Visual Programming Fundamentals: Program Structure

Indah Permatasari, M.Kom.

(2nd meeting)

Outline

- Framework .NET
- Visual C# vs Visual Basic
- Program Structure
  - Type
  - Variable
  - Class
  - Method
  - Operator

Framework .NET

- .NET is an **open source development platform** which is just a way of saying it's a collection of languages and libraries that can all work together to build all kinds of apps!
- The architects of .NET realized that all **procedural languages** require certain base functionality.
- For example, many languages ship with their own runtime that provides features such as memory management, but what if, instead of each language shipping with its own runtime implementation, all languages used a common runtime? This would provide languages with a standard environment and access to all of the same features. This is exactly what the **CLR** provides.
- The CLR manages the execution of code on the .NET platform.
- The CLR enabled Microsoft to concentrate on building this plumbing one time and then reuse it across ***different programming languages***.

Language .NET

- .NET apps in C#, F#, or Visual Basic.
  - **C#** is a simple, modern, object-oriented, and type-safe programming language.
  - **F#** is a cross-platform, open-source, functional programming language for .NET. It also includes object-oriented and imperative programming.
  - **Visual Basic** is an approachable language with a simple syntax for building type-safe, object-oriented apps.

Language .NET

```
1  var names = new List<String>
2  {
3      "Ana",
4      "Felipe",
5      "Emillia"
6  };
7
8  foreach (var name in names)
9  {
10     Console.WriteLine($"Hello {name}");
11 }
```

```
let names = [ "Ana"; "Felipe"; "Emillia"]

for name in names do
    printfn $"Hello {name}"
```

```
Dim names As New List(Of String)({
    "Ana",
    "Felipe",
    "Emillia"
})

For Each name In names
    Console.WriteLine($"Hello {name}")
Next
```

Element of a .NET application

- A .NET application is composed of four primary entities:
  - **Types**—The common unit of transmitting data between modules running in the CLR
  - **Classes**—The basic units that encapsulate data and behavior
  - **Modules**—The individual files that contain the intermediate language (IL) for an assembly
  - **Assemblies**—The primary unit of deployment of a .NET application

Types

- Provides a template that is used to describe the encapsulation of data and an associated set of behaviors.
- Common template for describing data that provides the basis for the metadata that .NET uses when classes interoperate at runtime.
- A type has fields, properties, and methods:
  - **Fields**—Variables that are scoped to the type. For example, a Pet class could declare a field called Name that holds the pet's name. In a well-engineered class, fields are typically kept private and exposed only as properties or methods.
  - **Properties**—Properties look like fields to clients of the type but can have code behind them (which usually performs some sort of data validation). For example, a Dog data type could expose a property to set its gender. Code could then be placed within the property definition so that only "male" or "female" are accepted values. Internally the selected value might be transformed and stored in a more efficient manner using a fi eld in the Dog class.
  - **Methods**—Methods define behaviors exhibited by the type. For example, the Dog data type could expose a method called Sleep, which would suspend the activity of the Dog.

Classes

- A Class is the code that defines an object, and all objects are created based on a class.
- Explains the properties and behavior of an object.
- Provides the basis from which you create instances of specific objects.
- For example, in order to have a *Customer* object representing customer number 123, you must first have a *Customer* class that contains all of the code (methods, properties, events, variables, and so on) necessary to create *Customer* objects. Based on that class, you can create any number of *Customer* instances. Each instance of the object *Customer* is identical to the others, except that it may contain different data. This means that each object represents a different customer.

Modules

- A module contains Microsoft intermediate language (MSIL, often abbreviated to IL) code, associated metadata, and the assembly's manifest.
- A platform-independent way of representing managed code within a module. Before IL can be executed, the CLR must compile it into the native machine code. The default method is for the CLR to use the JIT (just-in-time) compiler to compile the IL on a method-by-method basis.

Assemblies

- An assembly is the primary unit of deployment for .NET applications.
- Either a dynamic link library (.dll) or an executable (.exe). An assembly is composed of a manifest, one or more modules, and (optionally) other fi les, such as .config, .ASPX, .ASMX, images, and so on.
- The manifest of an assembly contains the following:
  - Information about the identity of the assembly, including its textual name and version number.
  - If the assembly is public, then the manifest contains the assembly's public key.
  - A declarative security request that describes the assembly's security requirements.
  - A list of other assemblies on which the assembly depends: this information to locate an appropriate version of the required assemblies at runtime.
  - A list of all types and resources exposed by the assembly.

Visual C# vs Visual Basic

- Visual C#
  - Development tool that using C programming language
  - C programming language are using Object-oriented programming (OOP) that focused on data.
  - Represents the composition of problem into modules, that represents into objects are called class and type.

```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

Visual C# vs Visual Basic

- Visual Basic
  - Development tool that using BASIC Programming Language
  - BASIC programming language is **B**eginner **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode.
  - Easier to learn than others as its commands are similar to English and has a simple set of rules for entering them.

