Telerik Academy

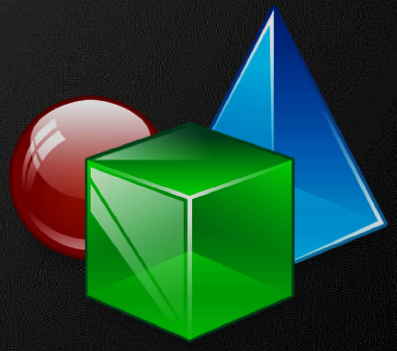
Defining Classes – Part 2

Static Members, Structures,Enumerations, Generic Classes, Namespaces

C# OOP

Telerik Software Academy

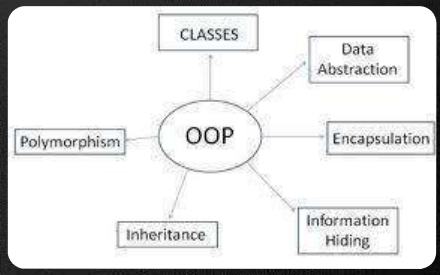
https://telerikacademy.com

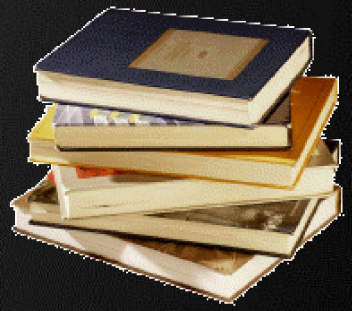


Follow us

Telerik Academy

Table of Contents





Telerik Academy

Static Members

Static vs. Instance Members

Telerik Academy

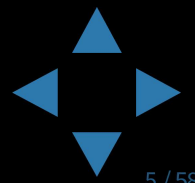
Static Members

- 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`

Telerik Academy

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



Telerik Academy

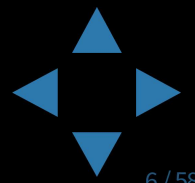
Static Members – *Example*

```
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)



Telerik Academy

Static Members – *Example*

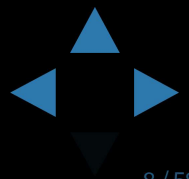
```
    // Static method
    public static int GetSqrt(int value)
    {
        return sqrtValues[value];
    }
}

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

Telerik Academy

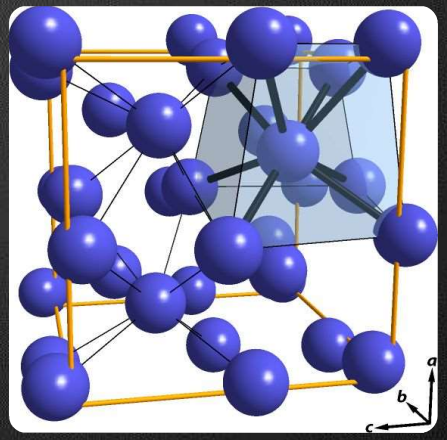
Static Members

Demo



Telerik Academy

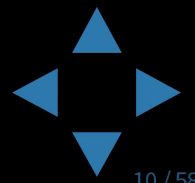
C# Structures



Telerik Academy

C# Structures

- What is a `structure` in C#?
  - A value data type (behaves like a primitive type)
    - *Examples* of structures: `int`, `double`, `DateTime`
    - Classes are reference types
  - Declared by the keyword `struct`
  - Structures, like classes, have properties, methods, fields, constructors, events, …
  - Always have a parameterless constructor
    - It cannot be removed
  - Mostly used to store data (bunch of fields)



Telerik Academy

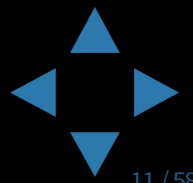
C# Structures – *Example*

```
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; }
}

enum Edges { Straight, Rounded }
```

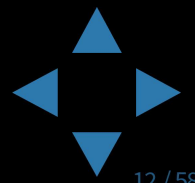
(example continues)



