Defining Classes

Classes, Fields, Constructors, Methods, Properties

Svetlin Nakov

Telerik Corporation

www.telerik.com

Table of Contents

Defining Simple Classes

Classes in OOP

- 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 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

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

Begin of class definition

Inherited (base) class

Fields

Constructor

Property

telerik

Simple Class Definition (2)

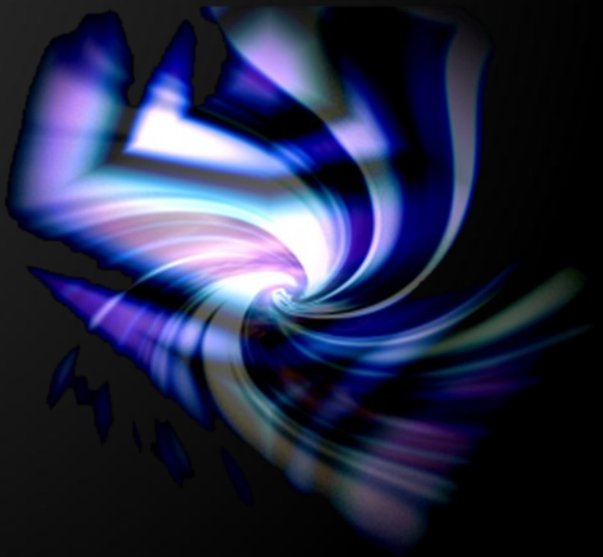
```
    public string Owner
    {
        get { return owner;}
        set { owner = value; }
    }

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

Method

End of class definition

Class Definition and Members

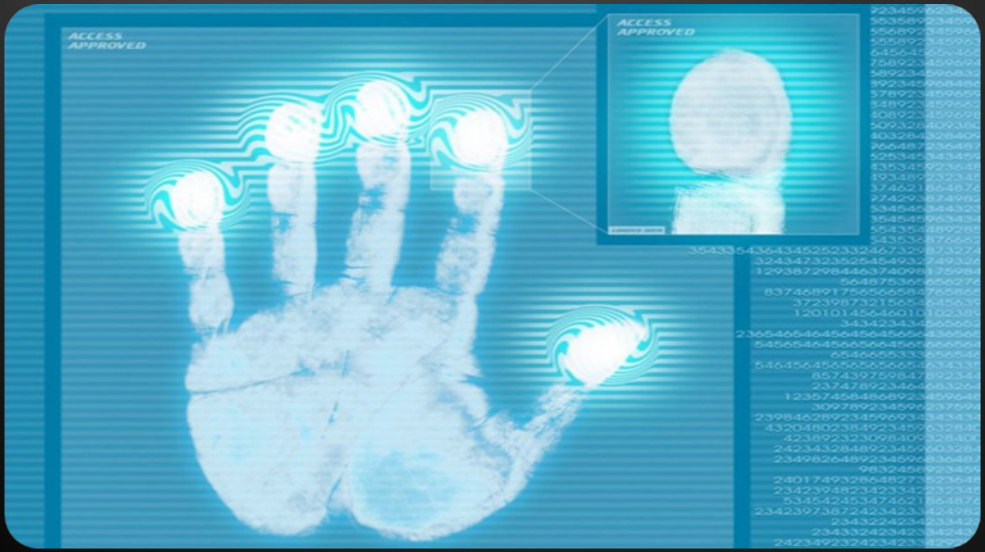
- Class definition consists of:
  - Class declaration
  - Inherited class or implemented interfaces
  - Fields (static or not)
  - Constructors (static or not)
  - Properties (static or not)
  - Methods (static or not)
  - Events, inner types, etc.



telerik

Access Modifiers

Public, Private, Protected, Internal



Access Modifiers

- Class members can have access modifiers
  - Used to restrict the classes able to access them
  - 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` – accessible from the current assembly (used by default)

Defining Simple Classes

Example

Task: Define Class Dog

- Our task is to define a simple class that represents information about a dog
  - The dog should have name and breed
  - If there is no name or breed assigned to the dog, it should be named "Balkan" and its breed should be "Street excellent"
  - It should be able to view and change the name and the breed of the dog
  - The dog should be able to bark

Defining Class Dog – Example

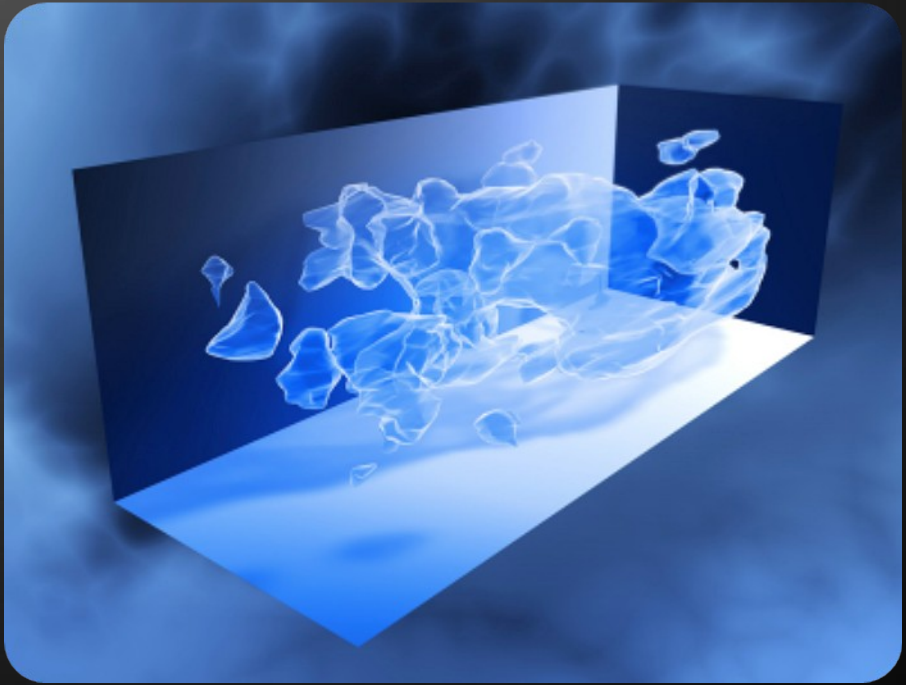
```
public class Dog
{
    private string name;
    private string breed;

