C#

Notes

With Code Examples

From C# Language Specification Version 1.2

By Haris Buchal

Introduction:

- C# is a strong typed language
- C# is completely Object Oriented
- .NET Framework v1.x is required to run programs written in C#
- Microsoft Visual Studio's version of C# is called Visual C#
- csc is a C# compiler
- Code should be written using UTF-8 character format with any file name (not necessarily .cs)
- Compilation Steps:
  File → Unicode Characters → Lexical Analysis → Syntactic Analysis → IL
- C# compiler can generate two types of files (also known as assemblies), .exe (or C# programs), .dll (or C# libraries)
- C# Entry Points:
  - `static void Main() { }`
  - `int void Main() { return 0; }`
  - `static void Main(string[] args) { }`
  - `int void Main(string[] args { return args != null ? args.Length : 0; }`
- **Example:**
  - Creating a C# Program: Save the following code in ANY file

    ```
    class Test {
    static void Main(){}
    }
    ```

    Open command-prompt and run

    ```
    csc filename i.e. csc Tests.cs
    ```

  - Compiling C# library: Save the following code in ANY file

    ```
    class Test {
    }
    ```

    Open command-prompt and run

    ```
    csc /target:library filename
    i.e. csc /target:library Tests.cs
    ```

C# Memory Management:

- Automatic Memory Management: System.GC.Collect(), System.GC.SupressFinalize()

- Types are contained in Application Domain (instances)
- Destructors are called by GC in deterministically
- **Example:**

```
using System;
public class ForceGC {

   static object myObject =
     new ForceGC(new string(new char[] {'A','B','C'}));

   static void Main() {
     GC.SuppressFinalize(myObject);
     //GC.Collect();
   }

   internal ForceGC(string s)
   {
      Console.WriteLine("myObject has been created with: " + s);
   }

  ~ForceGC()
  {
      Console.WriteLine(myObject + " will be destroyed now.");
  }
}
```

Literals In C#:

- Integers Suffix: Any combination of ULul→ 4 x 3 = 12, Ff/Dd/mM
- Hex Prefix: `0x/0X`
- Strings:
  - Regular/Verbatim, `"hello","\"hello\""/@"hello",@"""hello"""`
  - `string == System.String`
- **Example:**

```
using System;
public class Literals {

  static void Main() {

    long @long = 25L;
    ushort @ushort = (ushort) 25Ul;

    int hex = 0X19;

    string regular = "Hello";
    string withQuotes = "\"Hello\"";
    string verbatim = @"  ""Hello""      ";

    Console.WriteLine(@"@long: {0}, @ushort: {1}, hex: {2}, regular: {3},
                        withQuotes: {4}, verbatim: {5}", @long, @ushort, hex,
                        regular, withQuotes, verbatim);
  }
}
```

Rarely Used Operators of C#:

- Left Shift : **<<, <<=**
- Right Shift: **>>, >>=**
- Bitwise Negation: **~**
- Integer bitwise AND, boolean logical AND: **&**
- Integer bitwise XOR, boolean logical XOR: **^**
- Integer bitwise OR, boolean logical OR: **|**
- **Example:**

```
using System;
public class RareOperators {

  static void Main() {
```

```
    Console.WriteLine("8 << 2:\t{0}", 8 << 2);
    Console.WriteLine("8 >> 2:\t{0}", 8 >>2);
    Console.WriteLine("~8:\t{0}", ~8);
    Console.WriteLine("8 & 2:\t{0}", 8 & 2);
    Console.WriteLine("8 ^ 2:\t{0}", 8 ^ 2);
    Console.WriteLine("2 | 8:\t{0}", 2 | 8);

  }
}

Output:
8 << 2: 32  (Same as *)
8 >> 2: 2   (Same as /)
~8:     -9
8 & 2:  0
8 ^ 2:  10  (Same as +)
2 | 8:  10  (Same as +)
```

Pre-processing Directives In C#:

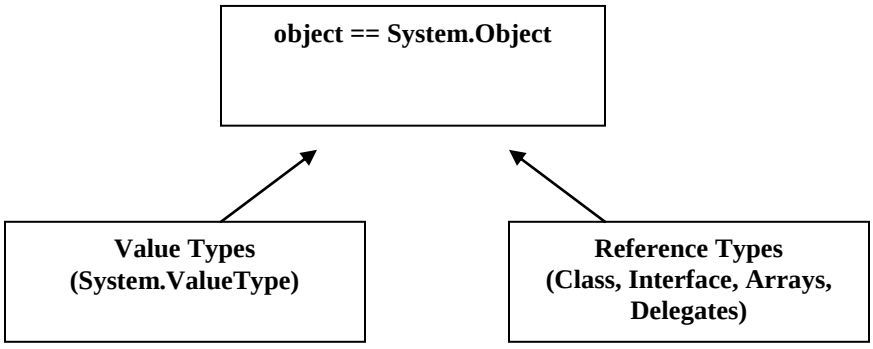
- Used to tell the compiler while code to process/skip
- Are used per file
- **#define, #undef**
- **#if, #elif, #else, #endif**
- **#error, #warning**
- **#region, #endregion**
- Types of preprocessor directives:
  Declaration, Conditional Compilation, Diagnostics, Region, Line
- **Example:**

```
#line 200          // Start with line 200
#warning Starting line is 200
#define CONIDITIONAL
#define Debug              // Debugging on
#undef Trace               // Tracing off
#undef Retail

#if CONDITIONAL
class PurchaseTransaction
{
  void Commit() {

  #if Debug
    CheckConsistency();
    #if Trace
      WriteToLog(this.ToString());
    #endif
  #endif
  CommitHelper();

 }
}
#elif Debug && Retail
  #error A build can't be both debug and retail
#else
  class Test {

  #region Application Entry Point
  static void Main() { }
  }
  #endregion
#endif
```

OOP Aspects of C#:

- C# Types (Value Types, Reference Types)
- **Reference Types:** Classes, Interfaces, Arrays, Delegates

- **Value Types:** Primitives Types – byte, short, int, float, double, decimal, etc…, Structs, Enums
- Class System.Object: The parent/root of everything in C#



- **Constructors(instance, static)**
  - **Example:**

```
using System;

class Constructor {
  static int ConstrcutedObjects;

  Constructor() { }
  Constructor(Constructor c ) { }
  // Will be called at load time to
  // initialize static fields.
  static Constructor() { ConstrcutedObjects++; }
  static int Main() {

    Console.WriteLine("static Constructor:\t{0}",
                ConstrcutedObjects);

    return 0;
  }
}
```

- **Destructors**
  - Call Finalize method of current object and every object up the hierarchy.
  - Only one destructor is allowed.
  - Invoked automatically by CLR and call is indeterminist
  - **Example:**

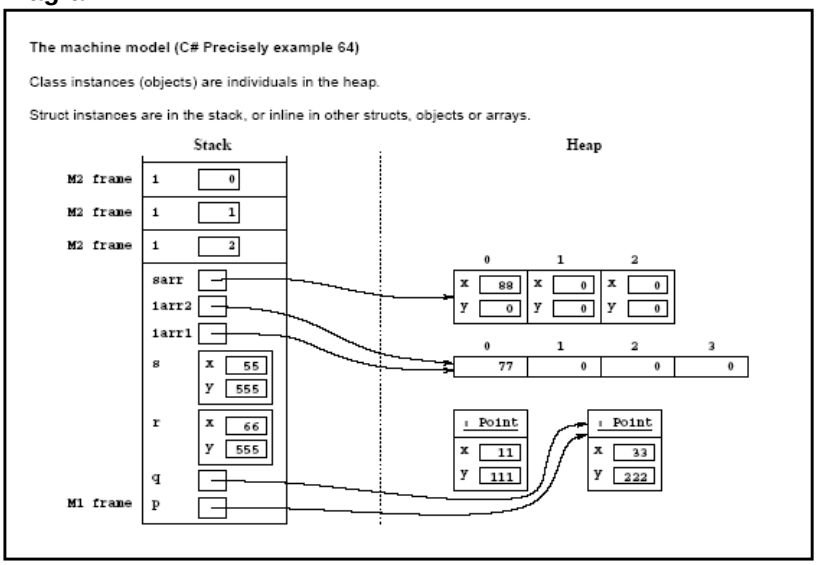
```
using System;
public class ForceGC {

  static object myObject =
    new ForceGC(new string(
      new char[] {'A','B','C'}));

  static void Main() { }

  internal ForceGC(string s) { }

  ~ForceGC()
  {
      Console.WriteLine(myObject + " will be destroyed now.");
  }
}
```

