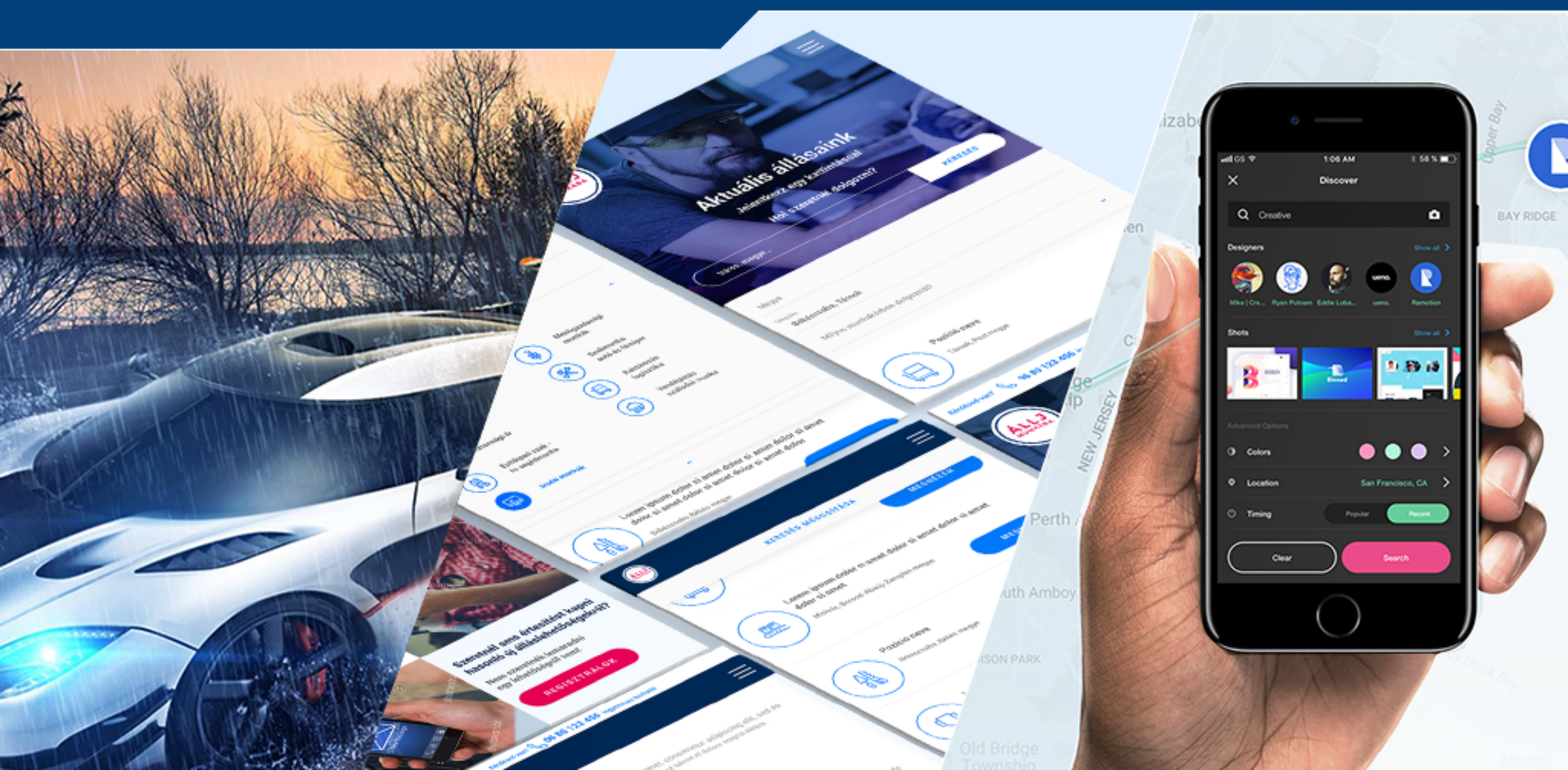
VT Academy
Thinking Ahead, Beyond Boundaries