    public Dog()
    {
        this.name = "Balkan";
        this.breed = "Street excellent";
    }

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

(example continues)

Defining Class Dog – Example (2)

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

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

    public void SayBau()
    {
        Console.WriteLine("{0} said: Bauuuuuu!", name);
    }
}
```

Using Classes and Objects



telerik

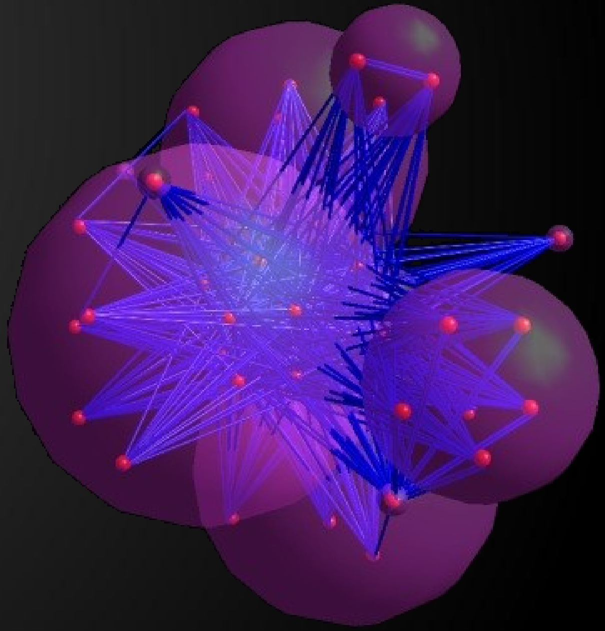
Using Classes

- How to use classes?
  - Create a new instance
  - Access the properties of the class
  - Invoke methods
  - Handle events
- How to define classes?
  - Create new class and define its members
  - Create new class using some other as base class

telerik

How to Use Classes (Non-static)?

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



Task: Dog Meeting

- Our task is as follows:
  - Create 3 dogs
    - First should be named “Sharo”, second – “Rex” and the last – left without name
  - Add all dogs in an array
  - Iterate through the array elements and ask each dog to bark
  - Note:
    - Use the Dog class from the previous example!

Dog Meeting – Example

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

   // Using the Dog constructor to set name and breed
   Dog firstDog = new Dog(dogName, dogBreed);
   Dog secondDog = new Dog();
   Console.WriteLine("Enter second dog's name: ");
   dogName = Console.ReadLine();
   Console.WriteLine("Enter second dog's breed: ");
   dogBreed = Console.ReadLine();

   // Using properties to set name and breed
   secondDog.Name = dogName;
   secondDog.Breed = dogBreed;
}
```

