31927 32998:
Application Development with .NET

Week-4 Lecture

C# Programming

Part-3

Outline

- set, get, and value Keywords
- “This” Keyword
- Polymorphism (Static)
- Method Overloading
- Constructor Overloading
- Operator Overloading
- Copy Constructors
- Structs
- File I/O
- Garbage Collection, Destructor

set and *get* Keywords

- **Properties** are members of class, structure of interface, which provides a flexible mechanism to read, write and compute/modify the values of private fields.

- Properties are special methods called *accessors*.

- set keyword is used to assign a new value to a private field
- get keyword is used to return the property value
- value keyword is used to define the values being assigned by the set accessor.

set and *get* Keywords

```
class SomeClass
{
    // Private field,
    // accessible only within the body of the class
    private int count;

    2 references
    public int Count
    {
        get // get defination
        {
            return count;
        }
        set // set defination
        {
            if (value > 0)
            {
                // use of Value
                count = value;
            }
            else
            {
                count = 0;
            }
        }
    }
}
```

Output

```
Value of X: 4
```

```
using System;
// Program to demonstrate
// set, get and value

namespace Week4ClassProgram
{
    0 references
    class AccessorDemo
    {
        2 references
        class SomeClass...

        0 references
        static void Main(string[] args)
        {
            SomeClass sc = new SomeClass();
            // Set the value of a private field
            sc.Count = 4;

            // Accessing a private field
            int getX = sc.Count;
            Console.WriteLine("Value of X: {0}", getX);

            Console.ReadKey();
        }
    }
}
```

”This” Keywords

- The “this” keyword refers to the current instance of the class
- Can be used as a modifier of the first parameter of an extension method.
- Use of this is implicit when calling any instance member, and it returns an instance of the object itself.

Reference: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/this

Example of “This”

```
class Box
{
    private int height, width, breadth;
    private int boxPaintCode = 0;

    0 references
    public Box()
    {
        height = width = breadth = boxPaintCode= 0;
    }
    // this used to qualify the fields
    // resolve ambiguity
    1 reference
    public Box(int height, int width, int breadth, int paintCode)
    {
        this.height = height;
        this.width = width;
        this.breadth = breadth;
        boxPaintCode = paintCode;
    }
    // Pass object as parameter
    1 reference
    public void Display()
    {
        Console.WriteLine("Box dimensions are:");
        Console.WriteLine("Height: {0}, Width: {1}", height, width);
        Console.WriteLine("Breadth: {0}, Color Code: {1},", breadth, boxPaintCode);
        Console.WriteLine("The box was {0} ", PaintBox.PaintIt(this));
    }
    1 reference
    public int BoxPaintCode
    {
        get { return boxPaintCode; }
    }
}
```

Output

```
Box dimensions are:
Height: 10, Width: 20
Breadth: 5, Color Code: 2,
The box was painted blue
```

```
using System;
// This program demonstrates the
// use of this keyword

namespace Week4ClassProgram
{
    5 references
    class Box...

    1 reference
    class PaintBox
    {
        1 reference
        public static string PaintIt(Box B1)
        {
            if (B1.BoxPaintCode == 1)
                return ("painted red");
            else
                return ("painted blue");
        }
    }

    0 references
    class thisDemo
    {
        0 references
        static void Main(string[] args)
        {
            Box box1 = new Box(10, 20, 5, 2); // Create a box
            box1.Display();
            Console.ReadKey();
        }
    }
}
```

Polymorphism (Static)

- Purpose: “Program in the general” instead of “Program in the specific”
- In static Polymorphism function/methods are linked to an object during compile time, also known as early/static binding.
- C# provides two techniques to implement static polymorphism:
  - Function overloading
  - Operator overloading

Reference: C# 2010 for programmers, Paul Deitel & Harvey Deitel

Constructor overloading

- Constructor Overloading is a technique to create multiple constructors with a different set of parameters and the different number of parameters.
- A class can be used in a different manner. The same class may behave different type based on constructors overloading.
- *Example*: With one object initialization, it may show simple message whereas, in the second initialization of an object with the different parameter, it may perform some mathematical calculation.