Object Oriented Programming



Property, Indexer, Namespace and Exception

Session 7

Objectives

VTC Academy
Thinking Ahead, Beyond Boundaries

- Properties
- Indexers
- Namespaces
- Exceptions

Properties

- A 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. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.
- Syntax:

```
public data_type Property_Name
{
    get
    {
        return private_field;
    }
    set
    {
        private_field = value;
    }
}
```

Properties Overview

- Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code
- A get property accessor is used to return the property value, and a set property accessor is used to assign a new value
- The **value** keyword is used to define the value being assigned by the set accessor
- Properties can be:
  - Read-Write: have both **get** and **set** accessor
  - Read-only: have only **get** accessor
  - Write-only: have only **set** accessor
- Write-only properties are rare and are most commonly used to restrict access to sensitive data

Properties Example

```
class Student {
   private int studentid = 0;
   private string studentname = "N/A";
   public string StudentID  // Declare a StudentID property of type int
   {
      get
      {
         return studentid;
      }
      set
      {
         studentid = value;
      }
   }
   public string StudentName // Declare a StudentName property of type string
   {
      get
      {
         return studentname;
      }
      set
      {
         studentname = value;
      }
   }
   public override string ToString()
   {
      return "Student ID = " + StudentID + ", Student Name = " + StudentName;
   }
}
```

Abstract Properties

- An abstract class may have an abstract property, which should be implemented in the derived class.
- Example:

```
public abstract class Person
{
    public abstract string Name
    {
        get;
        set;
    }
    public abstract int Age
    {
        get;
        set;
    }
}
```

Abstract Properties

```
class Student : Person
{
    private string name = "N/A";
    private int age = 0;
    // Declare a Name property of type string
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
    // Declare a Age property of type int
    public override int Age
    {
        get
        {
            return age;
        }
        set
        {
            age = value;
        }
    }
}
```

Indexers

- An **indexer** allows an object to be indexed such as an array. When you define an indexer for a class, this class behaves similar to a **virtual array**.
- You can then access the instance of this class using the array access operator ([ ])
- One dimensional indexer has the following syntax:

```
element-type this[int index]
{
   // The get accessor.
   get
   {
      // return the value specified by index
   }

   // The set accessor.
   set
   {
      // set the value specified by index
   }
}
```

Use of Indexers

- Indexers are a syntactic convenience that enable you to create a class, struct, or interface that client applications can access just as an array
- Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array
- For example:
  - TempRecord class that represents the temperature in Farenheit as recorded at 10 different times during a 24 hour period (contains an temperatures array and a DateTime that represents the date the temperatures were recorded)
  - By implementing an indexer in this class, clients can access the temperatures in a TempRecord instance as `float temp = tr[4]` instead of as **float temp = tr.temps[4].**

Use of Indexers

- To declare an indexer on a class or struct, use the this keyword:

```
public int this[int index]    // Indexer declaration
{
    // get and set accessors
}
```

- Example of Indexers:
  - Continue with TempRecord class example
  - Declaring a private array field and an indexer
  - Using the indexer is to declare the array as a public member and access its members

Example of Indexers

```
class TempRecord
{
    // Array of temperature values
    private float[] temps = new float[10] { 56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
                                            61.3F, 65.9F, 62.1F, 59.2F, 57.5F };
    // To enable client code to validate input when accessing your indexer
    public int Length
    {
        get { return temps.Length; }
    }
    // Indexer declaration
    // If index is out of range, the temps array will throw the exception
    public float this[int index]
    {
        get
        {
            return temps[index];
        }

        set
        {
            temps[index] = value;
        }
    }
}
```

Example of Indexers (cont..)

```
class Program
{
    static void Main()
    {
        TempRecord tempRecord = new TempRecord();
        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            System.Console.WriteLine("Element #{0} = {1}", i, tempRecord[i]);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();

    }
}
```

Indexers in Interfaces

- Indexers can be declared on an interface. Accessors of interface indexers differ from the accessors of class indexers in:
  - Interface accessors do not use modifiers
  - An interface accessor does not have a body
- The purpose of the accessor is to indicate whether the indexer is read-write, read-only, or write-only
- Example of an interface indexer accessor:

```
public interface ISomeInterface
{
    //...
    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

How to Implement Interface Indexers?


