C# Programming Language (Lecture 31-34)
An Introduction

C# was released by Microsoft in June 2000 as part of the .Net framework. It was co-authored by Anders Hejlsberg (who is famous for the design of the Delphi language), and Scott Wiltamuth. C# is a strongly-typed object-oriented language. It is Similar to Java and C++ in many respects. The .NET platform is centered on a Common Language Runtime (CLR - which is similar to a JVM) and a set of libraries which can be exploited by a wide variety of languages which are able to work together by all compiling to an intermediate language (IL).

Java and C# - Some Commonalities

Java and C# are very similar and have a number of commonalities.

a. Both of these languages compile into machine-independent language-independent code which runs in a managed execution environment.
b. Both C# and Java compile initially to an intermediate language: C# to Microsoft Intermediate Language (MSIL), and Java to Java bytecode. In each case the intermediate language can be run - by interpretation or just-in-time compilation - on an appropriate 'virtual machine'. In C#, however, more support is given for the further compilation of the intermediate language code into native code.
c. Both of these have garbage Collection coupled with the elimination of pointers (in C# restricted use is permitted within code marked unsafe).
d. Both have powerful reflection capabilities.
e. In case of both these languages, there is no header files, all code scoped to packages or assemblies, no problems declaring one class before another with circular dependencies.
f. Both of these support only OOP and classes all descend from object and must be allocated on the heap with new keyword.
g. Both support concurrency through thread support by putting a lock on objects when entering code marked as locked/synchronized.
h. Both have single inheritance and support for interfaces.
i. There are no global functions or constants, everything belongs to a class.
j. Arrays and strings with built-in bounds checking.
k. The "." operator is always used and there are no more ->, :: operators.
l. null and boolean/bool are keywords.
m. All values must be initialized before use.
n. Integers expressions cannot be used in 'if' statements and conditions in the loops.
o. In both languages try Blocks can have a finally clause.

Some C# features which are different from Java

a. C# has more primitive data types than Java and has more extension to the value types.
b. It supports safer enumeration types whereas Java does not have enumeration types.

c. It also has support for struct which are light weight objects.
d. There is support for operator overloading.
e. C# has the concept of 'delegates' which are type-safe method pointers and are used to implement event-handling.
f. It supports three types of arrays: single dimensional, multi-dimensional rectangular and multi-dimensional jagged arrays.
g. It has restricted use of pointers. The 'switch' statements in C# have been changed so that 'fall-through' behavior is disallowed.
h. It also has support for class 'properties'.

C# Hello World

Let us have a look at the Hello World Program in C#.

```
using System;                    // System namespace
public class HelloWorld
{
   public static void Main()              // the Main function starts
                                                  // with capital M
   {
            Console.WriteLine("Hello World!");
    }
}
```

In this example, it may be noted that just like Java and C++, C# is case sensitive. As we have in Java, everything in C# has to be inside a class and there is no semicolon at the end of the class. However, unlike Java, name of the class and the name of the file in which it is saved do not need to match up. You are free to choose any extension for the file, but it is usual to use the extension '.cs'. It supports single line and multiple line comments.

Variable Types: Reference Types and Value Types

As mentioned earlier, C# is a type-safe language. Variables can hold either value types or reference types, or they can be pointers. When a variable v contains a value type, it directly contains an object with some value and when it contains a reference type, what it directly contains is something which refers to an object.

Built-in Types

The following table lists the built-in C# types and their ranges.

C# Type	**.Net Framework (System) type**	**Signed?**	**Bytes Occupied**	**Possible Values**
sbyte	System.Sbyte	Yes	1	-128 to 127
short	System.Int16	Yes	2	-32768 to 32767
int	System.Int32	Yes	4	-2147483648 to 2147483647
long	System.Int64	Yes	8	-9223372036854775808 to 9223372036854775807
byte	System.Byte	No	1	0 to 255
ushort	System.Uint16	No	2	0 to 65535
uint	System.UInt32	No	4	0 to 4294967295
ulong	System.Uint64	No	8	0 to 18446744073709551615
float	System.Single	Yes	4	Approximately ±1.5 x 10-45 to ±3.4 x 1038 with 7 significant figures
double	System.Double	Yes	8	Approximately ±5.0 x 10-324 to ±1.7 x 10308 with 15 or 16 significant figures
decimal	System.Decimal	Yes	12	Approximately ±1.0 x 10-28 to ±7.9 x 1028 with 28 or 29 significant figures
char	System.Char	N/A	2	Any Unicode character (16 bit)

