Classes and Objects - Part II

`struct` – comparison to `class`

- Similar to **`c++ struct`**
- Syntax similar to C# class
- Recommended to be used for smaller data structure.
- No support for inheritance or referential identity
- struct are value types and not reference types. Pass by value, copy by value. **`null`** value cannot be assigned.
- The advantage of value type over reference type is that there are fewer objects created in heap → less work for garbage collection modules.

Enumeration

- Like structs, enumerations are also value types.
- Enumeration is used to define a set of values its instances can be assigned to.
- The values are represented in the form of strings
- However internally enumeration is an integer type in .NET.

Example

```
using System;
public enum Color{
Red,
Green,
Blue}

class EnumExample{
public static int Main(){
Color c=Color.Red;
Console.WriteLine(c);
return 0;
}
}
```

You can also assign a value

```
Red=1,
Green=2
```

Etc.

Displays Red

Behind the scenes

- enums in C# is are instantiated as structs derived from the base class, **`System.Enum.`**
- Try invoking
  - **`Console.WriteLine(c.ToString());`**
- Parse method

Parse method

- To get the value of an enum from a string

```
Color c1

  = (Color)

   Enum.Parse(typeof(Color),

  "Blue", true);

Console.WriteLine((int)c1);
```

What does this print?

Custom Indexer

- An array elements can be accessed using [] operator.
- C# also provides a capability to use [] operators on any user-defined classes (or struct)
- For instance,
  - `class FlowerVase{….}`

  -
    ```
    FlowerVase f= new FlowerVase()
    f[0]= new Flower();
    f[1]= new Flower();
    ```
- But for this we need to create something called a custom indexer.

Syntax

```
public return-type this[int index-
  position]{
get { return some-value}
set { some-value= value}
}
```

- For the whole thing to make sense the class with the indexer facility will invariably have an array or collection as its instance member .

Example-Custom Indexer

```
class Flower{
string name;
string color;
public Flower(string name, string
  color){
this.color=color;
this.name=name;
}
public void display(){
System.Console.WriteLine(name + " in
  color "+ color);
}}
```

```
class FlowerVase{
Flower[] flowers= new Flower[10];

public Flower this[int pos]{
get{ return flowers[pos];}
set{ flowers[pos]=value;}
}

public static void Main(){
FlowerVase fv= new FlowerVase();
fv[0]=new Flower("Rose","Red");
fv[1]=new Flower("Lilly","White");
fv[2]=new Flower("Rose","Yellow");
for(int i=0;i<3;i++)
fv[i].display();            }}
```

Custom Indexer

Overloading methods

- Methods with the same name but different signature are called overloaded methods.
- Method overloading is used for methods that do semantically the same thing.
- Methods may differ in terms of
  - Number of parameters
  - Types of parameter
  - Order of parameter
- When the method is called, compiler determines the best match based on the number, type and the order of the arguments of the method call statement.
- Note that if two methods are identical except for their return types or access specifiers then the methods are not overloaded.

Overload1.cs

```
using System;
class Overload1{

static void add(int a, int b){
int c=0;
c=a+b;
Console.WriteLine(c);
}

static void add(int a, int b, int c){
int x=0;
x=a+b+c;
Console.WriteLine(x);}
static void add(double a, double b){
double c=0;
c=a+b;
Console.WriteLine(c);
}

public static void Main(){
add(1,2);
add(1,2,3);
add(1.2,1.3);
}
}
```

Output:

```
3
6
2.5
```

Activity

- Try adding the method given below to the previous code and compile.