Dog Meeting

Live Demo

Constructors

Defining and Using Class Constructors

What is Constructor?

- Constructors are special methods
  - Invoked when 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 default constructor
    public Point()
    {
        xCoord = 0;
        yCoord = 0;
    }

    // More code ...
}
```

Defining Constructors (2)

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

    // Default constructor
    public Person()
    {
        name = null;
        age = 0;
    }

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

    // More code ...
}
```

As rule constructors should initialize all own class fields.

Constructors and Initialization

- Pay attention when using inline initialization!

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

   // Default constructor
   public ClockAlarm()
   { }

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

   // More code ...
}
```

Chaining Constructors Calls

- Reusing constructors

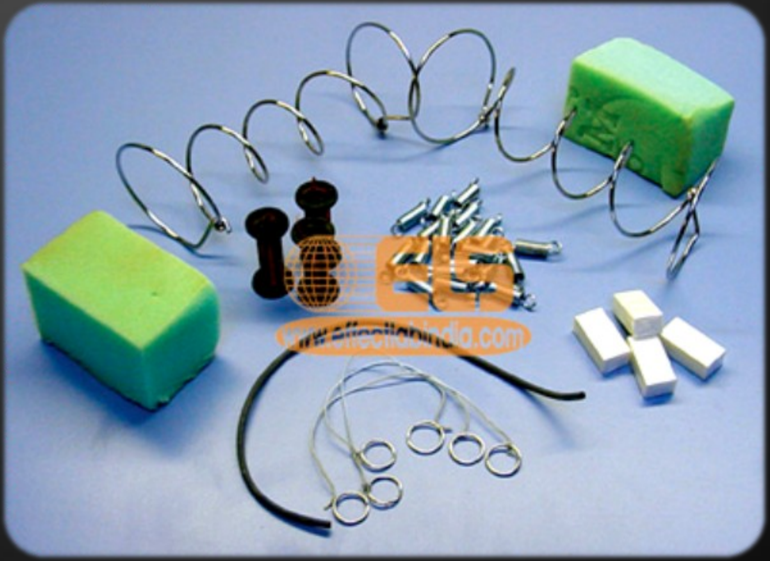
```
public class Point
{
    private int xCoord;
    private int yCoord;

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

    public Point(int xCoord, int yCoord)
    {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }

    // More code ...
}
```

Constructors

Live Demo

Properties

Defining and Using Properties



The Role of Properties

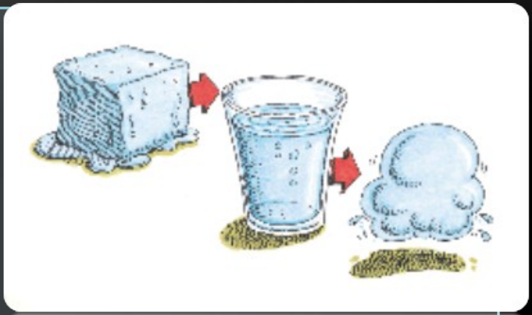
- Expose object's data to the outside world
- Control how the data is manipulated
- Properties can be:
  - Read-only
  - Write-only
  - Read and write
- Give good level of abstraction
- Make writing code easier

Defining Properties

- Properties should have:
  - Access modifier (`public`, `protected`, etc.)
  - Return type
  - Unique name
  - Get and / or Set part
  - Can contain code processing data in specific way

Defining Properties – Example

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

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

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

    // More code ...
}
```



Dynamic Properties

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

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

    // More code ...