Method overloading

- Having multiple different definitions for the same method name in the same class.
- The definition of the method must differ from each other:
  - by the types of arguments and/or
  - the number of arguments in the argument list.
  - You cannot overload method declarations that differ only by return type.

```
public bool Search(int studentID)
public bool Search(string studentName)
public bool Search(string studentName, int studentID)
```

```
StudentDataBase studentDB = new StudentDataBase();
studentDB.Search(5);
studentDB.Search("George");
studentDB.Search("George", 5);
```

Example of Constructor & Method overloading

```
class StudentDataBase
{
    private int[] studentID;
    private string[] studentName;

    // Default constructor
    0 references
    public StudentDataBase()...

    // Overloaded constructor
    1 reference
    public StudentDataBase(string[] name, int[] ids)...
    // Method to search for a student ID
    1 reference
    public bool Search(int ID)
    {
        for (int loopVar = 0; loopVar < studentID.Length ; loopVar++)
        {
            if (studentID[loopVar] == ID)
                return true;
        }
        return false;
    }
    // Overload Method to search for a student Name
    1 reference
    public bool Search(string name)...

    // Overload Method to search for a student Name and ID
    1 reference
    public bool Search(string name, int ID)...

}
```

Output

```
Student ID 1002 found!
Student: George found!
Student: George with ID 1002 not found!
```

```
using System;

namespace Week4ClassProgram
{
    4 references
    class StudentDataBase...

    0 references
    class MethodOverloadingDemo
    {
        0 references
        static void Main(string[] args)
        {
            string[] studentName = { "George", "Barry", "Jack" };
            int[] ids = { 1001, 1002, 1003 };
            StudentDataBase studentDB = new StudentDataBase(studentName, ids);
            if (studentDB.Search(1002))
                Console.WriteLine("Student ID 1002 found!");
            else
                Console.WriteLine("Student ID 1002 not found!");

            if (studentDB.Search("George"))
                Console.WriteLine("Student: George found!");
            else
                Console.WriteLine("Student: George not found!");

            if(studentDB.Search("George", 1002))
                Console.WriteLine("Student: George with ID 1002 found!");
            else
                Console.WriteLine("Student: George with ID 1002 not found!");

            Console.ReadKey();
        }
    }
}
```

Operator overloading

- Redefine the built-in operators available in C# for specific task
- Enables programmer to use operators with user-defined types
- Overloaded operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined.
- An overloaded operator has a return type and a parameter list.

Example of Operator overloading

```
class Box
{
    private double length;   // Length of a box
    private double breadth;  // Breadth of a box
    private double height;   // Height of a box

    2 references | 0 changes | 0 authors, 0 changes
    public Box()...
    // Overloaded Box constructor
    2 references | 0 changes | 0 authors, 0 changes
    public Box(double lt, double bd, double ht)...

    // Overload + operator to add two Box objects.
    1 reference | 0 changes | 0 authors, 0 changes
    public static Box operator +(Box b1, Box b2)
    {
        Box box = new Box();
        box.length = b1.length + b2.length;
        box.breadth = b1.breadth + b2.breadth;
        box.height = b1.height + b2.height;
        return box;
    }
    // Display the Box information
    3 references | 0 changes | 0 authors, 0 changes
    public void Display()...
}
```

Output

```
Box 1 specification:
Length = 6
Height = 5
breadth = 7

Box 2 specification:
Length = 12
Height = 10
breadth = 13

Box 3 specification:
Length = 18
Height = 15
breadth = 20
```

```
using System;
// Program to demonstrate operator overloading

namespace Week4ClassProgram
{
    13 references | 0 changes | 0 authors, 0 changes
    class Box...
    0 references | 0 changes | 0 authors, 0 changes
    class OperatorOverloadingDemo
    {
        0 references | 0 changes | 0 authors, 0 changes
        static void Main(string[] args)
        {
            Box Box1 = new Box(6.0, 7.0, 5.0);   // Create Box1 of type Box
            Box Box2 = new Box(12.0, 13.0, 10.0);   // Create Box2 of type Box
            Box Box3 = new Box();   // Declare Box3 of type Box

            Console.WriteLine("Box 1 specification:");
            Box1.Display();

            Console.WriteLine("Box 2 specification:");
            Box2.Display();

            // Add two object as follows:
            Box3 = Box1 + Box2;

            Console.WriteLine("Box 3 specification:");
            Box3.Display();

            Console.ReadKey();
        }
    }
}
```

Which Operators can be Overloaded?

