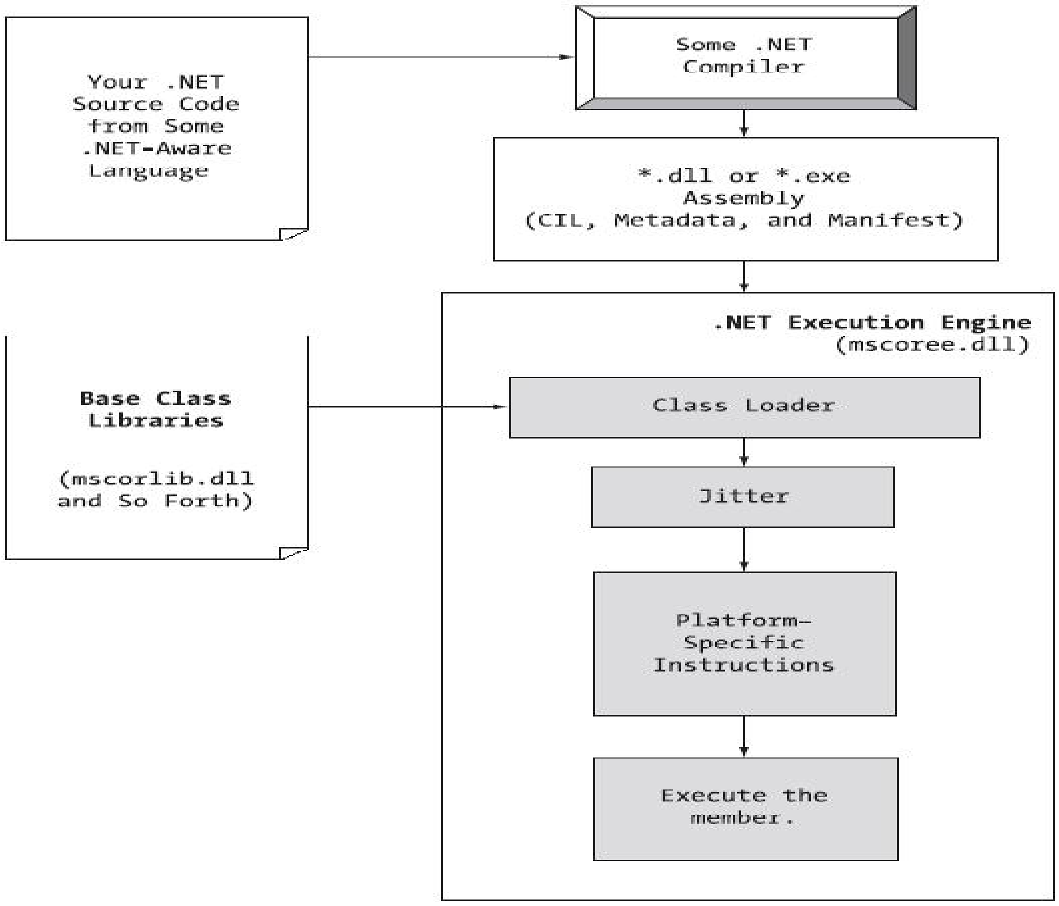
1. With neat diagram describe CRT?


Your .NET Source Code from Some .NET-Aware Language
Some .NET Compiler
*.dll or *.exe Assembly (CIL, Metadata, and Manifest)
.NET Execution Engine (mscoree.dll)
Base Class Libraries (mscorlib.dll and So Forth)
Class Loader
Jitter
Platform-Specific Instructions
Execute the member.

The crux of the CLR is physically represented by a library named mscoree.dll (aka the Common Object Runtime Execution Engine). When an assembly is referenced for use, mscoree.dll is loaded automatically, which in turn loads the required assembly into memory. The runtime engine is responsible for a number of tasks.