```
static int add(int a,
int b){
int c=0;
c=a+b;
Console.WriteLine(c);
return c;
}
```

Overloaded methods and Parameter modifiers

- Assume a method
  - `static int myMethod(int a, int b)`
- Candidates overloaded methods for the above method could be
  - `static int myMethod(int a,ref int b)`
  - `static int myMethod(int a,out int b)`
  - `static int myMethod(int a,params int[] b)`
- But following two methods are not overloaded
  - `static int myMethod(int a,params int[] b)`
  - `static int myMethod(int a, int[] b)`

Overloading operators

- Operator overloading is like function overloading except that instead of overloading ordinary function names we overload operators.
- For instance the result of a+b is different based on the type of a and b.

$$\begin{tikzcd}
a & + \arrow[rrrd, bend left] & b & & \\
1 & & 2 \arrow[r] & 3 & \text{Overloaded meaning for } + \\
\text{“xy”} & & \text{“z”} \arrow[r] & \text{“xyz”} &
\end{tikzcd}$$

Syntax

- **`public static`** ***`return-type`*** **`operator`** ***`operator(`*** ***`parameter-list`*** **`)`**
- Should be a static method
- Has **`operator`** keyword
- Operator can be anything except those listed
  - **`[]`** **`()`** compound operators like (**`+=`**, **`*=`** etc).
- Will have at least one parameters
- Should have a return type

Example: + operator overloaded for matrix

```
using System;
class Matrix{
public const int DIMSIZE = 2;
private byte[,] matrix = new byte[DIMSIZE,
  DIMSIZE];
```

Custom indexer for multidimensional array

```
public byte this[int x, int y]     {
        get { return matrix[x, y]; }
        set { matrix[x, y] = value; }    }

public static Matrix operator +(Matrix mat1,
  Matrix mat2) {
    Matrix newMatrix = new Matrix();
    for (int x=0; x < DIMSIZE; x++)
    for (int y=0; y < DIMSIZE; y++)
newMatrix[x, y] = (byte)(mat1[x, y]+mat2[x,y]);
 return newMatrix;
    }}
```

Why are we casting here ?

```
class MatrixTest{
  static Random rand = new Random();
    static void Main() {
     Matrix mat1 = new Matrix();
     Matrix mat2 = new Matrix();

        InitMatrix(mat1);
        InitMatrix(mat2);

        Console.WriteLine("Matrix 1: ");
        PrintMatrix(mat1);
        Console.WriteLine("Matrix 2: ");
        PrintMatrix(mat2);
```

System.Random class is used here to initialize matrix with random values

```
   Matrix mat3 = mat1 + mat2;
   Console.WriteLine("Matrix 1 + Matrix 2 = ");
   PrintMatrix(mat3);
 }
 public static void InitMatrix(Matrix mat) {
 for (int x=0; x < Matrix.DIMSIZE; x++)
 for (int y=0; y < Matrix.DIMSIZE; y++)
   mat[x, y] = (byte)(rand.NextDouble()*255);
 }
 public static void PrintMatrix(Matrix mat)
 for (int x=0; x < Matrix.DIMSIZE; x++){
 for (int y=0; y < Matrix.DIMSIZE; y++){
 Console.Write("{0:d3}  ",mat[x, y]); }
 Console.WriteLine();
 }
 Console.WriteLine();

  }}
```

initializing matrix with random values

Try

- What will happen if you were to add a statement like the one below in the previous program?

```
Matrix mat4=mat1+mat2+mat3;
```

Some automatic and some compulsory implementations

- C# enforces certain rules when you overload operators.
  - must implement matching operators. For example, overloading == will require overloading != also. The same goes for <= and >=
  - implementing an operator implies its compound operator also works!
    - For Matrix example, mat2+=mat1 also works!

Overloading pre and post increment

- Unlike C++, C# handles the pre & post Increment Operators by itself. Hence, we don’t need to overload them separately.

```
using System;
struct Box{
private int width,height;
public Box(int x,int y){
width=x;  height=y;}

public static Box operator --(Box b1){
b1.width--;
b1.height--;
return b1;
}
public static Box operator ++(Box b1){
b1.width++;
b1.height++;
return b1;
}
public void display(){
Console.WriteLine("width={0}
height={1}",width,height);}

static void Main(){
Box b1= new Box(10,20);
Box b2= new Box(10,20);
Box b3= --b1;
Box b4=b2++;
b1.display();
b2.display();
b3.display();
b4.display();}}
```