bool	System.Boolean	N/A	1 / 2	true or false

Structs

1. Java did not have structs which have been brought back by C#. However, as compared to C++, they are significantly different in C#.
2. In C++ a struct is exactly like a class, except that the default inheritance and default access are public rather than private.
3. In C# structs are very different from classes.
4. Structs in C# are designed to encapsulate lightweight objects.
5. They are value types (not reference types), so they're passed by value.
6. They are sealed, which means they cannot be derived from or have any base class other than System.ValueType, which is derived from Object.
7. Structs cannot declare a default (parameterless) constructor.
8. Structs are more efficient than classes, that's' why they are perfect for the creation of lightweight objects.

Properties

C# has formalized the concept of getter/setter methods. The relationship between a get and set method is inherent in C#, while has to be maintained in Java or C++. For example, in Java/C++ we will have to write code similar to the one shown below:

```
public int getSize() {
        return size;
}

public void setSize (int value) {
        size = value;
}

foo.setSize (getSize () + 1);
```

In C# we can define a property and can use it as if we were using a public variable. This is shown below:

```
public int Size {
        get {return size; }
        set {size = value; }
}

foo.size = foo.size + 1;
```

Reference types

In C#, the pre-defined reference types are object and string. As mentioned earlier, object is the ultimate base class of all other types. New reference types can be defined using 'class', 'interface', and 'delegate' declarations. Reference types actually hold the value of a memory address occupied by the object they reference.

Reference types however suffer from the problem of aliasing as shown below:

```
object x = new object();
x.myValue = 10;
object y = x;
y.myValue = 20;      // after this statement both x.myValue and y.myValue
                     // equal 20
```

There is however no aliasing in string. That is, strings are immutable. The properties of an immutable object can't be modified. So in order to change what a string variable references, a new string object must be created. Following is an example that elaborates this concept:

```
string s1 = "hello";
string s2 = s1; // s2 points to the same strings as s1
s1 = “world”;        // a new string is created and s1 points to it.
                     // s2 keeps pointing to the old strings
```

--------------------------------------END OF LECTURE 31--------------------------------------

Pointers

Pointers were present in C++ but Java designers took them out. C# has brought them back. However, C#, pointers can only be declared to hold the memory addresses of value types. That is, we cannot have pointers for reference types. The rest is very similar to C++. This is illustrated with the help of the following example:

```
int i = 5;
int *p;
p = &i;          // take address of i
*p = 10;         // changes the value of i to 10
```

One major difference between C++ and C# is that the ‘*’ applies to the type. That is, as opposed to C++, in C#, the following statement would declare two pointers p1 and p2:

```
int * p1, p2;
```

Just like C++, the dereference operator '->' is used to access elements of a struct type.

Pointers and unsafe code

In C#, we have two modes for the code, the managed and unmanaged. Theses are elaborated as below:

Managed code

1. Managed code is executed under the control of Common Language Runtime (CRL).
2. It has automatic garbage collection. That is, the dynamically allocated memory area which is no longer is in use is not destroyed by the programmer explicitly. It is rather automatically returned back to heap by the built-in garbage collector.
3. There is no explicit memory's allocation and deallocation and there is no explicit calls to the garbage collector i.e. call to destructor.

Unmanaged code (Java does not have this concept**)**

The unmanaged code provides access to memory through pointers just like C++. It is useful in many scenarios. For example:

- Pointers may be used to enhance performance in real time applications.
- **External Functions:** In non-.net DLLs some external functions requires a pointer as a parameter, such as Windows APIs that were written in C.
- **Debugging:** Sometimes we need to inspect the memory contents for debugging purposes, or you might need to write an application that analyzes another application process and memory.

unsafe

The keyword unsafe is used while dealing with pointer. The name reflects the risks that you might face while using it. We can declare a whole class as unsafe as shown below:

```
unsafe class Class1 {
        //you can use pointers here!
}
```

Or only some class members can be declared as unsafe:

```
class Class1 {
        //pointer
        unsafe int * ptr;
        unsafe void MyMethod() {
                //you can use pointers here
        }
}
```

To declare unsafe local variables in a method, you have to put them in unsafe blocks as the following:

```
static void Main() {
        //can't use pointers here
        unsafe
        {
                //you can declare and use pointer here
        }
        //can't use pointers here
}
```

Disadvantages of using unsafe code in C#:

- Complex code
- Harder to use
- Pointers are harder to debug
- You may compromise type safety
- Misusing might lead to the followings:
    - Overwrite other variables
    - Illegal access to memory areas not under your control
    - Stack overflow

Garbage Collector Problem in C# and the fixed keyword

When pointers are used in C#, Garbage collector can change physical position of the objects. If garbage collector changes position of an object the pointer will point at wrong place in memory. To avoid such problems C# contains ‘*fixed*’ keyword – informing the system not to relocate an object by the garbage collector.