In addition to loading your custom assemblies and creating your custom types, the CLR will also interact with the types contained within the .NET base class libraries when required. Although the entire base class library has been broken into a number of discrete assemblies, the key assembly is mscorlib.dll. mscorlib.dll contains a large number of core types that encapsulate a wide variety of common programming tasks as well as the core data types used by all .NET languages.

2. Write a note on Command Line Debugger?

b[reak] :Set or display current breakpoints.
del[ete] :Remove one or more breakpoints.

ex[it] :Exit the debugger.

g[o] :Continue debugging the current process until hitting next breakpoint.

o[ut] :Step out of the current function.

p[rint] :Print all loaded variables (local, arguments, etc.).

si : Step into the next line.

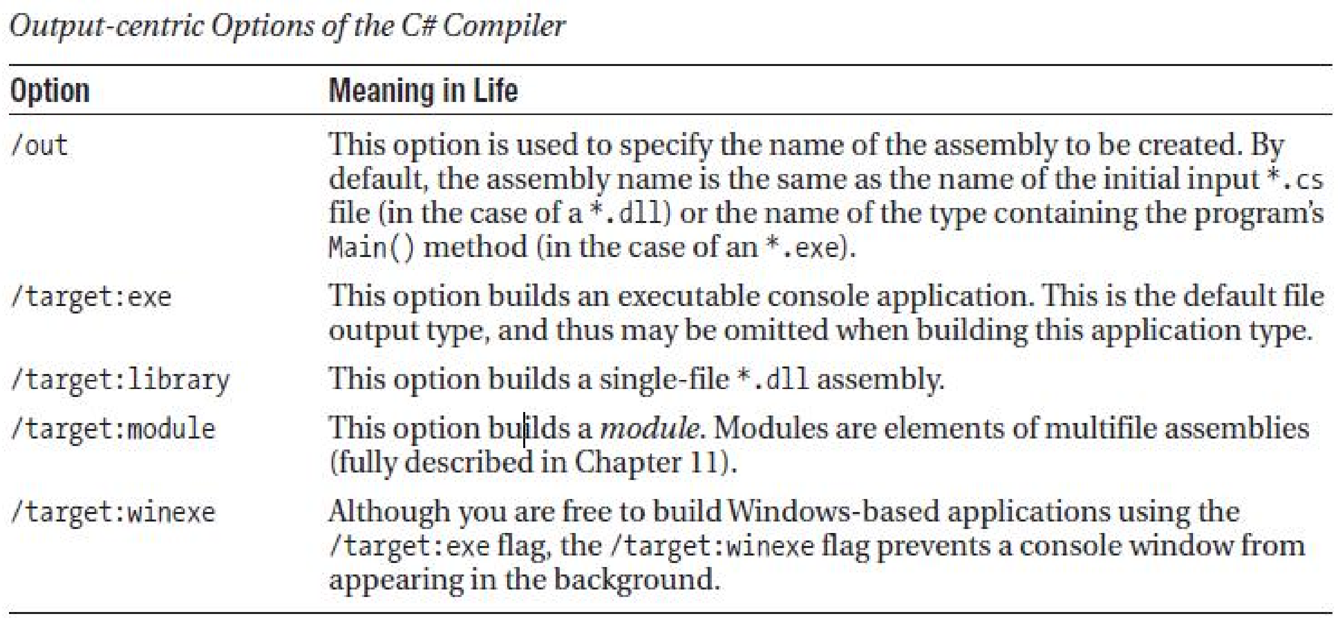
so :Step over the next line.

3. Discuss, how to build C# application using cse.exe?

To build a simple single file assembly named TestApp.exe using the C# command-line compiler and Notepad. First, you need some source code. Open Notepad and enter the following:

```
// A simple C# application.
using System;
class TestApp
{
public static void Main()
{
Console.WriteLine("Testing! 1, 2, 3");
}
}
```

Once we have finished, save the file in a convenient location (e.g., C:\CscExample) as TestApp.cs. Each possibility is represented by a specific flag passed into csc.exe as a command-line parameter see below table which are the core options of the C# compiler.