- **Interfaces:**
  - Contract for classes or structs (Signatures of methods that classes must implement or declared themselves to be abstract)

- Signatures for methods, properties, events, indexers
- **Example:**

```
using System;

interface BaseInterface {

  void Invoke();
  int  ObjectCount { get; }
  event EventHandler Invoked;
  object this[int index] { get; set; }

}

abstract class BaseClass: BaseInterface {

  public abstract void Invoke();
  public abstract int  ObjectCount { get; }
  public abstract event EventHandler Invoked;
  public abstract object this[int index] { get; set; }

  static void Main() { }

}
```

- **Structs**
  - Don't require Heap → less memory
  - Used for small Data Structure
  - Properties, Constructors, Data, Methods, Overloaded Operators
  - Default value underlying type
  - **Diagram:**



  - **Example:**

```
struct Struct {
  internal static int StructCounter;
  int x;
```

```
  internal Struct(int x) {
    StructCounter++;
    this.x = x;
  }

  public static Struct operator ++(Struct a) {
    a.X = 100;
    a.IncX();
    return a;
  }

  private void IncX() { this.x++; }
    internal int X {
      get { return this.x; }
      set { this.x = value;}
    }
}
class Class {
  internal static int ClassCounter;

  Class() { ClassCounter++; }

  static void Main() {

    object[] classArray = new Class[10];
    for(int i =0; i < classArray.Length; i++)
      classArray[i] = new Class();

    // Only one object is created in the Heap
    Struct[] structArray = new Struct[10];
    for(int i =0; i < structArray.Length; i++)
      structArray[i] = new Struct(i);

    System.Console.WriteLine("Length of object[]:\t{0}\nLength of
StructType[]:\t{1}", classArray.Length, structArray.Length);
    System.Console.WriteLine("Total of Class objects
created:\t{0}\nTotal of Struct objects created:\t{1}",
     Class.ClassCounter,  Struct.StructCounter);

  }
}
```

- **Enums**
  - Explicit Cast to underlying type
  - Default value underlying type
  - System.Enum
  - **Example:**

```
using System;

enum Letters: long {
  A = 100,B,C,D,E,F,G,H,I
}

class Test
{
  static void Main() {

   int letterA = (int) Letters.A;
   Letters a = (Letters) 100;

   Console.WriteLine("letterA:\t{0}\nLetters a:\t{1}",
    letterA,Letters.A);

  }
}
```

- **Delegates**
  - Similar to Function Pointers in C/C++ but OO & type safe
  - Indirect References to Class Methods (instance and static)

- System.Delegate
- **Example:**

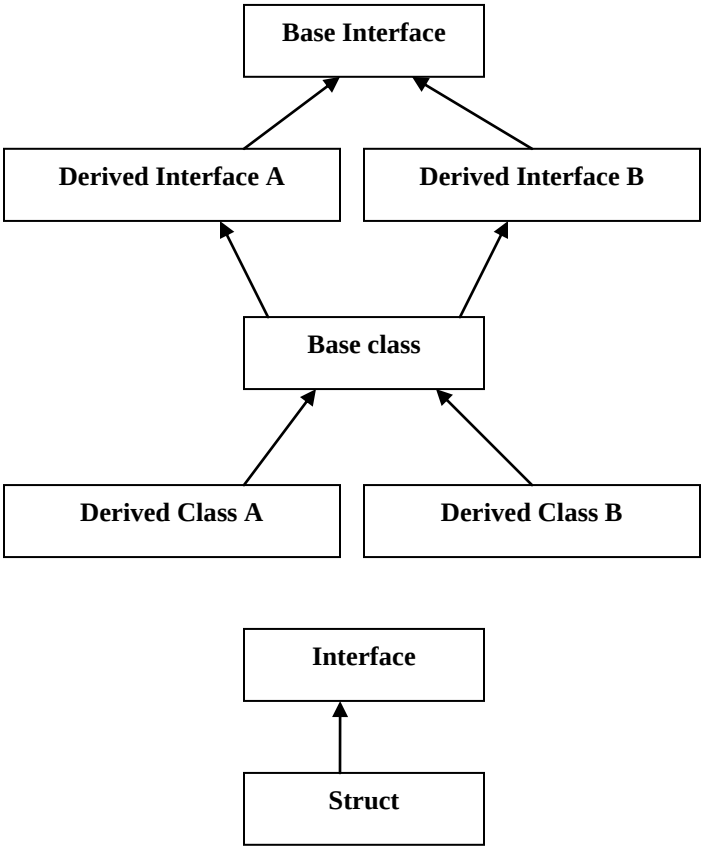
```
using System;

class DelegateTest {
  delegate void NoArgsNoReturnDelegate();

  static void Main() {

  NoArgsNoReturnDelegate nnrd =
    new NoArgsNoReturnDelegate(new   DelegateTest().PrintMe);
    nnrd();
  }
   void PrintMe() { Console.WriteLine(this); }
}
```