```
// Indexer on an interface:
public interface ISomeInterface
{
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}
```

```
// Implementing the interface.
class IndexerClass : ISomeInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        get
        {
            // The arr object will throw IndexOutOfRange exception.
            return arr[index];
        }
        set
        {
            arr[index] = value;
        }
    }
}
```

How to Implement Interface Indexers?


```
class Program
{
    static void Main()
    {
        IndexerClass test = new IndexerClass();
        System.Random rand = new System.Random();
        // Call the indexer to initialize its elements.
        for (int i = 0; i < 10; i++)
        {
            test[i] = rand.Next();
        }
        for (int i = 0; i < 10; i++)
        {
            System.Console.WriteLine("Element #{0} = {1}", i, test[i]);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Properties vs Indexers

Property	Indexer
Allows methods to be called as if they were public data members.	Allows elements of an internal collection of an object to be accessed by using array notation on the object itself.
Accessed through a simple name.	Accessed through an index.
Can be a static or an instance member.	Must be an instance member.
A get accessor of a property has no parameters.	A get accessor of an indexer has the same formal parameter list as the indexer.
A set accessor of a property contains the implicit value parameter.	A set accessor of an indexer has the same formal parameter list as the indexer, and also to the value parameter.
Supports shortened syntax with Auto-Implemented Properties.	Does not support shortened syntax.

Namespaces

- A **namespace** is designed for providing a way to keep one set of names separate from another.
- The class names declared in one namespace does not conflict with the same class names declared in another.
- A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace namespace_name
{
   // code declarations
}
```

Namespaces

- A **namespace** is designed for providing a way to keep one set of names separate from another.
- The class names declared in one namespace does not conflict with the same class names declared in another
- Namespaces implicitly have public access and this is not modifiable
- A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace namespace_name
{
   // code declarations
}
```

Namespaces Overview

- They organize large code projects.
- They are delimited by using the . (dot) operator.
- The `using directive` obviates the requirement to specify the name of the namespace for every class.
- The `global` namespace is the "root" namespace: `global::System` will always refer to the .NET Framework namespace `System`.

Example of Namespaces

VTC Academy
Thinking Ahead, Beyond Boundaries

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

System Namespaces

- System namespace is imported by default in the .NET Framework
- The System namespace contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
- Example of resources in System namespace:
  - System.Console
  - System.Text
  - System.Colllections
  - System.IO
  - System.Data

Accessing Namespaces

- By using top-level section of C# application - **using** directives.
- **using** directives section lists the namespaces that the application will be using frequently, and saves the programmer from specifying a fully qualified name every time that a method that is contained within is used
- For example, by including the line:
- After declaring this directive:

```
using System;
```

- In the method use can use statment:

```
Console.WriteLine("Hello, World!");
```

- Instead of:

```
System.Console.WriteLine("Hello, World!");
```

Namespace Aliases

- The using Directive can also be used to create an alias for a namespace
- For example, if you are using a previously written namespace that contains nested namespaces, you might want to declare an alias to provide a shorthand way of referencing one in particular, as in the following example:

```
// define an alias to represent a namespace
using Co = Company.Proj.Nested;
```

Using Namespaces to Control Scope

- The **namespace** keyword is used to declare a scope. The ability to create scopes within your project helps organize code and lets you create globally-unique types.
- In the following example, a class titled **SampleClass** defined in two namespaces, one nested inside the other. The . (dot) Operator is used to differentiate which method gets called.