Telerik Academy

C# Structures – *Example*

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

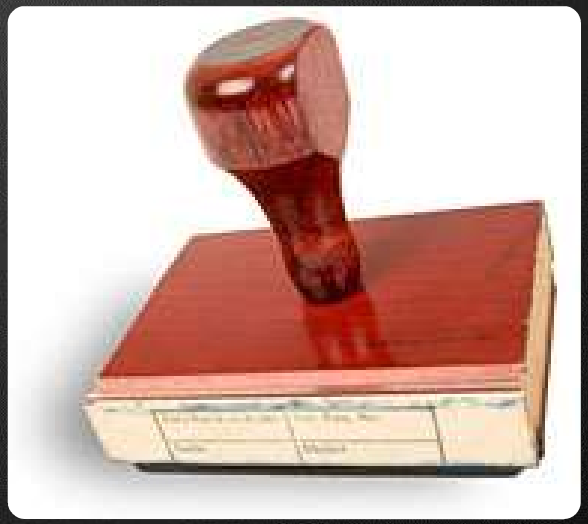
Telerik Academy

C# Structures

Demo

Telerik Academy

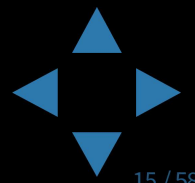
Generic Classes



Telerik Academy

What are Generics?

- Generics allow defining parameterized classes that process data of unknown (generic) type
  - The class can be instantiated (specialized) with 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



Telerik Academy

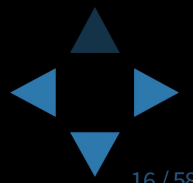
Generics – *Example*

```
public class GenericList<T>
{
  public void Add(T element) { … }
}
class GenericList_Example_
{
  static void Main()
  {
    // Declare a list of type
    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

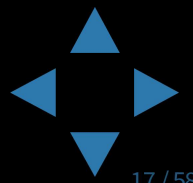
T can be used in any method in the class

T can be replaced with int during the instantiation



Telerik Academy

Generic Classes



Telerik Academy

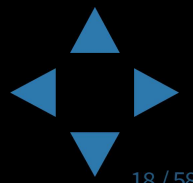
Defining Generic Classes

- Generic class declaration:

```
class MyClass   : class-base
where
{
  // Class body
}
```

- *Example*:

```
class MyClass : BaseClass
where T : new()
{
  // Class body
}
```

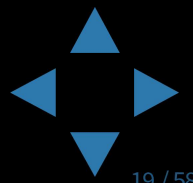
Telerik Academy Generic Constraints Syntax

- Parameter constraints clause:

```
public SomeGenericClass
where type-parameter : primary-constraint,
    secondary-constraints,
    constructor-constraint
```

- *Example*:

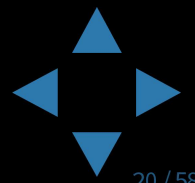
```
public class MyClass
where T: class, IEnumerable, new()
{…}
```

Telerik Academy

Generic Constraints

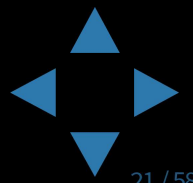
- Primary constraint:
  - `class` (reference type parameters)
  - `struct` (value type parameters)
- Secondary constraints:
  - Interface derivation
  - Base class derivation
- Constructor constraint:
  - `new()` – parameterless constructor constraint



Telerik Academy

Generic Constraints

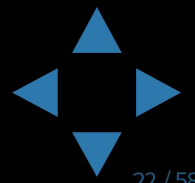
Demo



Telerik Academy

Generic Method – *Example*

```
public static T Min(T first, T second)
    where T : IComparable
{
    if (first.CompareTo(second) <= 0)
        return first;
    else
        return second;
}
static void Main()
{
    int i = 5;
    int j = 7;
    int min = Min(i, j);
}
```

Telerik Academy

Generic Methods

Demo

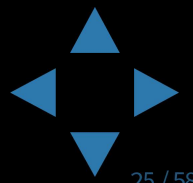
Telerik Academy

Namespaces

Telerik Academy

Namespaces

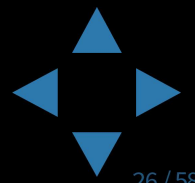
- `Namespaces` logically group type definitions
  - May contain classes, structures, interfaces, enumerators and other types and namespaces
  - Can not contain methods and data directly
  - Can be allocated in one or several files



Telerik Academy

Namespaces

- Namespaces in .NET are similar to namespaces in C++ and packages in Java
- Allows definition of types with duplicated names
  - E.g. a type named Button is found in Windows Forms, in WPF and in ASP.NET Web Forms



Telerik Academy

Including Namespaces

- Including a namespace
  - The `using` directive is put at the start of the file

```
using System.Windows.Forms;
```

  - `using` allows direct use of all types in the namespace
  - Including is applied to the current file
  - The directive is written at the beginning of the file
  - When includes a namespace with `using` its subset of namespaces is not included

Telerik Academy

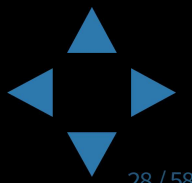
Including Namespaces

- Types, placed in namespaces, can be used and without `using` directive, by their full name:

```
System.IO.StreamReader reader =
    System.IO.File.OpenText("file.txt");
```

- `using` can create alias for namespaces :

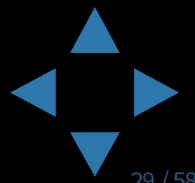
```
using IO = System.IO;
using WinForms = System.Windows.Forms;
IO.StreamReader reader =
    IO.File.OpenText("file.txt");
WinForms.Form form = new WinForms.Form();
```

Telerik Academy

Defining Namespaces

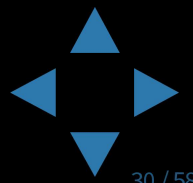
- Divide the types in your applications into namespaces
  - When the types are too much (more than 15-20)
  - Group the types logically in namespaces according to their purpose
- Use nested namespaces when the types are too much
  - E.g. for Tetris game you may have the following namespaces: Tetris.Core, Tetris.Web, Tetris.Win8, Tetris.HTML5Client



Telerik Academy

Defining Namespaces

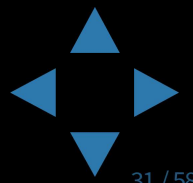
- Distribute all public types in files identical with their names
  - E.g. the class Student should be in the file Student.cs
- Arrange the files in directories, corresponding to their namespaces
  - The directory structure from your project course-code have to reflect the structure of the defined namespaces



Telerik Academy

Namespaces – *Example*

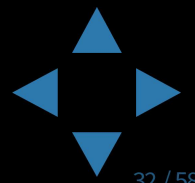
```
namespace SofiaUniversity.Data
{
    public struct Faculty
    {
        // …
    }
    public class Student
    {
        // …
    }
    public class Professor
    {
        // …
    }
    public enum Specialty
    {
        // …
    }
}
```



Telerik Academy

Namespaces – *Example*

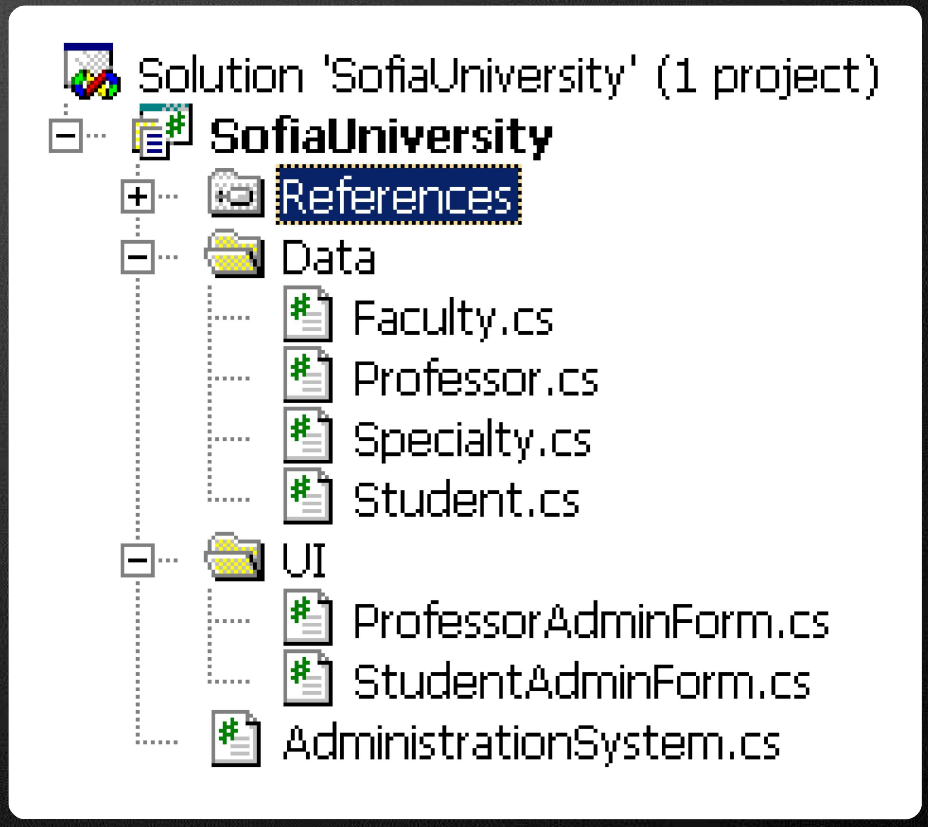
```
namespace SofiaUniversity.UI
{
    public class StudentAdminForm : System.Windows.Forms.Form
    {
        // …
    }
    public class ProfessorAdminForm : System.Windows.Forms.Form
    {
        // …
    }
}
namespace SofiaUniversity
{
    public class AdministrationSystem
    {
        public static void Main()
        {
            // …
        }
    }
}
```

Telerik Academy

Namespaces – *Example*

- Recommended directory structure and classes organization in them



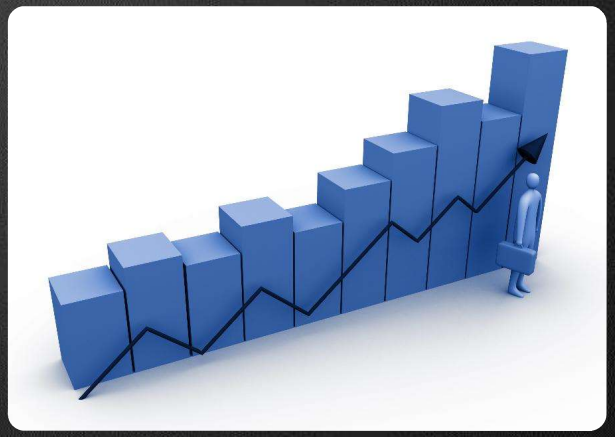
Telerik Academy

Namespaces

Demo

Telerik Academy

Indexers



Telerik Academy

Indexers

- Indexers provide indexed access class data
  - Predefine the [ ] operator for certain type
    - Like when accessing array elements

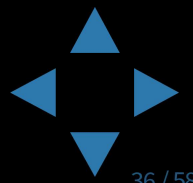
```
IndexedType t = new IndexedType(50);
int i = t[5];
t[0] = 42;
```

- Can accept one or multiple parameters