- **Attributes**
  - MetaInfo about classes, methods, properties
  - System.Attribute
  - HelpAttribute → [Help()]
  - Retrieve Member Info using Reflection
  - Single Use Attribute, Multiuse Attribute
  - Constructors for positional parameters
  - Properties for named parameters
  - [type: MyAttribute(…)]
  - [@Xattribute]: for removing ambiguity
  - Useful Attributes: AttributeUsage, Conditional, Obsolete, IndexerName, etc…

- **Example:**

```
#define DEBUG

using System;
using System.Diagnostics;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface |
                AttributeTargets.Enum | AttributeTargets.Struct |
                AttributeTargets.Method, AllowMultiple = true,
Inherited = false)]
public class ProjectInfoAttribute: Attribute {

  private double version;
  private string description;
  private string name;
  private string programmer;

  // Positional Parameters
  public ProjectInfoAttribute(string name): this(name,null){}
  public ProjectInfoAttribute(string name, string description):
this(name,description, 0) {}
  public ProjectInfoAttribute(string name, string description, double
version) {
    this.version = version;
    this.description = description;
    this.name = name;
  }

  // Named Parameter
  public string Progammer {
    get { return this.programmer; }
    set { this.programmer = value; }
  }
}

[ProjectInfo("C# Notes", "Notes from C# Specification",Progammer="Haris
Buchal")]
[ProjectInfo("C# Notes", "Notes from C# Specification", 1.0,
Progammer="Nuaman Cheema")]
class TestAttribute
{
  [Conditional("DEBUG")]
  public void Display(string text) {
    Console.WriteLine(text);
  }

  [Obsolete("Use string Display method")]
  public void Display(char[] text) {
    Console.WriteLine(text);
  }

  static void Main() {
    new TestAttribute().Display(new char[]{'A','B','C'});
  }
}
```

- **Inheritance:**
  - Base Classes, Derived Classes, Structs, Multiple inheritance via Interfaces.

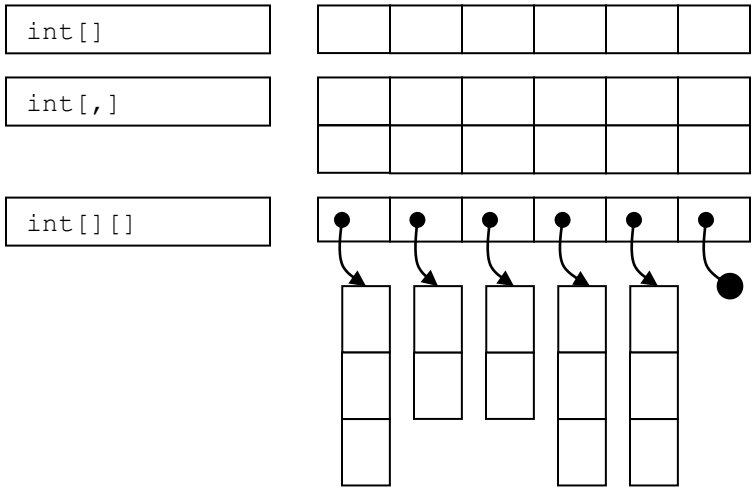


  - **Example:**

```
interface BaseInterface { }
interface DerivedInterfaceA: BaseInterface { }
interface DerivedInterfaceB: BaseInterface { }

class BaseClass: DerivedInterfaceA, DerivedInterfaceB { }
class DerivedClassA: BaseClass { }
class DerivedClassB: BaseClass { }

class Test {
   static int Main() { return 0; }
}

struct StructType: DerivedInterfaceA, DerivedInterfaceB { }
```

Arrays In C#:

- 3 types of arrays: **single-dimensional**, **multi-dimensional**, and **arrays of arrays**

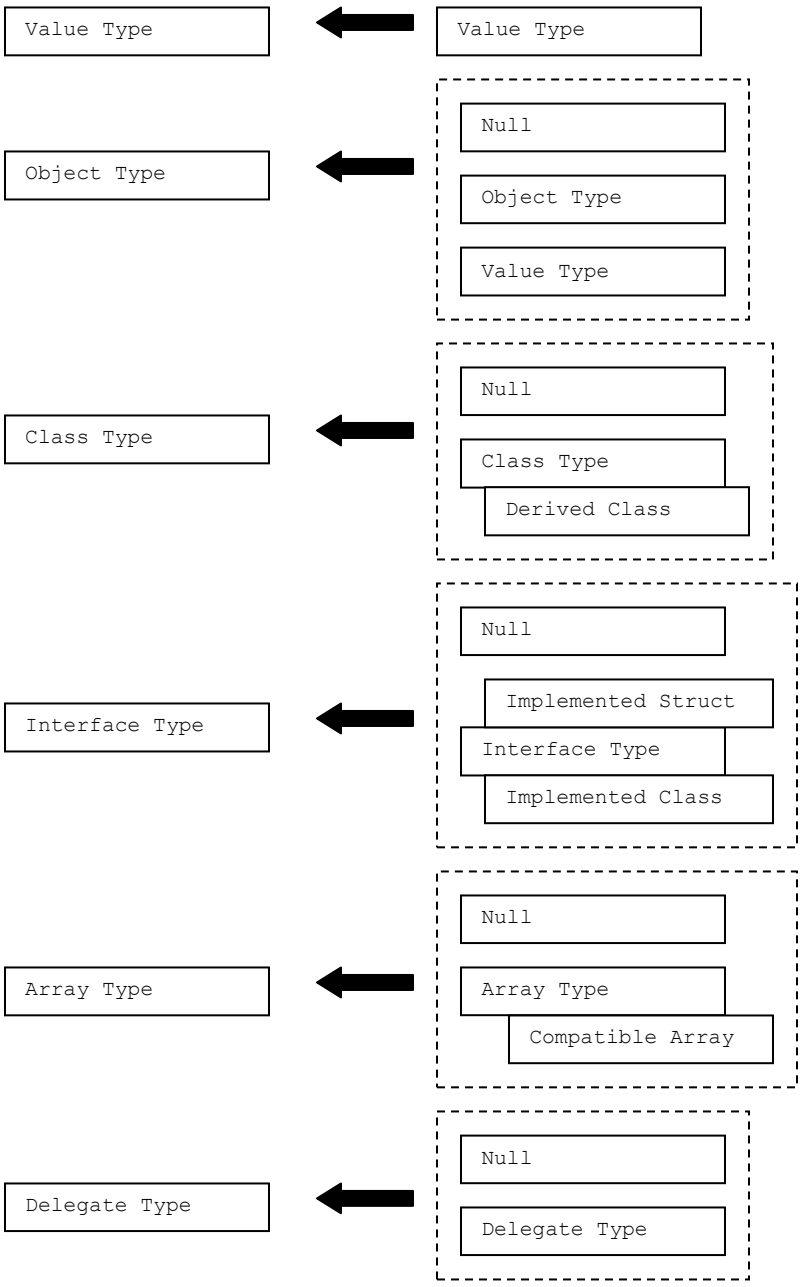


- An array is a single object in C# of type System.Array.
- **Examples:**

```
using System;
class CSharpArrays {

  static void Main() {

    int size = 2;

    int[] singleDimensionArray = new int[size];

    Console.Write("Elements of Single Dimensional Array:\t");
    foreach( int i in singleDimensionArray )
      Console.Write("{0} ", i);
    Console.WriteLine();

    int[,] multiDimensionArray = new int[size,size];

   Console.Write("Elements of Multi Dimensional Array:\t");
    foreach( int i in multiDimensionArray )
      Console.Write("{0} ", i);
    Console.WriteLine();

    int[][] arrayofArrays  = new int[size][];

    //Quiz: What is the output of the following code?

    Console.Write("Elements of Array of Arrays:\t");
    foreach( int[] i in arrayofArrays )
      Console.Write("{0} ", i.ToString());
    Console.WriteLine();

  }
}
```

Assignments in C#:

- Following rules are considered when using assignment **operator (=)** in C#.