Using Namespaces to Control Scope


```
namespace SampleNamespace {
    class SampleClass {
        public void SampleMethod()
        {
            System.Console.WriteLine("SampleMethod inside SampleNamespace");
        }
    }
    // create a nested namespace, and define another class
    namespace NestedNamespace {
        class SampleClass {
            public void SampleMethod()
            {
                System.Console.WriteLine("SampleMethod inside NestedNamespace");
            }
        }
    }
    class Program {
        static void Main(string[] args)
        {
            // display SampleMethod inside SampleNamespace
            SampleClass outer = new SampleClass();
            outer.SampleMethod();
            // display SampleMethod inside SampleNamespace
            SampleNamespace.SampleClass outer2 = new SampleNamespace.SampleClass();
            outer2.SampleMethod();
            // display SampleMethod inside NestedNamespace
            NestedNamespace.SampleClass inner = new NestedNamespace.SampleClass();
            inner.SampleMethod();
        }
    }
}
```

Exceptions

- An exception is a problem that arises during the execution of a program.
- A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.
- Exceptions provide a way to transfer control from one part of a program to another.

Exceptions

- C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.
  - **try**: A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
  - **catch**: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
  - **finally**: The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
  - **throw**: A program throws an exception when a problem shows up. This is done using a throw keyword.

Exceptions Handling Syntax

- Assuming a block raises an exception, a method catches an exception using a combination of the try and catch keywords.
- A try/catch block is placed around the code that might generate an exception looks like the following:

```
try
{
   // statements causing exception
}
catch( ExceptionName e1 )
{
   // error handling code
}
catch( ExceptionName e2 )
{
   // error handling code
}
finally
{
   // statements to be executed
}
```

Exception Classes in C#

VTC Academy
Thinking Ahead, Beyond Boundaries

Exception Class	Description
System.IO.IOException	Handles I/O errors.
System.IndexOutOfRangeException	Handles errors generated when a method refers to an array index out of range.
System.ArrayTypeMismatchException	Handles errors generated when type is mismatched with the array type.
System.NullReferenceException	Handles errors generated from referencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.
System.StackOverflowException	Handles errors generated from stack overflow.

Handling Exceptions

- C# provides a structured solution to the exception handling in the form of try and catch blocks.
- Using these blocks the core program statements are separated from the error-handling statements.
- These error handling blocks are implemented using the **try**, **catch**, and **finally** keywords.
- Following is an example of throwing an exception when dividing by zero condition occurs:

Handling Exceptions (cont..)

```
using System;
namespace ExceptionsHandlingDemo {
   class Program {
      int result;
      Program()
      {
         result = 0;
      }
      public void division(int num1, int num2)
      {
         try
         {
            result = num1 / num2;
         }
         catch (DivideByZeroException e)
         {
            Console.WriteLine("Exception caught: {0}", e);
         }
         finally
         {
            Console.WriteLine("Result: {0}", result);
         }
      }
      static void Main(string[] args)
      {
         Program p = new Program();
         p.division(25, 0);
         Console.ReadKey();
      }
   }
}
```

Creating User-Defined Exceptions

- User-defined exception classes are derived from the **Exception** class as following:

```
public class TempIsZeroException: Exception
{
    public TempIsZeroException(string message): base(message)
    {
    }
}
```

```
public class Temperature
{
    int temperature = 0;
    public void showTemp()
    {
        if(temperature == 0)
        {
            throw (new TempIsZeroException("Zero Temperature found"));
        }
        else
        {
            Console.WriteLine("Temperature: {0}", temperature);
        }
    }
```

Creating User-Defined Exceptions

- Write a program to test user-defined excepttion:

```
using System;
namespace UserDefinedExceptionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Temperature temp = new Temperature();
            try
            {
                temp.showTemp();
            }
            catch(TempIsZeroException e)
            {
                Console.WriteLine("TempIsZeroException: {0}", e.Message);
            }
            Console.ReadKey();
        }
    }
}
```

Summary

VTC Academy
Thinking Ahead, Beyond Boundaries

- A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field.
- An **indexer** allows an object to be indexed such as an array. When you define an indexer for a class, this class behaves similar to a **virtual array**
- A **namespace** is designed for providing a way to keep one set of names separate from another.
- System namespace is imported by default in the .NET Framework
- An exception is a problem that arises during the execution of a program.
- Exception handling blocks are implemented using the **try**, **catch**, and **finally** keywords.