Operators & Description
+, -, !, ~, ++, -- → These unary operators take one operand and can be overloaded.
+, -, *, /, % → These binary operators take one operand and can be overloaded.
==, !=, <, >, <=, >= → The comparison operators can be overloaded.
&&, || → The conditional logical operators cannot be overloaded directly.
+=, -=, *=, /=, %= → The assignment operators cannot be overloaded.
=, ., ?:, ->, new, is, sizeof, typeof → These operators cannot be overloaded.

Copy Constructors

- C# doesn’t provide a copy constructor of object
- Copy constructor are similar to creating a clone of the an existing object
- An object is usually passed as an argument to the copy constructor, and the property values of the object are assigned to the new instance of that class.

Copy Constructors Example

```
class Box
{
    private int height, width, breadth;

    // Copy Costructor
    1 reference
    public Box(Box tempBox)
    {
        height = tempBox.height;
        width = tempBox.width;
        breadth = tempBox.breadth;
    }
    // Overloaded Constructor
    1 reference
    public Box(int height, int width, int breadth)
    {
        this.height = height;
        this.width = width;
        this.breadth = breadth;
    }
    // Function to display Box properties
    2 references
    public void Display()
    {
        Console.WriteLine("The Box properties are:");
        Console.WriteLine("Height: {0}, Width: {1}, Breadth: {2}", height, width, breadth);
    }
}
```

```
using System;
// The Program demonstrates
// Constructors

namespace Week4ClassProgram
{
    7 references
    class Box...

    0 references
    class CopyConstructorDemo
    {
        0 references
        static void Main(string[] args)
        {
            Box b1 = new Box(12, 10, 5);
            Box b2 = new Box(b1); // Copy constructor called

            b1.Display();
            b2.Display();

            Console.ReadKey();
        }
    }
}
```

Output

```
The Box properties are:
Height: 12, Width: 10, Breadth: 5
The Box properties are:
Height: 12, Width: 10, Breadth: 5
```

Structs

- The struct type is suitable for representing lightweight objects such as Point, Rectangle, and Colour. Although it is just as convenient to represent a point as a class with Auto-Implemented Properties, a struct might be more efficient in some scenarios.
- It is an error to define a default (parameter less) constructor for a struct.
- It is also an error to initialize an instance field in a struct body.
- new is not required to create a struct object!, however new can be used as well
- Structs are value type, classes are reference type.
- Doesn’t support inheritance.
- Structs cannot declare Default constructor or finalizer!

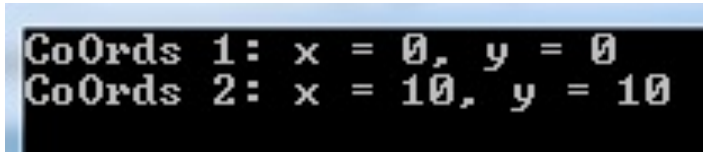
Reference : https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/using-structs

Example of Structs

```
public struct CoOrdinates
{
    public int x, y;

    1 reference | 0 changes | 0 authors, 0 changes
    public CoOrdinates(int xCord, int yCord)
    {
        x = xCord;
        y = yCord;
    }
}
```

Output




```
using System;
// Program to demonstrate struct

namespace Week4ClassProgram
{
        5 references | 0 changes | 0 authors, 0 changes
        public struct CoOrdinates...

        // Declare and initialize struct objects.
        0 references | 0 changes | 0 authors, 0 changes
        class StructDemo
        {
            0 references | 0 changes | 0 authors, 0 changes
            static void Main()
            {
                // Initialize:
                CoOrdinates coords1 = new CoOrdinates(); // Default constructor called
                // Parameterised constructor called
                CoOrdinates coords2 = new CoOrdinates(10, 10);

                // Display results:
                Console.Write("CoOrds 1: ");
                Console.WriteLine("x = {0}, y = {1}", coords1.x, coords1.y);

                Console.Write("CoOrds 2: ");
                Console.WriteLine("x = {0}, y = {1}", coords2.x, coords2.y);

                Console.ReadKey();
            }
        }
}
```