- **Example:**

```
using System;

class AssignmentRules: IDisposable {

  static void Main() {

    ValueType monday = DayOfWeek.Monday;
    object day = null;
    day = monday;
    day = monday.ToString();

    A a = new S();
    IDisposable dispose = new AssignmentRules();

    IDisposable[] disposeables = null;
    disposeables = new AssignmentRules[10];

    NoArgs na1 = new NoArgs(Main);
    NoArgs na2 = null;
    na2 = new NoArgs(new AssignmentRules().Dispose);
    na1 = na2;

  }

  delegate void NoArgs();

  interface A { }

  struct S: A { }

  public void Dispose() { }

}
```

Access Modifiers in C#:

- **public:** `public class A {}`
- **protected:** `protected void Method() {}`
- **internal:** `protected void Method() {}`
- **protected internal:** `protected internal string getFullName() {}`
- **private:** `private int x;`
- **new**
  - **Hide specific method:** `new public void Method() {}`
  - **Hide all methods:** `class A{ new class B{} }`
- **sealed**
  - Cannot be derived
  - Classes/Methods/Properties/Events
  - Must include override for methods/properties
  - Used for preventing inheritance
  - **Example:**

```
sealed class Square {
  sealed override public string ToString() { return "Square"; }
  public string SIDES { get { return ""; } }

}
```

- **volatile**
  - Reordering optimizations are restricted
  - Used for Acquired Semantics(read), Release Semantics(write)
  - Used on Integral Types, Enum Types, Reference Types, Booleans

- **Example:**

```
using System;
using System.Threading;
class Test
{
  public static int result;
  public static volatile bool finished;
  static void Thread2()
  {
    result = 143;
    finished = true;
  }
  static void Main()
  {
    finished = false;
    // Run Thread2() in a new thread
    new Thread(new ThreadStart(Thread2)).Start();
    // Wait for Thread2 to signal that it has a result by setting
    // finished to true.
    for (;;)
    {
            if (finished)
            {
                    Console.WriteLine("result = {0}", result);
                    return;
            }
    }
  }
}
```

- **Override**
  - Used for overriding inherited methods
  - **Example:** `public override string ToString() { return "Test";}`

- **Virtual**
  - Methods/Properties/Indexers
  - **Example:**

```
using System;

class Test
{
  public static int result;
  static void Main() {}

  public override string ToString() { return "Test";}

  public virtual void CallMe() { }

  public virtual int Result { get { return result; } }

  public virtual int this[int result] { get { return Test.result; } }
}
```

- **Abstract**
  - Classes/Methods/Properties/Events/
  - Derived class must implement abstract information or declare itself to be abstract

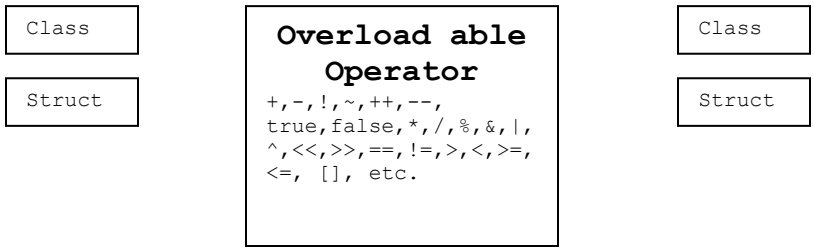
- **Example:**

```
using System;

interface Shape {
   void Draw();
}

abstract class RoundShape: Shape {
  public abstract void Draw();
  public abstract double Area { get; set;}
  public abstract event EventHandler AreaChanged;
}
```

Namespaces and Default Access Scope:

- **Namespaces** → `public`
- **Types → internal** `(public/Internal)`
- **Class Members** → `private (public/protected/internal/protected internal/private)`
- **Struct Members** → `private (public/internal/private)`
- **Interface Members** → `public`
- **Enum Members** → `public`

Method/Operator Overloading:



- Some Rules Governing Operator Overloading in C#:
  - Overloading operators signature must contain keywords `public` and `static` access modifiers.
  - Return Type is the class containing the operator, not base class, not any other type,
  - At least one of the parameters must be the class which is overloading the operator

