Properties, Array, Encapsulation

Course Code: CSC 3115 Course Title: Object Oriented Programming 2

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	03	**Week No:**	03	**Semester:**	
Lecturer:					

Topics

1. Properties
2. OOP principles
3. Encapsulation
4. Array

Encapsulation

- Encapsulation
  - Encapsulate the inner details of implementation
  - Protect data

Encapsulation support in C#

- An object's field data and implementation details should not be directly accessed.
  - Define a pair of traditional accessor and mutator methods
    - Making use of access modifier (private, public)
  - Define a named property

```
public class Employee
{
        private string fullName;
        .......

        // Accessor
        public string GetFullName()
        {
              return fullName;
         }

        //Mutator
        public void SetFullName (string s)
        {
              fullName = s;
        }
}
```

enforcing encapsulation using traditional accessors and mutators

Class properties

- NET languages use properties to enforce encapsulation
- Stimulate public accessible data

```
static void Main()
{
        Employee p = new Employee();
        p.Name = "Tom";
        Console.WriteLine (p.Name);
}
```

Define a property

- A C# property is composed of a get block and set block
- value represents the implicit parameter used during a property assignment

```
public class Employee
{
        private string fullName;
        ...

        // Property for fullName
        public string Name
        {
                get { return fullName; }
                set { fullName = value; }
        }


}
```

Property visibility

```
// The get and set logic is both public,
// given the declaration of the property.
public string SocialSecurityNumber
{
        get { return empSSN; }
        set { empSSN = value; }
}
```

```
// Object users can only get the value,
// however derived types can set the value.
public string SocialSecurityNumber
{
        get { return empSSN; }
         protected set { empSSN = value; }
}
```

```
public string SocialSecurityNumber
{
        // Now as a write-only property

        set { empSSN = value; }
}
```

```
public string SocialSecurityNumber
{
        // Now as a  read-only property

        get { return empSSN; }
}
```

Static property

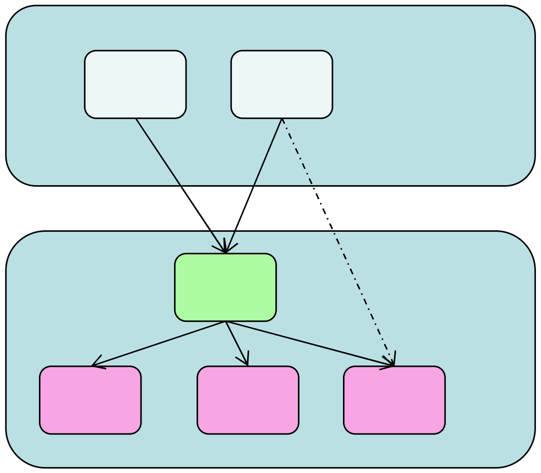
```
public class Employee
{
	private static string fullName;
	...

	// Property for fullName
	public static string Name
	{
		get { return fullName; }
		set { fullName = value; }
	}


}
```