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

Automatic Properties

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

```
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

Static Members

Static vs. Instance Members

Static Members

- Static members are associated with a type rather than with an instance
  - Defined with the modifier `static`
- Static can be used for
  - Fields
  - Properties
  - Methods
  - Events
  - Constructors

Static vs. Non-Static

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

Static Members – Example

```
static class SqrtPrecalculated
{
   public const int MAX_VALUE = 10000;

   // Static field
   private static int[] sqrtValues;

   // Static constructor
   static SqrtPrecalculated()
   {
      sqrtValues = new int[MAX_VALUE + 1];
      for (int i = 0; i < sqrtValues.Length; i++)
      {
         sqrtValues[i] = (int)Math.Sqrt(i);
      }
   }
```

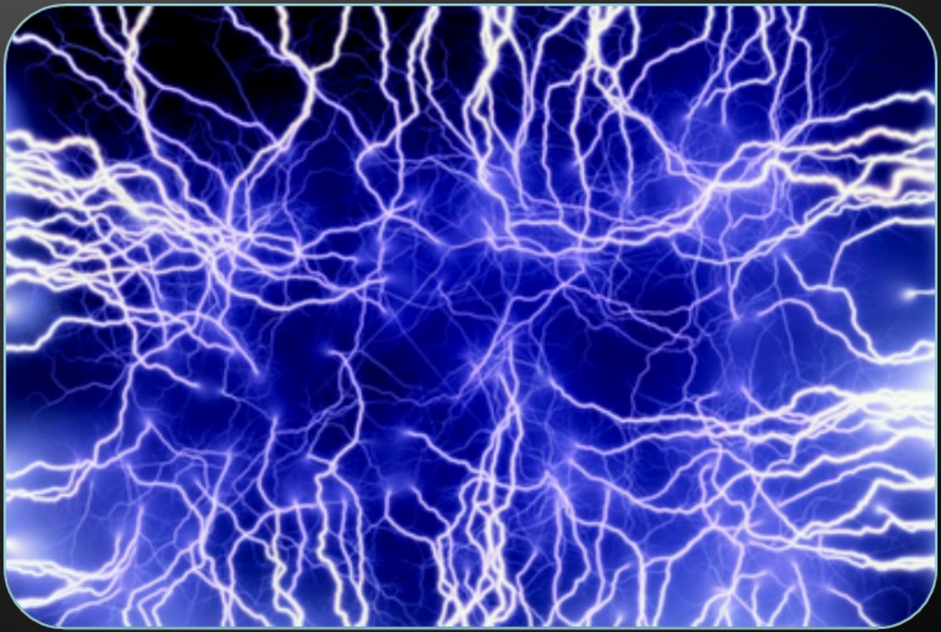
(example continues)

Static Members – Example (2)

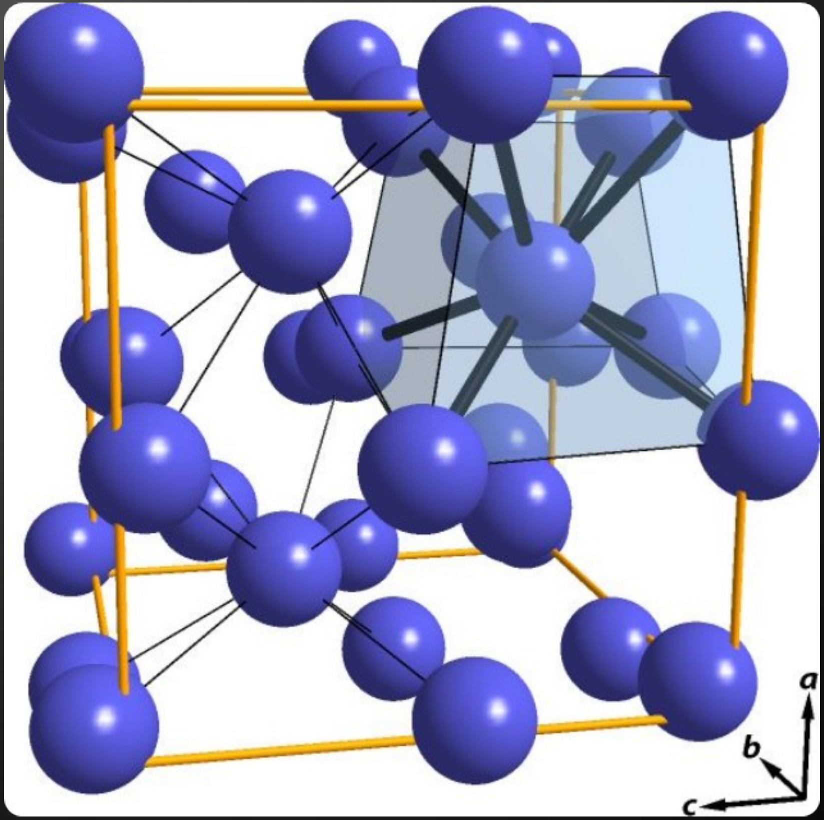
```
    // Static method
    public static int GetSqrt(int value)
    {
        return sqrtValues[value];
    }
}