Arrays

Arrays in C# are more similar to Java than to C++. We can create an array as did in Java by using the new operator. Once created, the array can be used as usual as shown below:

```
int[] i = new int[2];
i[0] = 1;
i[1] = 2;
```

By default all arrays start with their lower bound as 0. Using the .NET framework's System.Array class it is possible to create and manipulate arrays with an alternative initial lower bound.

Types of Arrays:

- Single Dimensional Arrays
- Multidimensional arrays
    - Rectangular
    - Jagged

Rectangular Arrays:

A rectangular array is a single array with more than one dimension, with the dimensions' sizes fixed in the array's declaration. Here is an example:

```
int[,] squareArray = new int[2,3];
```

As with single-dimensional arrays, rectangular arrays can be filled at the time they are declared.

Jagged Arrays (Similar to Java jagged arrays**)**

Jagged arrays are multidimensional arrays with irregular dimensions. This flexibility derives from the fact that multidimensional arrays are implemented as arrays of arrays.

```
int[][] jag = new int[2][];
jag[0] = new int [4];
jag[1] = new int [6];
```

Each one of jag[0] and jag[1] holds a reference to a single-dimensional int array.

Pointers and Arrays

Just as in C++, a pointer can be declared in relation to an array:

```
int[] a = {4, 5};
int *b = a;
```

In the above example memory location held by b is the location of the first type held by a. This first type must, as before, be a value type. If it is reference type then compiler will give error.

Enumerations

C# brought back enumerations which were discarded by Java designers. However, they are slightly different from C++.

- An enumeration is a special kind of value type limited to a restricted and unchangeable set of numerical values.
- By default, these numerical values are integers, but they can also be longs, bytes, etc. (any numerical value except char) as will be illustrated below.
- Type safe
- Consider the following example:

```
public enum DAYS {
        Monday=1,
        Tuesday,
        Wednesday,
        Thursday,
        Friday,
        Saturday,
        Sunday
}
```

In C# enumerations are type-safe, by which we mean that the compiler will do its best to stop you assigning illicit values to enumeration typed variables. For instance, the following code should not compile:

```
int i = DAYS.Monday;
DAYS d = i;
```

In order to get this code to compile, you would have to make explicit casts both ways (even converting from DAYS to int), ie:

```
int i = (int)DAYS.Monday;
DAYS d = (DAYS)i;
```

A useful feature of enumerations is that one can retrieve the literal as a string from the numeric constant with which it is associated. In fact, this is given by the default ToString() method, so the following expression comes out as true:

```
DAYS.Monday.ToString()=="Monday"
```

---END OF LECTURE 32----------------------------------

Boolean

- In C#, Boolean values do not convert to integers and Boolean values (true, false) do not equate to integer variables. Thus, you may not write:

```
if ( someFuncWhichReturnsAnIntegerValue() )
```

- No arithmetic expression is allowed where Boolean expressions are expected. Thus, if you write:

```
if (x = y)     // compilation error
```

  It will not be compiled.

Operator evaluation order

As discussed earlier, in C/C++ the operator evaluation order is not specified. We have discussed at length how that creates problems. In C#, it is strictly from left to right and hence there is no ambiguity.

For example,

```
a=5;
 x=a++ + --a;
```

If evaluated left-to-right

```
x=10
```

If evaluated right-to-left

```
x=8
```

Conversion

Like Java supports implicit widening conversion only. For narrowing conversion, fro example from float to int, the programmer has to explicitly state his intentions.

Loops

C# provides a number of the common loop statements. These include while, do-while, for, and foreach.

