safe, controlled way

OOP – Defining Classes

SOFTWARE UNIVERSITY FOUNDATION



https://softuni.bg/courses/oop

License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

- Attribution: this work may contain portions from
  - "Fundamentals of Computer Programming with C#" book by Svetlin Nakov & Co. under CC-BY-SA license
  - "OOP" course by Telerik Academy under CC-BY-NC-SA license

SoftUni Diamond Partners

SOFTWARE UNIVERSITY FOUNDATION

Free Trainings @ Software University

- Software University Foundation – softuni.org
- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University @ YouTube
  - youtube.com/SoftwareUniversity
- Software University Forums – forum.softuni.bg