Access Modifiers

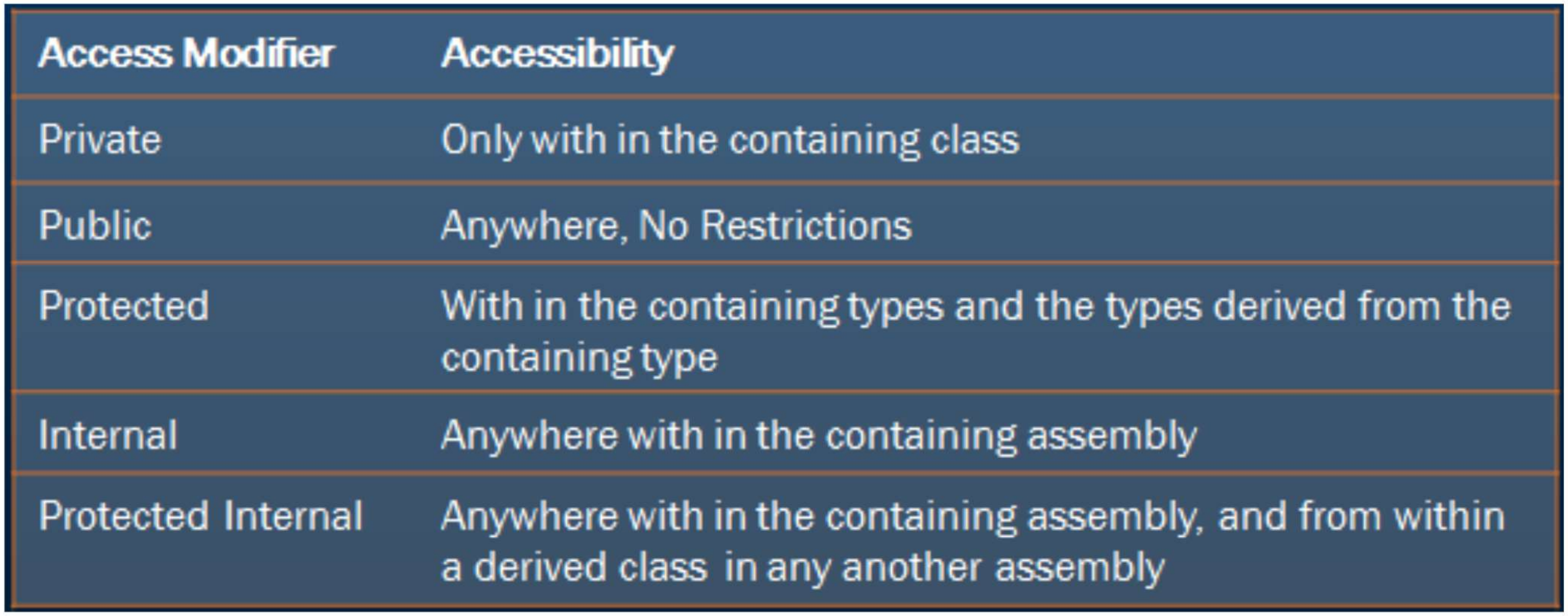
- Access Modifiers are used to define the visibility of a class property or method.
- There are times when you may not want other programs to see the properties or the methods of class. In such cases, C# gives the ability to put modifiers on class properties and methods.
- The class modifiers have the ability to restrict access so that other programs cannot see the properties or methods of a class.

Access Modifiers

- Access modifiers control the “visibility” of a type or member to other types and members.
- They are key to encapsulation...
- There are defaults if you do not specify an access modifier; providing more restrictive access



- pink is not visible to light blue but visible to green
- green is visible to light blue



Access Modifier	Accessibility
Private	Only with in the containing class
Public	Anywhere, No Restrictions
Protected	With in the containing types and the types derived from the containing type
Internal	Anywhere with in the containing assembly
Protected Internal	Anywhere with in the containing assembly, and from within a derived class in any another assembly

Access Modifiers

Access Modifier	Restrictions
public	• **Applicable** to types and members • No restrictions. **Accessible** to all...
private	• **Applicable** only to members • A private member is **accessible** to only other members in the same class • A private is the **default** access for members
protected	• **Applicable** only to members (and types that are members – defined in another type) • **Accessible** to other members in the same class and sub-class
internal	• **Applicable** to types and members • **Accessible** to other types and members in the same assembly • internal is **default** access to class
protected internal	• is a **union** of protected and internal

ACCESS MODIFIERS IN OOPS

Accessibility Matrix to Remember		*Same Assembly*		*Different Assembly*
Access Modifier Type	**Short Name**	**Same Class**	**Different Class**	**Any class(inherits previous assembly class)**
Protected	**P**	*Yes*	*No*	*Yes*
Internal	**I**	*Yes*	*Yes*	*No*
Protected Internal	**PI**	*Yes*	*Yes*	*Yes*

Method Parameter Modifiers

1. **Value (or default) parameter**: If a parameter is not attached with any modifier, then the parameter's value is passed to the method. This is also known as call-by-value and it is the default for any parameter.
2. **ref (reference) parameter:** If a parameter is attached with a ref modifier, then changes will be made in a method that affect the calling method. This is also known as call-by-reference.