The syntax of while, do-while, and for loops is similar to C++. The only difference is that the loop control expression must be of Boolean type.

foreach loop

The 'foreach' loop is used to iterate through the values contained by any object which implements the IEnumerable interface. It has the following syntax:

```
foreach (variable1 in variable2) statement[s]
```

When a 'foreach' loop runs, the given variable1 is set in turn to each value exposed by the object named by variable2. Here is an example:

```
int[] a = new int[]{1,2,3};
foreach (int b in a)
System.Console.WriteLine(b);
```

Other Control Flow Statements

C# supports a number of control statements including break, continue, goto, if, switch, return, and throw.

switch statement

They are more or less similar to their counterparts in C++. The switch statement is however significantly different is explained below. The syntax of the switch statement is given below:

```
switch (expression)
{
        case constant-expression:
        statements
        jump statement
        [default:
                statements
                jump statement
        ]
}
```

The expression can be an integral or string expression. Control does not fall through. Jump statement is required for each block – even in default block. Fall through is allowed to stack case labels as shown in the following example:

```
switch(a){
case 2:
        Console.WriteLine("a>1 and ");
        goto case 1;
case 1:
        Console.WriteLine("a>0");
        break;
default:
        Console.WriteLine("a is not set");
        break;
}
```

As mentioned earlier, we can also use a string in the switch expression. This is demonstrated with the help of the following example:

```
void func(string option){
   switch (option)
   {
   case "label":
      goto case "jump":
   case "quit":
      return;
   case "spin":
      for(;;){ }
    case "jump":
    case "unwind":
      throw new Exception();
   default:
      break;
   }
}
```

In this example, note that jump and unwind are stacked together and there is no jump statement in the case of jump. When a case is empty, that is there is no statement in the body of the case then it may not have any jump statement either.

Class

1. **Class Attributes:**
   - Attributes are used to give information to .Net compiler
     - E.g. it is possible to tell the compiler that a class is compliant with >Net Common Language Specification.

```
[CLSCompliant (true)]
publc class MyClass
{
        //class code
}
```

2. **Class Modifiers**

   Support for OOP is provided through classes. In C#, we have the following modifiers:

- **public**: same as C++
  - **internal**: internal is accessible only to types within the same assembly which is similar to package in Java.
- **protected**: same as C++
- **internal protected**: protected within the same assembly
- **private**: same as C++
  - **new:**
  - The 'new' keyword can be used for 'nested' classes.
  - A nested class is one that is defined in the body of another class; it is in most ways identical to a class defined in the normal way, but its access level cannot be more liberal than that of the class in which it is defined.
  - Classes are usually specified independently of each other. But it is possible for one class to be specified within another's specification. In this case, the latter class is termed a nested class.
  - A nested class should be declared using the 'new' keyword just in case it has the same name as (and thus overrides) an inherited type.
- **abstract**: A class declared as 'abstract' cannot itself be instanced - it is designed only to be a base class for inheritance.
- **sealed**: A class declared as 'sealed' cannot be inherited from. It may be noted that structs can also not be inherited from.But it can inherit from other class.

---------------------------------------END OF LECTURE 33-----------------------------------

Method Modifiers

C# provides the following methods modifiers:

- **static:** The 'static' modifier declares a method to be a class method.
- **new, virtual, override**:

- **extern:** Similar to C extern. Mean that the method is defined externally, using a language other than C#

Hiding and Overriding

The main difference between hiding and overriding relates to the choice of which method to call where the declared class of a variable is different to the run-time class of the object it references.

For example:

```
public virtual double getArea()
{
        return length * width;
}

public override double getArea()
{
        return length * length;
}
```

For one method to override another, the overridden method must not be static, and it must be declared as either 'virtual', 'abstract' or 'override'. Now look at the following:

```
public double getArea()
{
        return length * width;
}

public new double getArea()
{
        return length * length;
}
```

Where one method 'hides' another, the hidden method does not need to be declared with any special keyword. Instead, the hiding method just declares itself as 'new'.

http://www.akadia.com/services/dotnet_polymorphism.html

Method Hiding

A 'new' method only hides a super-class method with a scope defined by its access modifier. In this case method calls do not always 'slide through' in the way that they do with virtual methods. So, if we declare two variables thus if we have:

```
Square sq = new Square(4);
Rectangle r = sq;
```

then

```
double area = r.getArea();
```

the getArea method run will be that defined in the Rectangle class, not the Square class.