Output-centric Options of the C# Compiler

Option	Meaning in Life
`/out`	This option is used to specify the name of the assembly to be created. By default, the assembly name is the same as the name of the initial input `*.cs` file (in the case of a `*.dll`) or the name of the type containing the program's `Main()` method (in the case of an `*.exe`).
`/target:exe`	This option builds an executable console application. This is the default file output type, and thus may be omitted when building this application type.
`/target:library`	This option builds a single-file `*.dll` assembly.
`/target:module`	This option builds a *module*. Modules are elements of multifile assemblies (fully described in Chapter 11).
`/target:winexe`	Although you are free to build Windows-based applications using the `/target:exe` flag, the `/target:winexe` flag prevents a console window from appearing in the background.

To compile TestApp.cs into a console application named TestApp.exe enter

csc /**target:exe** TestApp.cs

C# compiler flags support an abbreviated version, such as **/t** rather than **/target**

csc /**t:exe** TestApp.cs

default output used by the C# compiler, so compile TestApp.cs simply by typing

csc TestApp.cs

TestApp.exe can now be run from the command line a shows o/p as;

C:\TestApp

Testing! 1, 2, 3

Referencing External Assemblies

- To compile an application that makes use of types defined in a separate .NET assembly. Reference to the System.Console type mscorlib.dll is *automatically referenced* during the compilation process.
- To illustrate the process of referencing external assemblies the TestApp application to display windows Forms message box.
- At the command line, you must inform csc.exe which assembly contains the "used" namespaces.
- Given that you have made use of the MessageBox class, you must specify the **System.Windows.Forms.dll** assembly using the /reference flag (which can be abbreviated to /r):

  csc /**r:System.Windows.Forms.dll** testapp.cs

Compiling Multiple Source Files with csc.exe

Most projects are composed of multiple *.cs files to keep code base a bit more flexible. Assume you have class contained in a new file named HelloMsg.cs:

// The HelloMessage class

```
using System;
using System.Windows.Forms;
class HelloMessage
{
public void Speak(){
MessageBox.Show("Hello...");
}
 }
```

Now, create TestApp.cs file & write below code

```
using System;
class TestApp
{
public static void Main()
{
Console.WriteLine("Testing! 1, 2, 3");
HelloMessage h = new HelloMessage();
h.Speak();
}
}
```

You can compile your C# files by listing each input file explicitly:

csc /r:System.Windows.Forms.dll **testapp.cs hellomsg.cs**

As an alternative, csc /r:System.Windows.Forms.dll ***.cs**

4. With program demonstrate the concept of passing reference types by value and by reference?

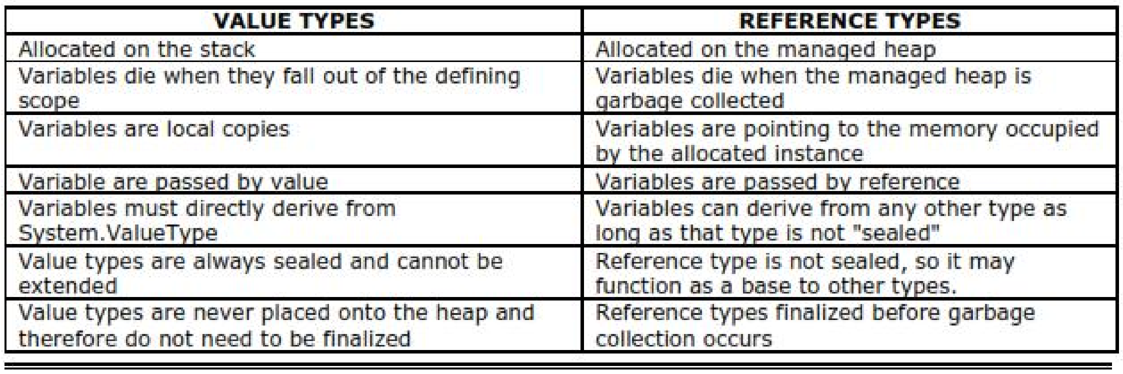
```
class A
{
   public int x;
   public A(int z)
   {
      x = z;
   }
}
   class Program
   {

      private static void passbyreferance(ref A a1)
      {
         a1.x = 50;
```

```
        }

        private static void passbyvalue(A a1)
        {
            a1.x = 50;
        }
        static void Main(string[] args)
        {
            A a1 = new A(10);
            Console.WriteLine("Pass by value");
            Console.WriteLine("Before");
            Console.WriteLine("{0}",a1.x);
            passbyvalue(a1);
            Console.WriteLine("after");
            Console.WriteLine("{0}", a1.x);

            Console.WriteLine("Pass by referance");
            Console.WriteLine("Before");
            Console.WriteLine("{0}", a1.x);
            passbyreferance(ref a1);
            Console.WriteLine("after");
            Console.WriteLine("{0}", a1.x);

            Console.ReadKey();
    }
      }
```