✓ Whenever we want to allow changes to be made in a method, then we will go for call by ref.

Method Parameter Modifiers in C#

3. **out (output) parameter:** If a parameter is attached with an out modifier, then we can return a value to a calling method without using a return statement.
4. **params (parameters) parameter:** If a parameter is attached with a params modifier, then we can send multiple arguments as a single parameter.

✓Any method can have **only one params modifier** and

✓it should be the last parameter for the method.

METHOD PARAMETER MODIFIERS IN C#

The following are the differences between **output (out**) and **reference (ref)** parameters:

1. The **output (out)** parameters do not need to be initialized before use in a called method. Because it is assumed that the called method will provide the value for such a parameter.

2. The **reference (ref)** parameters must be initialized before sending to called method. Because we are passing a reference to an existing type and if we don't assign an initial value, it would be equivalent to working on a NULL pointer.

Arrays In C#

1. Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type.

2. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Declaring Arrays

- To declare an array in C#, the syntax would be

```
type[] arrayName = new typeName
[size];
```

Array Overview

An array has the following properties:

1. An array can be Single-Dimensional, Multidimensional or Jagged.

2. The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.

3. The default values of numeric array elements are set to zero, and reference elements are set to null.

Array Overview

An array has the following properties:

4. A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to null.
5. Arrays are zero indexed: an array with n elements is indexed from 0 to n-1.
6. Array elements can be of any type, including an array type.
7. Array types are reference types derived from the abstract base type Array.

ARRAYS AS OBJECTS

- In C#, arrays are actually objects, and not just addressable regions of contiguous memory as in C and C++.
- Array is the abstract base type of all array types. We can use the properties, and other class members, that Array has.

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

Array Overview

- This example uses the Rank property to display the number of dimensions of an array.

```
class TestArraysClass
{
        static void Main()
        { // Declare and initialize an array:
                int[,] theArray = new int[5, 10];
                System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
        }
}

// Output: The array has 2 dimensions
```

SINGLE-DIMENSIONAL ARRAYS

- You can declare a single-dimensional array of **five integers** as shown in the following example:

```
int[] array = new int[5];
```

- An array that stores string elements can be declared in the same way. For example:

```
string[] stringArray = new string[6];
```

Array Initialization

- It is possible to initialize an array upon declaration, in which case, the rank specifier is not needed because it is already supplied by the number of elements in the initialization list. For example:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

```
string[] weekDays = { "Sun", "Mon", "Tue", "Wed",
                      "Thu", "Fri", "Sat" };
```

Multidimensional Arrays

- Arrays can have more than one dimension.
- For example, the following declaration creates a two-dimensional array of four rows and two columns.

```
int[,] array = new int[4, 2];
```

- We also can initialize the array without specifying the rank.

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Jagged Arrays

- A jagged array is an array whose elements are arrays.
- The elements of a jagged array can be of different dimensions and sizes.
- A jagged array is sometimes called an "array of arrays.“
- The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

```
int[][] jaggedArray = new int[3][];
```

Jagged Arrays

- Before you can use jaggedArray, its elements must be initialized.
- We can initialize the elements like this:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

- It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size.

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

Jagged Arrays

- We can also initialize the array upon declaration like this:

```
int[][] jaggedArray2 = new int[][]
{     new int[] {1,3,5,7,9},
      new int[] {0,2,4,6},
      new int[] {11,22}
};
```

JAGGED ARRAYS

- We can access individual array elements like these examples:

```
// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray3[0][1] = 77;
// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray3[2][1] = 88;
```

Jagged Arrays

- The following is a declaration and initialization of a single-dimensional jagged array that contains three two-dimensional array elements of different sizes.

```
int[][,] jaggedArray4 = new int[3][,]
{
        new int[,] { {1,3}, {5,7} },
        new int[,] { {0,2}, {4,6}, {8,10} },
        new int[,] { {11,22}, {99,88}, {0,9} }
};
```

- We can access individual elements as shown in this example, which displays the value of the element [1,0] of the first array (value 5) :

```
System.Console.Write("{0}", jaggedArray4[0][1, 0]);
```

Using foreach with Arrays

- C# also provides the foreach statement.
- This statement provides a simple, clean way to iterate through the elements of an array.
- The foreach statement processes elements in the order returned by the array, which is usually from the 0th element to the last.