```
Output:
width=9 height=19
width=11 height=21
width=9 height=19
width=10 height=20
```

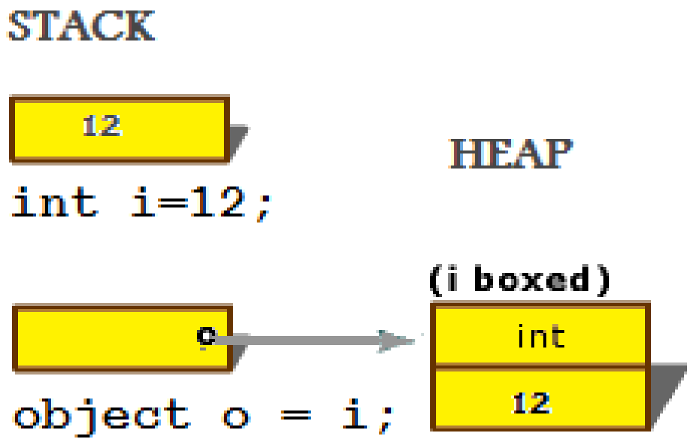
Note that the same thing does not work if the struct is changed to class. Why?

Boxing

- Boxing and unboxing allows a value-type to be converted to and from type object(`object` or `ValueType` or to any interface implemented by the value type).
- Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an object.
- Converting a value type to reference type is called Boxing. Unboxing is an explicit operation.
- C# provides a “unified type system”. All types—including value types—inherit from the type `object`. It is possible to call object methods on any value, even values of “primitive” types such as `int`.
- For instance - `Console.WriteLine(3.ToString());`

Example- boxing and un-boxing

```
class Test {
static void Main() {
int i = 1;
object o = i; // boxing→ implicit
 int j = (int) o; //
} }
```



Memory

- Original value type and the boxed object use separate memory locations, and therefore can store different values.
- For instance,

```
int i = 12;
object o = i;
i = 14;
```

Value of i is 14, but o is 12.

Un-boxing conversions

- Legal code

```
int i = 12;
object o = i;
int j = (int) o;
```

- Invalid cast

```
int i = 12;
object o = i;
int j = (short) o;
```

Error Reported at Runtime:

```
Unhandled Exception: System.InvalidCastException:
Specified cast is not valid.
   at CastTest.Main()
```

Garbage collection

- The .NET Framework's garbage collector manages the allocation and release of memory for the .NET application.
- Each time the new operator is used to create an object, the runtime allocates memory for the object from the managed heap.
- When the garbage collector performs a collection, it checks for objects in the managed heap that are no longer being used by the application and performs the necessary operations to reclaim their memory.

Destructors

- Destructors can be defined for a class so that cleanup code can be written.
- This is written so that they can be executed just before the object is garbage collected.
- A class can only have one destructor.
- Destructors cannot be inherited or overloaded.
- Destructors cannot be called. They are invoked automatically.
- When an object is eligible for destruction, garbage collector calls the destructor (if any) and reclaims the memory used to store the object. Destructors are also called when the program exits.
- The programmer has no control over when the destructor is called because this is determined by the garbage collector.

Example-Destructor

```
using System;
class Employee{

~ Employee() {
  Console.WriteLine("Employee
  destroyed");
}

public static void Main(){
Employee e1= new Employee();
e1=null;
}}
```

A destructor does not take modifiers or have parameters.

Forcing a Garbage Collection

- The garbage collection `GC` class provides the `GC.Collect` method, which can be used to give your application some direct control over the garbage collector.

IDisposable Interface

- An alternative to using a destructor is using **`System.IDisposable`** interface.
- The System.IDisposable interface defines a method of releasing unmanaged resources which have been allocated.
- Unmanaged resources are object created and-or manipulated outside the control of the Common Language Runtime (CLR). Examples: windows file handles and ODBC database connections.
- **`public interface IDisposable`** with a method
  - **`void Dispose ()`**

```
class MyDisposable : IDisposable {
FileStream myStream;

public MyDisposable (string filePath)
{ myStream = new FileStream
(filePath, FileAccess.ReadWrite);

}

public void Dispose() {
myStream.Dispose();

GC.SuppressFinalize(myStream);

}}
```

remove myStream from the GC finalize queue

Note that **`MyDisposable`** class can be called only with-in

- a try-catch-finally statement or
- a using statement.

```
using(MyDisposable d = new
MyDisposable ("File.txt"))
{
…
}
```

d out of scope here as it has been disposed