class SqrtTest
{
    static void Main()
    {
      Console.WriteLine(SqrtPrecalculated.GetSqrt(254));
      // Result: 15
    }
}
```

Static Members

Live Demo



C# Structures



C# Structures

- What is a structure in C#
  - A primitive data type
    - Classes are reference types
    - Examples: `int`, `double`, `DateTime`
  - Represented by the key word `struct`
  - Structures, like classes, have Properties, Methods, Fields, Constructors
  - Always have a parameterless constructor
    - This constructor cannot be removed
  - Mostly used to store data

C# Structures – Example

```
struct Point
{
   public int X { get; set; }
   public int Y { get; set; }
}

struct Color
{
   public byte RedValue { get; set; }
   public byte GreenValue { get; set; }
   public byte BlueValue { get; set; }
}
```

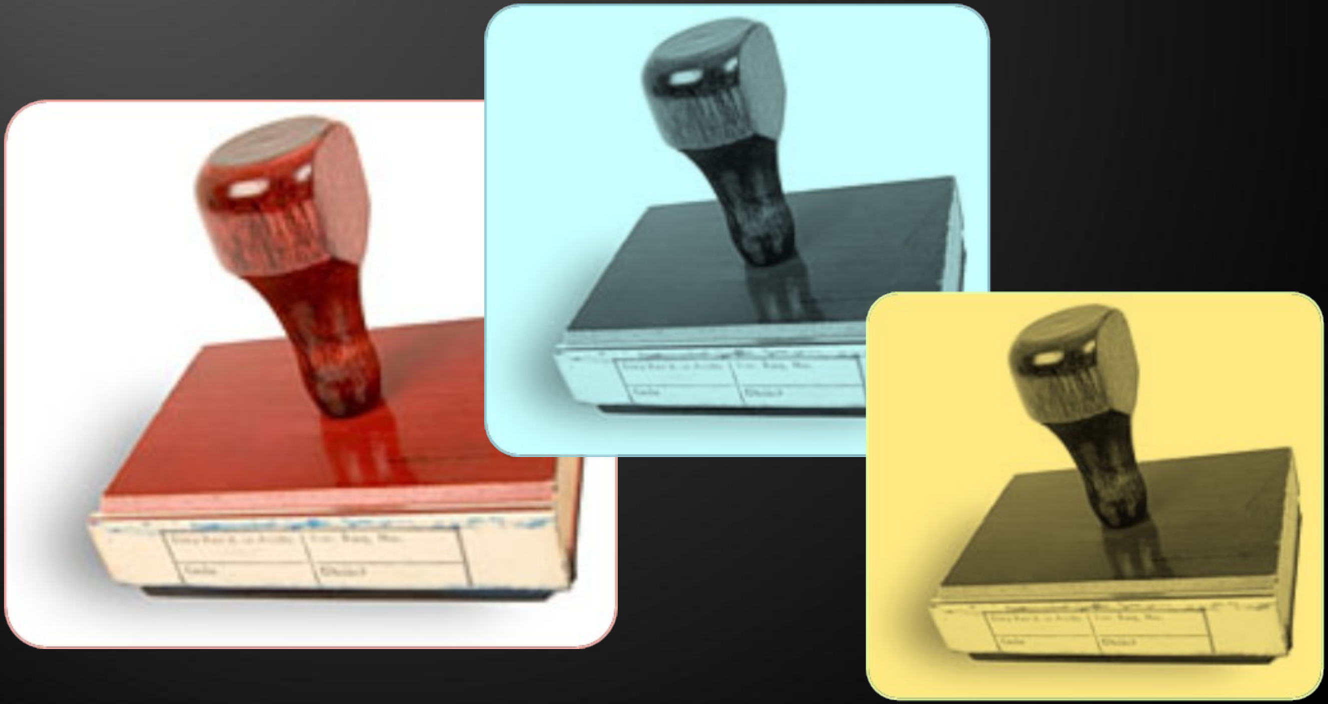
(example continues)

C# Structures – Example (2)