Output:

```
Pass by value
Before
10
After
50

Pass by reference
Before
50
After
50
```

5. **List the differences between value type and reference type? Write a program to illustrate value type containing reference type?**

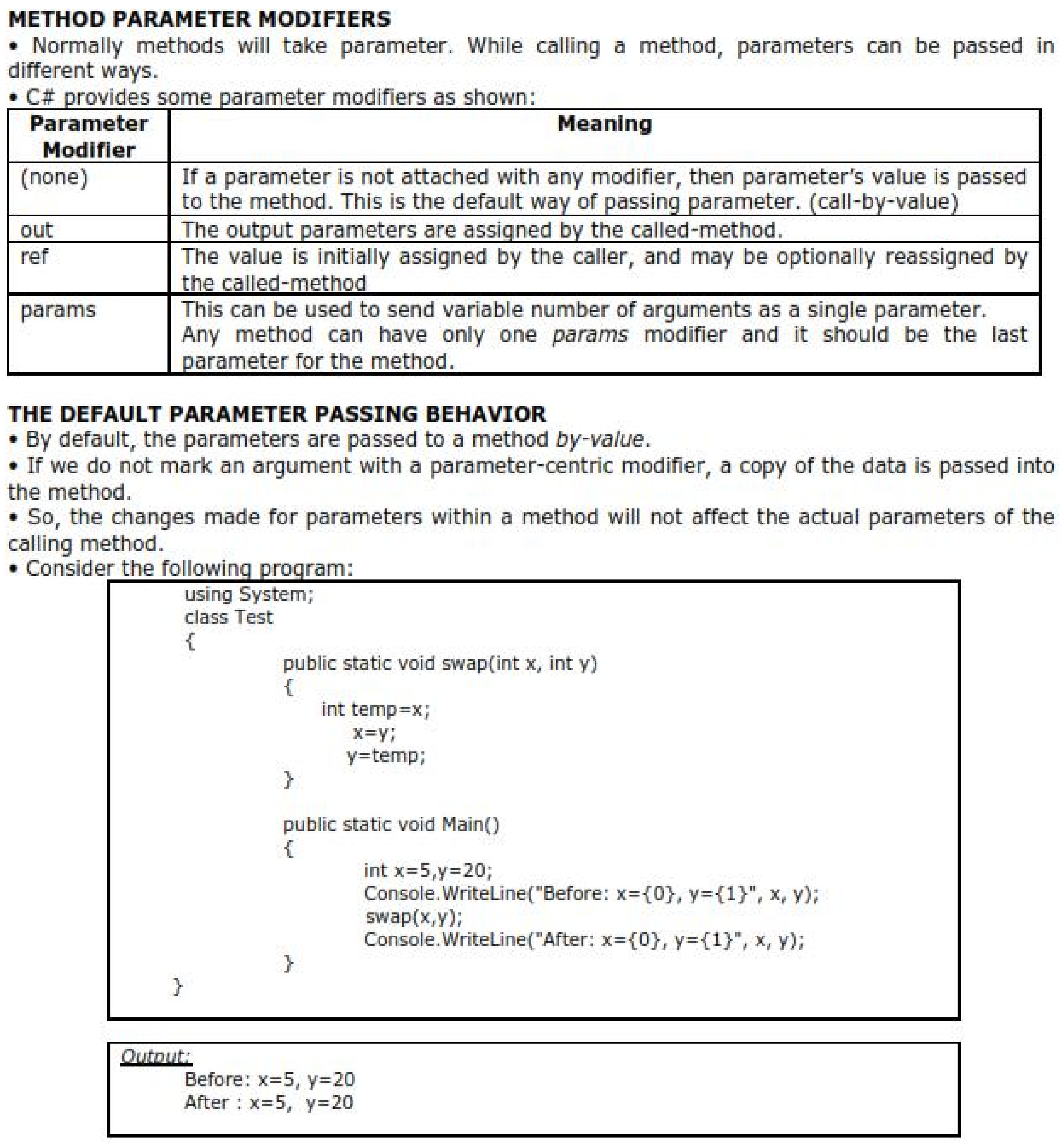


VALUE TYPES	REFERENCE TYPES
Allocated on the stack	Allocated on the managed heap
Variables die when they fall out of the defining scope	Variables die when the managed heap is garbage collected
Variables are local copies	Variables are pointing to the memory occupied by the allocated instance
Variable are passed by value	Variables are passed by reference
Variables must directly derive from System.ValueType	Variables can derive from any other type as long as that type is not "sealed"
Value types are always sealed and cannot be extended	Reference type is not sealed, so it may function as a base to other types.
Value types are never placed onto the heap and therefore do not need to be finalized	Reference types finalized before garbage collection occurs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
class A
{
    public int x;


    public A(int z)
    {
        x = z;
    }
}

struct B
{
    public int y;
    public A a1;
    public B(int z)
    {
        a1 = new A(z);
        y = 30;
    }
}

    class Program
    {
        static void Main(string[] args)
        {
            B b1 = new B(40);
            B b2 = b1;
            Console.WriteLine("before making any change");
            Console.WriteLine("{0} {1}", b1.y,b1.a1.x);
            Console.WriteLine("{0} {1}", b2.y, b2.a1.x);