- **Example:**

```
using System;

interface BaseInterface { }
interface DerivedInterfaceA: BaseInterface {}
interface DerivedInterfaceB: BaseInterface { }

class BaseClass: DerivedInterfaceA, DerivedInterfaceB {

  BaseInterface a;
  BaseInterface b;

  protected internal BaseClass(): this(null,null) { }

  protected internal BaseClass(BaseInterface a, BaseInterface b)
  {
    this.a = a;
    this.b = b;
  }

  public static BaseInterface operator +(BaseClass a, BaseClass b) {

     return new BaseClass(a,b);
  }

  public BaseClass GetInstance() { return this; }

  public BaseClass GetInstance(BaseClass b) { return b; }

}

class DerivedClassA: BaseClass { }
class DerivedClassB: BaseClass { }

class Test {

   static int Main() {

     BaseInterface baseInt = new DerivedClassA() + new DerivedClassB();

     Console.WriteLine("+ operator overloading example with classes:\t" +
baseInt);

     StructType structPreInc = new StructType(100);

     Console.WriteLine("++ operator overloading example with structs:\t" +
(++structPreInc).X);
     return 0;

   }
}

struct StructType: DerivedInterfaceA, DerivedInterfaceB {

    private int x;

    internal StructType(int x) { this.x = x; }

    public static StructType operator ++(StructType a) {

      a.x++;
      return a;
    }

    internal int X { get { return this.x; } }
}
```

Some Important C# keywords/features:

- **Indexers**
  - Used to index objects similar to arrays
  - Can be Overloaded
  - `new/public/protected/internal/private/virtual/sealed/override/abstract/extern`
  - **Example:**

```
using System;
class Indexer
{
  static Indexer[] objects;
  internal static int objectCount;
  int objectId;

  Indexer() {

    if(objectCount == 0)
    {
      objects = new Indexer[++objectCount];
      objects[0] = this;
    }
    else
    {
      Indexer[] temp = new Indexer[++objectCount];
      objects.CopyTo(temp,0);
      objects = temp;
      objects[objectCount - 1] = this;
    }
    objectId = objectCount - 1;
  }

  public Indexer this[int objectCount] {
    get { return objects[objectCount]; }
    set { objects[objectCount] = value; }
  }

  public Indexer this[Indexer i] {
    get { return objects[i.objectId]; }
  }

  public int Length {
    get { return objects.Length; }
  }

  static void Main() {

    for(int i = 0; i < 5; i++ )
      new Indexer();

    Indexer idx = new Indexer();

    for(int i = 0; i < 6; i++)
      Console.WriteLine(idx[i]);

    Console.WriteLine("Total {0} Indexers are indexed", idx.Length);

    Console.WriteLine(idx[idx]);
  }

}
```

- **Inner Classes**
  - **Example:**

```
class A {

 static void Main() {

  System.Console.WriteLine(A.B.id);
 }
 class B{ internal static int id;}

}
```

- **checked/unchecked**
  - Used for checking overflow
  - **Example:**

```
using System;

class CheckedCode {

  static int x = 1000000, y = 1000000;

  static void Main() {

    Console.WriteLine(unchecked(x* y));

    try {
      Console.WriteLine(checked(x * y));
    }

    catch(Exception e) {
      Console.WriteLine(e.Source + "\n" + e.StackTrace);
    }
 }
}
```

- **lock**
  - Mutual-Exclusion access to object
  - **Example**

```
using System;
using System.Threading;

class LockTest {


  static int sharedResource = 100;

  static void Main() {

     Thread t1 = new Thread(new ThreadStart(Increment));
     Thread t2 = new Thread(new ThreadStart(Decrement));

     lock((object) sharedResource) {
       t1.Start();
       t2.Start();
     }

     Console.WriteLine(sharedResource);
  }


  static void Increment() { ++sharedResource; }
  static void Decrement() { --sharedResource; }
}
```

- **using**
  - **Example:**

```
using System;

class Using: IDisposable {

  static void Main() {

    using(Using u = new Using()) {
      Console.WriteLine(u);
    }
  }

  public void Dispose()
  {
    GC.Collect();
  }
}
```

- **ref/out/params**
  - **Example:**

```
using System;

class Params {

  static void Main() {

    Params p1 = new Params();
    Params p2;
    DoSomethings(ref p1, out p2, new Params(), new Params());
  }

  public int x = 100;

  static void DoSomethings(ref Params pRef, out Params pOut, params
object[] args)
  {
    pRef.x = 100;
    pOut = new Params();
    Console.WriteLine(args.Length);
  }
}
```