```
Module Example
   Public Sub Main()
      Dim arr(3, 3) As Integer
      Console.WriteLine(arr.Length)
   End Sub
End Module
' The example displays the following output:
'      16
```

Program Structure

Visual C# (C Programming Language)

Namespaces

- .NET uses namespaces to organize its many classes

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

  - **System** is a namespace and **Console** is a class in that namespace.
- declaring your own namespaces can help you control the scope of class and method names in larger programming projects.

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

Namespaces Properties

- They organize large code projects.
- They are delimited by using the **.** 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 System namespace.

Type and Variable

Variable

- Define of the name given a storage data
- Declare constant in a program that hold a data

Type

- C# is a strongly typed language.
- Every variable and constant has a type, as does every expression that evaluates to a value.
- Every method signature specifies a type for each input parameter and for the return value.

Type and Variable (2)

- There are two kinds of types in C#: value types and reference types.
- Variables of **value types** directly contain their data, and **reference types** store references to their data, the latter being known as objects.
- With **reference types**, it's *possible* for two variables to reference the **same** object and possible for operations on one variable to affect the object referenced by the other variable
- With **value types**, the variables each have their own copy of the data, and it isn't possible for operations on one to affect the other (except for ref and out parameter variables)

Type and Variable (3)

- C#'s value types are further divided into ***simple types, enum types, struct types, nullable value types, and tuple value types***.
- C#'s reference types are further divided into ***class types, interface types, array types, and delegate types.***
- C# programs use type declarations to create new types.
  - A type declaration specifies the name and the members of the new type.
- There are several kinds of variables in C#, including fields, array elements, local variables, and parameters. Variables represent storage locations.

Declare of Type, Variable and Operator

- The compiler uses type information to make sure that all operations that are performed in your code are *type safe*. For example, if you declare a variable of type **int**, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type **bool**, the compiler generates an error, as shown in the following example:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

Class

- A class is a reference type.
- When you declare a variable of a reference type, the variable contains the value *null* until you explicitly create an instance of the class by using the *new* operator, or assign it an object of a compatible type that may have been created elsewhere

```
//Declaring an object of type MyClass.
MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the first object.
MyClass mc2 = mc;
```

Class (2)

- Data structure that combines state (fields) and actions (methods and other function members) in a single unit.
- Provides a definition for instances of the class, also known as objects.
- Classes support inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);
}
```

Method / Function

- A method is a code block that contains a series of statements.
- Local functions are private methods of a type that are nested in another member.

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) {
    /* Method statements here */
    }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) {
      /* Method statements here */
    return 1;
    }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Identifier Names

- An **identifier** is the name you assign to a type (class, interface, struct, delegate, or enum), member, variable, or namespace.
- Valid identifiers must follow these rules:
  - Identifiers must start with **a letter, or _**.
    - Example: namespace, a, visual or _name, _a
  - Identifiers may contain Unicode letter characters, decimal digit characters, Unicode connecting characters, Unicode combining characters, or Unicode formatting characters.

Operators

- defines the meaning of applying a particular expression operator to instances of a class.
- Three kinds of operators can be defined: unary operators, binary operators, and conversion operators.
- All operators must be declared as **public** and **static**.

Operators (2)

- The **MyList<T>** class declares two operators, operator **==** and operator **!=**. These overridden operators give new meaning to expressions that apply those operators to **MyList** instances. Specifically, the operators define equality of two **MyList<T>** instances as comparing each of the contained objects using their Equals methods. The following example uses the **==** operator to compare two **MyList<int>** instances.

```
MyList<int> a = new MyList<int>();
a.Add(1);
a.Add(2);
MyList<int> b = new MyList<int>();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b);  // Outputs "True"
b.Add(3);
Console.WriteLine(a == b);  // Outputs "False"
```

- The first **Console.WriteLine** outputs True because the two lists contain the same number of objects with the same values in the same order. Had **MyList<T>** not defined operator ==, the first **Console.WriteLine** would have output False because a and b reference different **MyList<int>** instances.

Program structure

- How about this code?
- Describe the parts in the following this code?
- Write your argument in discussion forum (vclass)

```
using System;

namespace Acme.Collections
{
    public class Stack<T>
    {
        Entry _top;

        public void Push(T data)
        {
            _top = new Entry(_top, data);
        }

        public T Pop()
        {
            if (_top == null)
            {
                throw new InvalidOperationException();
            }
            T result = _top.Data;
            _top = _top.Next;

            return result;
        }

        class Entry
        {
            public Entry Next { get; set; }
            public T Data { get; set; }

            public Entry(Entry next, T data)
            {
                Next = next;
                Data = data;
            }
        }
    }
}
```

Summary

- **.NET Framework** adalah platform yang digunakan untuk membangun aplikasi
- **Namespaces** adalah kumpulan class yang saling berhubungan dan memiliki kegunaan untuk menjalankan program
- **Class** digunakan untuk mendeklarasikan sebuah kelas
- **Variable** merupakan atribut atau data dari sebuah kelas, digunakan untuk menyimpan data
- **Function** merupakan sekumpulan statement yang menjalankan tugas tertentu
- **Identifier** merupakan nama yang digunakan untuk mengidentifikasikan sebuah kelas, variable, fungsi, atau item lainnya

Terima kasih.