```
personInfo["Nikolay Kostov", 25]
```

- Defining an indexer:

```
public int this [int index] { … }
```

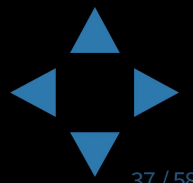


Telerik Academy

Indexers – *Example*

```
struct BitArray32
{
    private uint value;
    // Indexer declaration
    public int this [int index]
    {
        get
        {
            if (index >= 0 && index <= 31)
            {
                // Check the bit at position index
                if ((value & (1 << index)) == 0)
                   return 0;
                else
             return 1;
            }
```

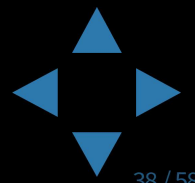
(the example continues)



Telerik Academy

Indexers – *Example*

```
    else
    {
        throw new IndexOutOfRangeException(
        String.Format("Index {0} is invalid!", index));
    }
}
set
{
    if (index < 0 || index > 31)
        throw new IndexOutOfRangeException(
         String.Format("Index {0} is invalid!", index));
    if (value < 0 || value > 1)
        throw new ArgumentException(
         String.Format("Value {0} is invalid!", value));
    // Clear the bit at position index
    value &= ~((uint)(1 << index));
    // Set the bit at position index to value
    value |= (uint)(value << index);
}
```



Telerik Academy

Indexers

Demo

Telerik Academy

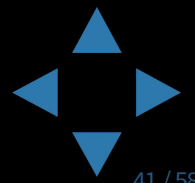
Operators Overloading

Telerik Academy

Overloading Operators

- In C# some operators can be overloaded(redefined) by developers
  - The priority of operators can not be changed
  - Not all operators can be overloaded
- Overloading an operator in C#
  - Looks like a static method with 2 operands:

```
public static Matrix operator *(Matrix m1, Matrix m2)
{
    return new m1.Multiply(m2);
}
```

Telerik Academy

Overloading Operators

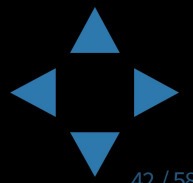
- Overloading is allowed on:
  - Unary operators

```
+, -, !, ~, ++, --, true and false
```

- Binary operators

```
+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >= and <=
```

- Operators for type conversion
  - Implicit type conversion
  - Explicit type conversion (type)

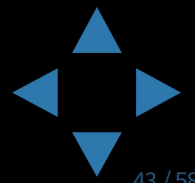


Telerik Academy

Overloading Operators – *Example*

```
public static Fraction operator -(Fraction f1,Fraction f2)
{
    long num = f1.numerator * f2.denominator -
    f2.numerator * f1.denominator;
    long denom = f1.denominator * f2.denominator;
    return new Fraction(num, denom);
}
public static Fraction operator *(Fraction f1,Fraction f2)
{
    long num = f1.numerator * f2.numerator;
    long denom = f1.denominator * f2.denominator;
    return new Fraction(num, denom);
}
```

(the example continues)



Telerik Academy

Overloading Operators – *Example*