- **event**
  - Bind methods to event via a delegate
  - add/remove
  - Return Type is Delegate Type
  - **+ (add) / - (remove)**
  - **+= (binds an event handler) / -= (unbind an event handler)**
  - Example:

```
using System;

class Events {

  public delegate void WakeupTasks(object sender, EventArgs e);
  public WakeupTasks W;
  public event WakeupTasks Wakeup {
    add {
      lock(this)  {
        W = W + value;
      }
    }
    remove {
      lock(this) {
        W -= value;
      }
    }
  }
```

```
}
```

- **implicit**
  - **Used to implement operators for implicitly converting user-defined types to system-defined types and vice versa.**
  - **Example:**

```
using System;

class Implicit {

  public int val;

  public Implicit(int val) { this.val = val; }

  public static implicit operator Implicit(int i )
  {  return new Implicit(i); }

  public static implicit operator int(Implicit i ) { return i.val; }

  static void Main() {

    Implicit i = new Implicit(0);
    i = 100;
    int num = i;
    Console.WriteLine(i);
  }
}
```

Output: 100

- **explicit**
  - Used to implement operators for explicitly converting user-defined types to system-defined types and vice versa.
  - **Example:**

```
using System;

class Explicit {


  public int val;

  public Explicit(int val) { this.val = val; }

  public static explicit operator Explicit(int i ) {  return new
Explicit(i); }

  public static explicit operator int(Explicit i ) { return i.val; }


  static void Main() {

    Explicit i = new Explicit(0);
    i = (Explicit) 100;
    int num = (int) i;
    Console.WriteLine((int)i);
  }
}
```

- **typeof/is/as**
  - **Example:**

```
using System;

class TypeOfAsIs {

  static void Main() {

    object t = typeof(TypeOfAsIs);

    if(t is Type)
    {
      Console.WriteLine(t);
      t = t as TypeOfAsIs;
    }

    if( t == null )
      Console.WriteLine("null");

  }
}
```

- **const**
  - Evaluated at compile time
  - Can Reference like static files
  - **Example:**

```
using System;

class Constants {

  internal const int ONE = 1;
  internal const int TWO = 2;
  //...
  internal const int HUNDRED = 100;

  public const string usa = "America";
}

class Test {


  static void Main() {

    Console.WriteLine("{0} is no. {1}",
      Constants.usa, Constants.HUNDRED);
  }
}
```

- **readonly**
  - Evaluated at runtime
  - **Example:**

```
using System;

struct Days {

  public static readonly int MONDAY = 1;
}
```

Native Features of C#:

- **Pointers: ***, **&**, **->**, **fixed**, **stackalloc**, **sizeof**, **unsafe**
  - unmanaged type
  - sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, Enum Type, Pointer Type, user-defined structs containing unsafe field(s) only.
  - **int*, int**, int*[], void***
- **unsafe**
  - **class/interface/struct/delegate**
  - **field/method/property/indexer/events/operator/ (instance & static) constructor/destructor**
  - **unsafe block { }**
  - **`csc /unsafe File`**
- **External Methods/Functions:** extern, [DllImport]
  - Used to specify calls to external methods in C/C++
- **Example:**

```
using System;
using System.Runtime.InteropServices;

struct Point
{
    public int x;
    public int y;
    public override string ToString() {
            return "(" + x + "," + y + ")";
    }
}

class Test
{
    static  Point point;
    unsafe static void Main() {

            char c = 'A';
            char* pc = &c;
            void* pv = pc;
            int* pi = (int*)pv;
            Console.WriteLine(++*pi);

            unsafe {
                    fixed(Point* pt = &point) {
                    pt->x = 10;
                    pt->y = 20;
                    Console.WriteLine(pt->ToString());
              }
            }

            double d = 123.456e23;
            unsafe {
               byte* pb = (byte*)&d;
                    for (int i = 0; i < sizeof(double); ++i)
                    Console.Write("{0:X2} ", *pb++);
                    Console.WriteLine();
            }
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
    }
     // Heap API functions
    [DllImport("kernel32")]
    static extern int GetProcessHeap();
    [DllImport("kernel32")]
    static extern void* HeapAlloc(int hHeap, int flags, int size);
}
```