```
struct Square
{
   public Point Location { get; set; }
   public int Size { get; set; }
   public Color SurfaceColor { get; set; }
   public Color BorderColor { get; set; }

   public Square(Point location, int size,
    Color surfaceColor, Color borderColor) : this()
   {
      this.Location = location;
      this.Size = size;
      this.SurfaceColor = surfaceColor;
      this.BorderColor = borderColor;
   }
}
```

Generic Classes

Parameterized Classes and Methods



What are Generics?

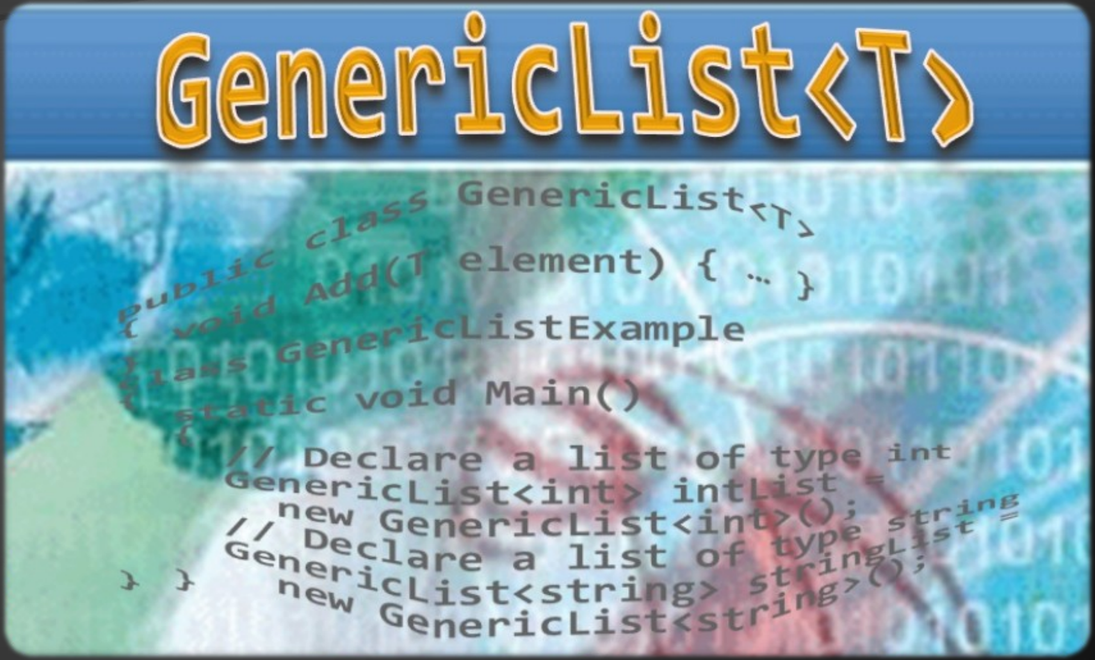
- Generics allow defining parameterized classes that process data of unknown (generic) type
  - The class can be instantiated with several different particular types
  - Example: `List<T>` → `List<int>` / `List<string>` / `List<Student>`
- Generics are also known as "parameterized types" or "template types"
  - Similar to the templates in C++
  - Similar to the generics in Java

Generics – Example