```
// Unary minus operator
public static Fraction operator -(Fraction fraction)
{
    long num = -fraction.numerator;
    long denom = fraction.denominator;
    return new Fraction(num, denom);
}
// Operator ++ (the same for prefix and postfix form)
public static Fraction operator ++(Fraction fraction)
{
    long num = fraction.numerator + fraction.denominator;
    long denom = Frac.denominator;
    return new Fraction(num, denom);
}
```

Telerik Academy

Overloading Operators

Demo

Telerik Academy

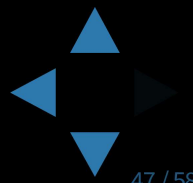
Attributes

Applying Attributes to Code Elements

Telerik Academy

What are Attributes?

- .NET `attributes` are:
  - Declarative tags for attaching descriptive information in the declarations in the code
  - Saved in the assembly at compile time
    - Objects derived from `System.Attribute`
  - Can be accessed at runtime (through reflection) and manipulated by many tools
- Developers can define custom attributes



Telerik Academy Applying Attributes – *Example*

- Attribute`s name is surrounded by square brackets []
  - Placed before their target declaration

```
[Flags] // System.FlagsAttribute
public enum FileAccess
{
    Read = 1,
    Write = 2,
    ReadWrite = Read | Write
}
```

- [Flags] attribute indicates that the enum type can be treated like a set of bit flags

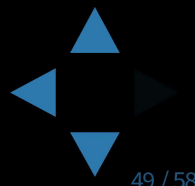
Telerik Academy

Attributes with Parameters

- Attributes can accept parameters for their constructors and public properties

```
[DllImport("user32.dll", EntryPoint="MessageBox")]
public static extern int ShowMessageBox(int hWnd,
    string text, string caption, int type);
…
ShowMessageBox(0, "Some text", "Some caption", 0);
```

- The `[DllImport]` attribute refers to:
  - `System.Runtime.InteropServices.DllImportAttribute`
  - "user32.dll" is passed to the constructor
  - "MessageBox" value is assigned to EntryPoint

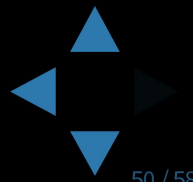


Telerik Academy Set a Target to an Attribute

- Attributes can specify its target declaration:

```
// target "assembly"
[assembly: AssemblyTitle("Attributes Demo")]
[assembly: AssemblyCompany("DemoSoft")]
[assembly: AssemblyProduct("Entreprise Demo Suite")]
[assembly: AssemblyVersion("2.0.1.37")]
[Serializable] // [type: Serializable]
class TestClass
{
    [NonSerialized] // [field: NonSerialized]
    private int status;
}
```

- See the Properties/AssemblyInfo.cs file



Telerik Academy

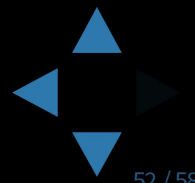
Using Attributes

Demo

Telerik Academy

Custom Attributes

- .NET developers can define their own custom attributes
  - Must inherit from `System.Attribute` class
  - Their names must end with '`Attribute`'
  - Possible targets must be defined via `[AttributeUsage]`
  - Can define constructors with parameters
  - Can define public fields and properties



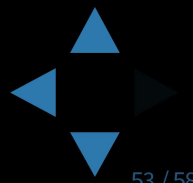
Telerik Academy

Custom Attributes – *Example*

```
[AttributeUsage(AttributeTargets.Struct |
  AttributeTargets.Class | AttributeTargets.Interface,
  AllowMultiple = true)]
public class AuthorAttribute : System.Attribute
{
  public string Name { get; private set; }

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

(example continues)



Telerik Academy

Custom Attributes – *Example*

```
[Author("Doncho Minkov")]
[Author("Nikolay Kostov")]
class CustomAttributesDemo
{
  static void Main(string[] args)
  {
    Type type = typeof(CustomAttributesDemo);
    object[] allAttributes =
      type.GetCustomAttributes(false);
    foreach (AuthorAttribute attr in allAttributes)
    {
      Console.WriteLine(
        "This class is written by {0}. ", attr.Name);
    }
  }
}
```

Telerik Academy

Defining, Applying and Retrieving Custom Attributes

Demo

Telerik Academy

Summary

- Classes define specific structure for objects
  - Objects are particular instances of a class
- 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 "value-type" classes
- Generics are parameterized classes