Method parameters

In C#, as in C++, a method can only have one return value. You overcome this in C++ by passing pointers or references as parameters. In C#, with value types, however, this does not work. If you want to pass the value type by reference, you mark the value type parameter with the ref keyword as shown below.

```
public void foo(int x, ref int y)
```

Note that you need to use the ref keyword in both the method declaration and the actual call to the method.

```
someObject.f00(a, ref b);
```

Out Parameters

C# requires definite assignment, which means that the local variables, a, and b must be initialized before foo is called.

```
        int a, b;
        b = 0;
        someObject.f00(ref a, b);       // not allowed
                                                // a has not been
                                                // initialized
        a = 0; b =0;
        someObject.f00(ref a, b);       // now it is OK
```

This is unnecessarily cumbersome. To address this problem, C# also provides the out keyword, which indicates that you may pass in un-initialized variables and they will be passed by reference.

```
        public void foo(int x, ref int y, out int z)

        a = 0; b = 0;
        someObject.f00(a, ref b, out c);     // no need to initialize c
```

params

One can pass an arbitrary number of types to a method by declaring a parameter array with the 'params' modifier. Types passed as 'params' are all passed by value. This is elaborated with the help of the following example:

```
public static void Main(){
        double a = 1;
        int b = 2;
        int c = 3;
        int d = totalIgnoreFirst(a, b, c);
}

public static int totalIgnoreFirst(double a, params int[] intArr){
        int sum = 0;
        for (int i=0; i < intArr.Length; i++)
                sum += intArr[i];
        return sum;
}
```

Readonly fields

Readonly field are instance fields that cannot be assigned to. That is, their value is initialized only once and then cannot be modified. This is shown in the following example:

```
class Pair
{
   public Pair(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
   public void Reset()
   {
      x = 0;                          // compile time errors
      y = 0;
   }
   private readonly int x, y; // this declares Pair as an immutable read only object
}
```

The is operator

- The **'is'** operator supports run time type information.
- It is used to test if the operator/expression is of certain type
    - **Syntax**: expression is type
- Evaluates to a Boolean result.
- It can be used as conditional expression.
- It will return true
    - if the expression is not NULL
    - the expression can be safely cast to type.
- The following example illustrates this concept:

```
class Dog {
…
}

class Cat {
…
}

…
```

```
// object o;
if (o is Dog)
        Console.Writeline(“it’s a dog”);
else if (o is Cat)
        Console.Writeline(“it’s a cat”);
else
        Console.Writeline(“what is it?”);
```

The as operator

- The as operator attempts to cast a given operand to the requested type.
  - **Syntax**: expression as type
- The normal cast operation – (T) e – generates an InvalidCastException when there is no valid cast.
- The as operator does not throw an exception; instead the result returned is null as shown below:
  - expression is type ? (type) expression : (type) null

The new operator

- In C++, the new keyword instantiates an object on the heap.
- In C#, with reference types, the new keyword does instantiate objects on the heap but with value types such as structs, the object is created on the stack and a constructor is called.
- You can, create a struct on the stack without using new, but be careful! New initializes the object.
- If you don't use new, you must initialize all the values in the struct by hand before you use it (before you pass it to a method) or it won't compile.

Boxing

Boxing is converting any value type to corresponding object type and convert the resultant 'boxed' type back again.

```
int i = 123;
object box = i;     // value of i is copied to the object box
if (box is int)     // runtime type of box is returned as boxed value type
{
        Console.Write("Box contains an int");  // this line is printed
}
```

Interfaces

- Just like Java, C# also has interfaces contain method signatures.
- There are no access modifier and everything is implicitly ***public.***
- It doest not have any fields, not even ***static*** ones.

- A ***class*** can implement many ***interface***s but unlike Java there is no implements keyword .
- Syntax notation is positional where we have base ***class*** first, then base ***interface***s as shown below:
    - class X: CA, IA, IB

```
{
        …
}
```

Delegates

Delegates are similar to function pointers. C/C++ function pointers lack instance-based knowledge whereas C# delegate are event based can be thought of a call-back mechanism where a request is made to invoke a specified method when the time is right.

Delegates are reference types which allow indirect calls to methods. A delegate instance holds references to some number of methods, and by invoking the delegate one causes all of these methods to be called. The usefulness of delegates lies in the fact that the functions which invoke them are blind to the underlying methods they thereby cause to run.

An example of delegates is shown below:

```
public delegate void Print (String s);
…

public void realMethod (String myString)
{
   // method code
}
…
```

Another method in the class could then instantiate the 'Print' delegate in the following way, so that it holds a reference to 'realMethod':

```
Print delegateVariable = new Print(realMethod);
```

Exception Handling and Multithreading

Exception handling is similar to java but is less restrictive. Multithreading is similar to java.