```
public class GenericList<T>
{
  public void Add(T element) { … }
}

class GenericListExample
{
  static void Main()
  {
    // Declare a list of type int
    GenericList<int> intList =
      new GenericList<int>();

    // Declare a list of type string
    GenericList<string> stringList =
      new GenericList<string>();
  }
}
```

T is an unknown type, parameter of the class

T can be used in any method in the class


GenericList<T>
public class GenericList<T>
{ void Add(T element) { … }
}
class GenericListExample
{ static void Main()
{
// Declare a list of type int
GenericList<int> intList =
new GenericList<int>();
// Declare a list of type string
GenericList<string> stringList =
new GenericList<string>();
} }

Generic Classes

Live Demo

Summary

- Classes define specific structure for objects
  - Objects are particular instances of a class and use this structure
- Constructors are invoked when creating new class instances
- Properties expose the class data in safe, controlled way
- Static members are shared between all instances
  - Instance members are per object
- Structures are classes that a primitive type

Defining Classes and Objects

Questions?

Exercises

1. Define a class that holds information about a mobile phone device: model, manufacturer, price, owner, battery characteristics (model, hours idle and hours talk) and display characteristics (size and colors). Define 3 separate classes: `GSM`, `Battery` and `Display`.
2. Define several constructors for the defined classes that take different sets of arguments (the full information for the class or part of it). The unknown data fill with `null`.
3. Add a static field `NokiaN95` in the `GSM` class to hold the information about Nokia N95 device.

telerik

Exercises (2)

1. Add a method in the class `GSM` for displaying all information about it.
2. Use properties to encapsulate data fields inside the `GSM`, `Battery` and `Display` classes.
3. Write a class `GSMTest` to test the functionality of the `GSM` class:
   - Create several instances of the class and store them in an array.
   - Display the information about the created `GSM` instances.
   - Display the information about the static member `NokiaN95`.

Exercises (3)

1. Create a class `Call` to hold a call performed through a GSM. It should contain date, time and duration.
2. Add a property `CallsHistory` in the `GSM` class to hold a list of the performed calls. Try to use the system class `List<Call>`.
3. Add methods in the `GSM` class for adding and deleting calls to the calls history. Add a method to clear the call history.
4. Add a method that calculates the total price of the calls in the call history. Assume the price per minute is given as parameter.

Exercises (4)

1. Write a class `GSMCallHistoryTest` to test the call history functionality of the `GSM` class.
   - Create an instance of the `GSM` class.
   - Add few calls.
   - Display the information about the calls.
   - Assuming that the price per minute is 0.37 calculate and print the total price of the calls.
   - Remove the longest call from the history and calculate the total price again.
   - Finally clear the call history and print it.

Exercises (5)

1. Write generic class `GenericList<T>` that keeps a list of elements of some parametric type `T`. Keep the elements of the list in an array with fixed capacity which is given as parameter in the class constructor. Implement methods for adding element, accessing element by index, removing element by index, inserting element at given position, clearing the list, finding element by its value and `ToString()`. Check all input parameters to avoid accessing elements at invalid positions.
2. Implement auto-grow functionality: when the internal array is full, create a new array of double size and move all elements to it.

Exercises (6)

1. Define class `Fraction` that holds information about fractions: numerator and denominator. The format is "numerator/denominator".
2. Define static method `Parse()` which is trying to parse the input string to fraction and passes the values to a constructor.
3. Define appropriate constructors and properties. Define property `DecimalValue` which converts fraction to rounded decimal value.
4. Write a class `FractionTest` to test the functionality of the `Fraction` class. Parse a sequence of fractions and print their decimal values to the console.

Exercises (7)

1. We are given a library of books. Define classes for the library and the books. The library should have name and a list of books. The books have title, author, publisher, year of publishing and ISBN. Keep the books in List<Book> (first find how to use the class `System.Collections.Generic.List<T>`).
2. Implement methods for adding, searching by title and author, displaying and deleting books.
3. Write a test class that creates a library, adds few books to it and displays them. Find all books by Nakov, delete them and print again the library.