C# Fundamentals For Visual Studio .NET Platform

Module 3

Agenda

- Course materials on internet
- Review homework problem
- Defining a class
- Declaring methods
- Using constructors
- Defining static class members

Course Materials on Internet

- Course materials are on the UCLA course site.
- To access the UCLA site
  - Email should have been sent to you
    - Web location information
    - Your personal username and password
  - Contact Frank Azzariti at (310) 206-3340 if you have not received your course login information.
- Web support for UCLA course site can be found at:

  www.uclaextension.edu/onlinehybridsupport

What is a Class/Object?

- Class
  - Template for an object
  - Similar to blueprints
  - Contain data and behavior (fields and methods)
- Object
  - An instance of a class
  - Use the **new** keyword to create
  - Has identity

How to Define a Class

```
public class Employee {
  public string firstName;
  public string lastName;
  public int age;
  public uint employeeID;
}
```

Instantiating Objects and Accessing Class Data

- Instantiating an object

```
Employee anEmployee = new Employee();
```

- Accessing class member data

```
anEmployee.firstName = “John";
anEmployee.lastName = “Smith”;
```

Namespace Usage

- Namespace declaration

```
namespace MyNamespace {
  public class Employee () { }
}
```

- Nest namespaces

```
namespace Organization {
  namespace Company {
      public class XYZ () { }
  }
}
// Or
namespace Organization.Company { ... }
```

- The using statement

```
using System;
using Organization.Company;
```

How to Define Accessibility and Scope

Use access modifiers to define class member accessibility

Modifier	Scope
public	No access limitation.
private	Accessible only to containing class.
internal	Any program element can access members.
protected	Accessible to containing class and derived classes
protected internal	Access rights is the union of internal and protected.

Methods

- Syntax

  {access modifier} {static or blank for instance} {return type} {method name} {parameter list} {code block}

- Example

```
public static void MyMethod(int someData)
{
	Console.WriteLine(“someData = {0}”, someData.ToString());
}
```

Methods (Continued)

- Main is a method
- Methods may be recursive
- Returning data
  - – Declare method with non-void type
  - – Use an expression with a **return** statement
  - – Non-void methods must return data
- Example

```
static int AddData(int i, int j){
  return i + j;
}
```

Using Local Variables

- Local variables
  - Declare within the method
  - Created when the method is called
  - Access scope is private to the method
  - Destroyed when method ends
- Shared variables
  - Class level variables are used for sharing data
- Scope conflicts
  - Local has precedence
  - Compiler will not warn if duplicate local and class names

Declaring and Using Parameters

- Declaring parameters
  - Place between parentheses after method name
  - Define type and name for each parameter
- Calling methods with parameters
  - Supply a value for each parameter

```
static void AParameterMethod(int i, string a)
{ ... }

AParameterMethod(1, “A string");
```

Passing Parameters

Pass Type	Direction
Pass by value	In
Pass by reference	In/out
Output parameters	Out

Pass by Value

- Default way for passing parameters:
  - Data copied
  - Parameter can be changed inside the method
  - Parameter changes have no effect on value outside the method
  - Parameter must be of the same type or compatible type

```
static int AddData(int a, int b)
{
       a = 3;
       return a + b;
}
static void Main( )
{
       int c = 1;
       int retD;
       retD = AddData(2, c);
       Console.WriteLine(c); // What value is displayed?
}
```

Pass by Reference

- Address logic rather than direct value in memory
- Using reference parameters
  - Must use the **ref** keyword in method declaration and call
  - Parameters must match the types passed into the method
  - Changes made in the method affect the calling code
  - Assign parameter value before calling the method

Pass by Reference Example

```
static int AddData(ref int a, ref int b)
{
      a = 2;
      return a + b;
}
static void Main( )
{
      int c = 1;
      int d = 2;
      int retValue = 0;
      retValue = AddData(ref c, ref d);
      Console.WriteLine(c); // What value is displayed?
}
```

Output Parameters

- Unique to C#
- Data is passed out but not in
- Syntax
  - Like **ref**, but values are not passed into the method
  - Use **out** keyword in method declaration and call

```
static void DataOut(out int i)
{
      i = 10;
}
int j;
OutDemo(out j);
Console.WriteLine(j);  // What value is j?
```

Guidelines for Passing Parameters

- Mechanisms
  - – Pass by value is most common
  - – Use return value normally for single values
  - – Use **ref** and/or **out** for multiple return values
  - – **Ref** should be used only if data is transferred both ways
- Efficiency
  - – Most efficient passing technique is by value

Using Recursive Methods

- A recursive method is a method that can call itself
- Useful for solving certain problems (Fibonnacci, tower of hanoi, etc.)

Overloaded Methods

- Methods that share the same name in a class
- Signature uniqueness determined by examing parameter list

```
class MethodOverloading
{
    static int AddData(int a, int b)
    {
        return a + b;
    }
    static int AddData(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        Console.WriteLine(AddData(1,2) + AddData(1,2,3));
    }
}
```

Method Signatures

- Method signatures must be unique within a class

Signature Item	No Effect on Signature
– Name of method – Parameter type – Parameter modifier	– Name of parameter – Return type of method

Using Overloaded Methods

- Useful situations for overloaded methods
  - Similar method action that require different parameters
  - Need to update functionality of a method
- Problems with overloaded methods
  - More complex to debug
  - Difficult to maintain

Property Usage

- Act like a field member
- Provide data validation capability
- A useful way to encapsulate information inside a class
- Concise syntax
- Flexibility

Using Accessors

- Properties provide field-like access
  - Use **get** accessor statements to provide read access
  - Use **set** accessor statements to provide write access

```
class UseProperty
{
    public string AName // Property
    {
        get { return holdName; }
        set { holdName = value; }
    }
    private string holdName; // Use a private field
}
```

Comparing Properties to Fields

- Properties are “logical fields”
  - The **get** accessor can return a computed value
- Similarities
  - Syntax for creation and use is the same
- Differences
  - Properties are not values; they have no address
  - Properties cannot be used as **ref** or **out** parameters to methods

Comparing Properties to Methods

- Similarities
  - – Both contain code to be executed
  - – Both can be used to hide implementation details
  - – Both can be virtual, abstract, or override
- Differences
  - – Syntactic – properties do not use parentheses
  - – Semantic – properties cannot be **void** or take arbitrary parameters

Property Types

- Read/write properties have both **get** and **set** accessors
- Read-only properties
  - Have **get** accessor only
  - Are not constants
- Write-only properties – very limited use
  - Have **set** accessor only
- Static properties
  - Apply to the class and can access only static data

Property Example

```
public class UseProperty
{
    public static string aProperty
    {
      get {
            return holdString;
      }
      set {
            holdString = value;
      }
    }
    ...
    private static string holdString = “”;
}
```

Practice: Create and Use a Class