File I/O

- A **file** is a collection of data stored in a disk with a specific name and a directory path.
- When a file is opened for reading or writing, it becomes a **stream**.
- **Stream** is the sequence of bytes passing through the communication path. C# uses the concept of Stream to link your program to a physical device.
- There are two main streams:
  - the **input stream:** used for reading data from file
  - the **output stream**: used for writing into the file

Reference: https://www.tutorialspoint.com/csharp/csharp_file_io.htm

File I/O

- System.IO namespace will be required for reading/writing data from file.
- Commonly used class:

FileStream : Used to read/write/close a file

Syntax:

```
FileStream <object_name> = new FileStream( <file_name>, <FileMode Enumerator>,
                                           <FileAccess Enumerator>, <FileShare Enumerator>);
```

Example:

```
FileStream newFile = new FileStream("sample.txt", FileMode.Open, FileAccess.Read, FileShare.Read);
```

Reference: https://www.tutorialspoint.com/csharp/csharp_file_io.htm

File I/O

Enumerator	Member and Description
FileMode	- **Append**: Opens a file with cursor placed at the end of the file, creates a file if it doesn't exist. - **Create:** Create a new file - **Open**: Open an existing file - **OpenOrCreate**: Open or create a file. - **Truncate**: Open and truncate a file to zero bytes
FileAccess	- **Read** - **ReadWrite** - **Write**
FileShare	- **None** : No sharing of the current file - **Read** : Allows opening the file for reading - **ReadWrite**: Allows opening a file for reading and writing - **Write**: Allows opening a file for writing

Reference: https://www.tutorialspoint.com/csharp/csharp_file_io.htm

File I/O

- The File class: Provide static methods for creating/copying/moving/opening a single file

File : Common methods

- ReadAllText: Opens a text file, reads all the lines into String array and closes it.
- WriteAllText: Creates a new file and write the contents to it, closes it. It overwrites the existing file.
- AppendAllText: Appends a specified string to the file and create a file if it doesn't exists.

Reference: https://docs.microsoft.com/en-us/dotnet/api/system.io.file?view=netframework-4.7.2

Example of File I/O

Output

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1

Text in the file is
 I came here first!
After Appending:
Text in the file is
 I came here first!
I was Appended here!
```

```
using System;
using System.IO;
// Program to Demostrate File I/O

namespace Week4ClassProgram
{
    0 references | 0 changes | 0 authors, 0 changes
    class FileIODemo
    {
        0 references | 0 changes | 0 authors, 0 changes
        static void Main(string[] args)
        {
            //Read the file
            FileStream newFile = new FileStream("testNumeric.txt", FileMode.OpenOrCreate,
               FileAccess.ReadWrite);

            for (int loopVar = 1; loopVar <= 20; loopVar++)
            {
                newFile.WriteByte((byte)loopVar);
            }
            newFile.Position = 0;
            //Read the file content and display
            for (int loopVar = 0; loopVar <= 20; loopVar++)
            {
                Console.Write(newFile.ReadByte() + " ");
            }
            //Close the file
            newFile.Close();

            // Reading writing text files
            File.WriteAllText("testString.txt", "I came here first!");
            string fileText = File.ReadAllText("testString.txt");
            Console.WriteLine("\n\nText in the file is \n {0}", fileText);

            // Appending to text file
            Console.WriteLine("After Appending:");
            File.AppendAllText("testString.txt", "\nI was Appended here!");
            fileText = File.ReadAllText("testString.txt");
            Console.WriteLine("Text in the file is \n {0}", fileText);