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };

foreach (int i in numbers)
{
       System.Console.Write("{0} ", i);
}
       // Output: 4 5 6 1 2 3 -2 -1 0
```

USING FOREACH WITH ARRAYS

- With multidimensional arrays, we can use the same method to iterate through the elements, for example:

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };
foreach (int i in numbers2D)
{
       System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

N.B. However, with multidimensional arrays, using a nested for loop gives you more control over the array elements.

PASSING ARRAYS AS ARGUMENTS

- Arrays can be passed as arguments to method parameters.
- Because arrays are reference types, the method can change the value of the elements.
- We can pass an initialized single-dimensional array to a method.

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

- The following code shows a partial implementation of the print method.

```
void PrintArray(int[] arr)
{
        // Method code.
}
```

Passing Multidimensional Arrays As Arguments

- We pass an initialized multidimensional array to a method in the same way that we pass a one-dimensional array.

```
int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);
```

- The following code shows a partial declaration of a print method that accepts a two-dimensional array as its argument.

```
void PrintArray(int[,] arr)
{
        // Method code.
}
```

Enumeration in C#

- The enum keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list.
- By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1.
- For example, in the following enumeration, Sat is 0, Sun is 1, Mon is 2, and so forth.

```
enum Day {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

- Enumerators can use initializers to override the default values

```
enum Day {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

ENUMERATION IN C#

- Enums are strongly typed constants.
- They are essentially unique types that allow you to assign symbolic names to integral values.
- In the C# tradition, they are strongly typed, meaning that an enum of one type may not be implicitly assigned to an enum of another type even though the underlying value of their members are the same.
- Enums lend themselves to more maintainable code because they are symbolic, allowing you to work with integral values, but using a meaningful name to do so.
- For example, what type of code would you rather work with – a set of values named North, South, East, and West or the set of integers 0, 1, 2, and 3 that mapped to the same values, respectively? Enums make working with strongly typed constants via symbolic names easy.

WHAT IS MAIN USE OF ENUMERATION?

- The main benefit of this is that constants can be referred to in a consistent, expressive and type safe way.

```
public class Employee
{            private string _Gender;
             public Employee  (string Gender)
              {
                   _ Gender = Gender;
             }

}
```

- But now we are relying upon users to enter just the right value for that string.

```
Employee employee = new Employee("Male");
```

WHAT IS MAIN USE OF ENUMERATION?

- Using enums, you can instead have:

```
public enum Gender { Male, Female}

public class Employee
{              private string _Gender;
               public Employee  (string Gender)
                {
                     _ Gender = Gender;
               }

}
```

- So now,

```
Employee employee = new Employee(Gender.Male);
```

WHAT IS MAIN USE OF ENUMERATION?

The following are advantages of using an enum instead of a numeric type:

- We can clearly specify for client code which values are valid for the variable.
- The main benefit of this is that constants can be referred to in a consistent, expressive and type safe way.
- Whenever there are situations where you are using a set of related numbers in a program, consider replacing those numbers with enums. It will make a program more readable and type safe.

Thank You

Books

- C# 4.0 The Complete Reference; Herbert Schildt; McGraw-Hill Osborne Media; 2010
- Head First C# by Andrew Stellman
- Fundamentals of Computer Programming with CSharp – Nakov v2013

References

MSDN Library; URL: http://msdn.microsoft.com/library

C# Language Specification; URL: http://download.microsoft.com/download/0/B/D/0BDA894F-2CCD-4C2C-B5A7-4EB1171962E5/CSharp%20Language%20Specixfication.doc

C# 4.0 The Complete Reference; Herbert Schildt; McGraw-Hill Osborne Media; 2010