            b2.a1.x = 60;
            b2.y = 60;

            Console.WriteLine("after making change");
            Console.WriteLine("{0} {1}", b1.y, b1.a1.x);
            Console.WriteLine("{0} {1}", b2.y, b2.a1.x);
            Console.ReadKey();

        }
    }
```

Output:
```
Before making any change
30 40
30 40

After making change
30 60
60 60
```

6. Illustrate the use of method parameter modifiers with an example?



METHOD PARAMETER MODIFIERS

• Normally methods will take parameter. While calling a method, parameters can be passed in different ways.

• C# provides some parameter modifiers as shown:

Parameter Modifier	Meaning
(none)	If a parameter is not attached with any modifier, then parameter's value is passed to the method. This is the default way of passing parameter. (call-by-value)
out	The output parameters are assigned by the called-method.
ref	The value is initially assigned by the caller, and may be optionally reassigned by the called-method
params	This can be used to send variable number of arguments as a single parameter. Any method can have only one *params* modifier and it should be the last parameter for the method.

THE DEFAULT PARAMETER PASSING BEHAVIOR

• By default, the parameters are passed to a method *by-value*.

• If we do not mark an argument with a parameter-centric modifier, a copy of the data is passed into the method.

• So, the changes made for parameters within a method will not affect the actual parameters of the calling method.

• Consider the following program:

```
using System;
class Test
{
        public static void swap(int x, int y)
        {
            int temp=x;
               x=y;
              y=temp;
        }

        public static void Main()
        {
              int x=5,y=20;
              Console.WriteLine("Before: x={0}, y={1}", x, y);
              swap(x,y);
              Console.WriteLine("After: x={0}, y={1}", x, y);
        }
}
```

Output:
```
Before: x=5, y=20
After : x=5,  y=20
```

out KEYWORD
• Output parameters are assigned by the called-method.
• In some of the methods, we need to return a value to a calling-method. Instead of using *return* statement, C# provides a modifier for a parameter as *out*.
• Consider the following program:

```
using System;
class Test
{
        public static void add(int x, int y, out int z)
        {
                z=x+y;
        }

        public static void Main()
        {
                int x=5,y=20, z;
                add(x, y, out z);
                Console.WriteLine("z={0}", z);
        }
}
```

Output:
```
z=25
```

• Useful purpose of out: It allows the caller to obtain multiple return values from a single method-invocation.
• Consider the following program:

```
using System;
class Test
{
        public static void MyFun(out int x, out string y, out bool z)
        {
                x=5;
                y="Hello, how are you?";
                z=true;
        }

        public static void Main()
        {
                int a;
                string str;
                bool b;

                MyFun(out a, out str, out b);
                Console.WriteLine("integer={0} ", a);
                Console.WriteLine("string={0}", str);
                Console.WriteLine("boolean={0} ", b);

        }
}
```

Output:
```
integer=5,
string=Hello, how are you?
boolean=true
```

ref KEYWORD

• The value is assigned by the caller but may be reassigned within the scope of the method-call.
• These are necessary when we wish to allow a method to operate on (and usually change the values of) various data points declared in the caller's scope.
• Differences between output and reference parameters:
  → The *output* parameters do not need to be initialized before sending to called-method. Because it is assumed that the called-method will fill the value for such parameter.
  → The *reference* 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 NULL pointer.
• Consider the following program:

```
using System;
class Test
{
        public static void MyFun(ref string s)
        {
                s=s.ToUpper();
        }

        public static void Main()
        {
                string s="hello";
                Console.WriteLine("Before:{0}",s);
                MyFun(ref s);
                Console.WriteLine("After:{0}",s);
        }

}
```

Output:
```
        Before: hello
        After: HELLO
```

• From the above example, we can observe that for *params* parameter, we can pass an array or individual elements.
• We can use *params* even when the parameters to be passed are of different types.
• Consider the following program:

```
using System;
class Test
{
        public static void MyFun(params object[] arr)
        {
                for(int i=0; i<arr.Length; i++)
                {
                        if(arr[i] is Int32)
                            Console.WriteLine("{0} is an integer", arr[i]);
                        else if(arr[i] is string)
                                Console.WriteLine("{0} is a string", arr[i]);
                            else if(arr[i] is bool)
                                    Console.WriteLine("{0} is a boolean",arr[i]);
                }
        }

        public static void Main()
        {
                int x=5;
                string s="hello";
                bool b=true;

                MyFun(b, x, s);
        }
}
```

Output:
```
        True is a Boolean
         5 is an integer
         hello is a string
```

7. List the functions associated with System. Object class and override any two methods?

```
Namespace System
{

   Public class Object
   {

      public Object();
      public virtual bool Equals(object obj);
      public static bool Equals(object objA, object objB);
      public virtual int GetHashCode();
      public Type GetType();
      protected object MemberwiseClone();
      public static bool ReferenceEquals(object objA, object objB)
      public virtual string ToString();
   }
}

Class person
   {
      string name;
      int id;


      public person(string p, int p_2)
      {
         // TODO: Complete member initialization
         name = p;
         id = p_2;
      }

      public override bool Equals(object obj)
      {
         person p = (person) obj;
         if ((this.id == p.id) && (this.name==p.name))
            return true;
         else
            return false;
      }

      public override int GetHashCode()
      {
         return id;
      }
   }

   class Program
   {
      static void Main(string[] args)
      {
        person p1 = new person("ram",10);
        person p2 = new person("shyam", 10);
        Console.WriteLine("{0} {1}",p1.Equals(p2),p1.GetHashCode());
    }
}
```