            Console.ReadKey();
        }
    }
}
```

Garbage Collection

- In the common language runtime (CLR), the garbage collector serves as an automatic memory manager. It provides the following benefits:
  - Enables you to develop your application without having to free memory.
  - Allocates objects on the managed heap efficiently.
  - Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations. Managed objects automatically get clean content to start with, so their constructors do not have to initialize every data field.
  - Provides memory safety by making sure that an object cannot use the content of another object.

Collecting the Garbage

- Full Collections

  In a full collection we must stop the program execution and find all of the *roots* into the GC heap. These roots come in a variety of forms, but are most notably stack and global variables that point into the heap.

- Partial Collections

  Unfortunately, the full garbage collection is simply too expensive to do every time, so now it's appropriate to discuss how having generations in the collection helps us out.

Reference: https://msdn.microsoft.com/en-us/library/ms973837.aspx

Conditions for a garbage collection

- Garbage collection occurs when one of the following conditions is true: The system has low physical memory. This is detected by either the low memory notification from the OS or low memory indicated by the host.
- The memory that is used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.
- The GC.Collect() method is called. In almost all cases, you do not have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

Reference: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

What happens during a garbage collection

- A garbage collection has the following phases:
  - A marking phase that finds and creates a list of all live objects.
  - A relocating phase that updates the references to the objects that will be compacted.
  - A compacting phase that reclaims the space occupied by the dead objects and compacts the surviving objects. The compacting phase moves objects that have survived a garbage collection toward the older end of the segment.

Reference: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

Code example

- There are different ways to clean-up unmanaged resources:
  - Example : DB Connections, open files, etc.
  - Implement IDisposable interface and Dispose method
  - 'using' block is also used to clean unmanaged resources
- Destructor
  - Backup- in case Dispose() is not called
  - Calls Dispose()

Finalizers

- Used to destruct instances of classes.
- A class can have only one Finalizer (Destructor)
- Finalizer cannot be called explicitly, they are automatically called.
- Inheritance and overload not applicable.
- Only used with Classes!
- Destructors have same name as the class,

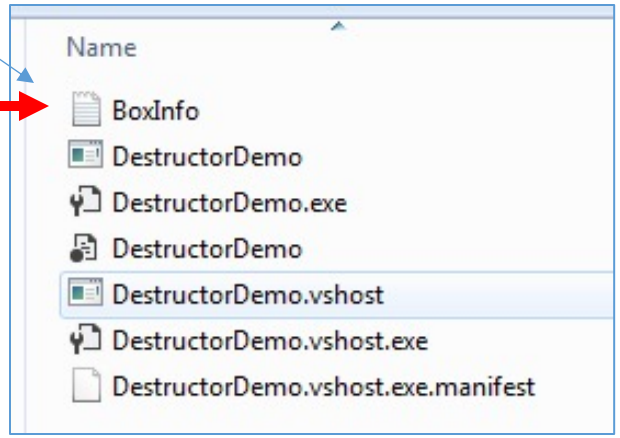
Syntax: ~<Class name>{ // Statements }

- called when object is removed from memory by garbage collector

Example of Destructor

```
// Destructor
0 references
~Box()
{
    // Save the Box information into a File
    string boxInfo = "Height:" + height.ToString() + " Width: " + width.ToString()
        + " Breadth:" + breadth.ToString();
    File.WriteAllText("BoxInfo.txt", boxInfo);
}
```

File Created and Box information saved



Found under bin/debug folder

```
using System;
using System.IO;
// Progam to show destructor demo

namespace Week4ClassProgram
{
    0 references
    class DestructorDemo
    {
        4 references
        class Box
        {
            // Private fields
            private int height, width, breadth;

            //Constructor
            1 reference
            public Box()
            {
                height = width = breadth = 10;
            }
            // Destructor
            0 references
            ~Box()
            {
                // Save the Box information into a File
                string boxInfo = "Height:" + height.ToString() + " Width: " + width.ToString()
                    + " Breadth:" + breadth.ToString();
                File.WriteAllText("BoxInfo.txt", boxInfo);
            }
        }
        0 references
        static void Main(string[] args)
        {
            Box b1 = new Box();

            Console.ReadKey();
        }
    }
}
```