Overview of C#
CS331

As described in the previous lecture, C# is one of the languages designed for the .NET platform. Here we will present an overview of the language, focusing on the unique aspects that differ from Java or C++. As you will see, it is quite similar to Java in many respects.

The structure of a C# program looks like the following:

```
// Specify namespaces we use classes from here
using System;
using System.Threading;  // Specify

namespace AppNamespace
{
      // Comments that start with /// used for
      // creating online documentation, like javadoc
      /// <summary>
      /// Summary description for Class1.
      /// </summary>
      class Class1
      {
            static void Main(string[] args)
            {
                  // Your code would go here, e.g.
                  Console.WriteLine("hi");
            }
            /* We can define other methods and vars for the class */

            // Constructor
            Class1()
            {
                  // Code
            }

            // Some method, use public, private, protected
            // Use static as well just like Java
            public void foo()
            {
                  // Code
            }
            // Variables
            private int m_number;
            public static double m_stuff;
      }
}
```

C# code normally uses the file extension of ".cs". Everything above is quite similar to Java; note the use of "Main" instead of "main". If a namespace is left out, your code is placed into the default, global, namespace.

The “using” directive tells C# what methods you would like to use from that namespace. If we left out the “using System” statement, then we would have had to write “System.Console.WriteLine” instead of just “Console.WriteLine”.

It is normal for each class to be defined in a separate file, but you could put all the classes in one file if you wish. Using Visual Studio .NET’s “P)roject, Add C)lass” menu option will create separate files for your classes by default.

Based on what little we have covered, together with your knowledge of Java, you should already be able to write quite sophisticated programs!

Getting Help

If you have installed MSDN on the system, you have a great online help resource built into Visual Studio .NET. You can look up the C# programming language reference from the Help menu and also get Dynamic Help, which will show related help as you are typing.

If MSDN is not installed, you can go online to access the references. It is accessible from:
http://msdn.microsoft.com/library/default.asp

You will have to drill down to VS.NET, Documentation, VB and C#, and then to the C# reference. There are also numerous tutorials here. Or you could just enter search terms into the search engine for the class or keyword you are interested in.

Output : Using WriteLine

Usually we cover how to do output first, so here are the basics.

System.Console.WriteLine() will output a string to the console. You can use this just like Java’s System.out.println():

```
System.Console.WriteLine(“hello world “ + 10/2);
```

will output:

```
hello world 5
```

We can also use {0}, {1}, {2}, … etc. to indicate arguments in the WriteLine statement to print. For example:

```
Console.WriteLine(“hi {0} you are {0} and your age is {1}”, “Kenrick”, 23);
```

will output:

```
hi Kenrick you are Kenrick and your age is 23
```

There are also options to control things such as the number of columns to use for each variable, the number of decimals places to print, etc. For example, we could use :C to specify the value should be displayed as currency:

```
Console.WriteLine("you have {0:C} dollars.", 1.3);
```

outputs as:

```
you have $1.30 dollars.
```

See the online help or the text for more formatting options.

Data Types

C# supports value types and reference types. The value types are essentially the primitive types found in most languages, and are stored directly on the stack. Reference types are objects and are created on the heap.

The built-in types are:

C# Type	.NET Framework type
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

All of these are value types except for string and object. Note the lowercase string, not String. Decimal is like a double, but uses 128 bits for greater precision (up to around 10^{28}). uint, ulong, and ushort are unsigned versions of these variables.

Due to automatic boxing and unboxing, the value types can be treated like objects. For example, the following public methods are defined for Object:

Public Object Methods

Equals	Overloaded. Determines whether two **Object** instances are equal.
GetHashCode	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType	Gets the Type of the current instance.
ToString	Returns a String that represents the current **Object**.

We can then write code such as:

```
int i;

Console.WriteLine(i.ToString());
int hash = i.GetHashCode();
```

This is equivalent to performing:

```
z = new Object(i);
Console.WriteLine(z.ToString());
```

But is much more efficient since the value type is converted to reference on demand and at the virtual code level instead of at the programmer's level.

The struct is another value type we can create. A struct can contain constructors, constants, fields, methods, properties, indexers, operators, and nested types. The declaration of a struct looks just like a declaration of a class, except we use the keyword struct instead of class. For example:

```
public struct Point
{
   public int x, y;
   public Point(int p1, int p2)
   {
      x = p1;
      y = p2;
   }
}
```

So what is the difference with a struct? Unlike classes, structs can be created on the stack without using the keyword new, e.g.:

```
Point p1, p2;
p1.x = 3; p1.y = 5;
```

etc. We also cannot use inheritance with structs.

Finally, C# provides an enumeration type that is also a value type. This is most easily shown through an example:

```
// Enum goes outside in the class definition
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};

// Inside some method
Days day1, day2;
int day3;

day1 = Days.Sat;
day2 = Days.Tue;
day3 = (int) Days.Fri;
Console.WriteLine(day1);
Console.WriteLine(day2);
Console.WriteLine(day3);
```

This program outputs:

Sat
Tue
6

As you can see, the enumeration really maps to the underlying data type of integer.

Strings

The built-in string type is much like Java's string type. We can concatenate using the + operator. Just like Java, there are a variety of methods available to find the index Of matching strings or characters, generate substrings, compare for equality (if we use == on strings we are comparing if the references are equal, just like Java), generate clones, trim, etc. See the reference for more details.

Classes

We have already seen defining a class above. To specify inheritance use a colon after the class name and then the base class. If we want to invoke the constructor for the base class, we must use the keyword "base" after the constructor in the derived class. We must also be explicit with virtual methods. The example illustrates basic usage:

```
public class BankAccount
{
        public double m_amount;
        BankAccount(double d) {
                m_amount = d;
        }
        public virtual string GetInfo() {
                return "Basic Account";
        }
}
```

```
public class SavingsAccount : BankAccount
{
        // Savings Account derived from Bank Account
        // usual inheritance of methods, variables
        public double m_interest_rate;
        SavingsAccount(double d) : base(100) {       // $100 bonus for signup
                m_interest_rate = 0.025;
        }
        public override string GetInfo() {
                string s = base.GetInfo();
                return s + " and Savings Account";
        }
}
```

Note that we must explicitly state that a method is virtual if we want to override it. By default, non-virtual methods cannot be overridden. We also have to explicitly state that we are overriding a method. To invoke a base method, use base.methodName().

If we have some code that invokes this as follows:

```
SavingsAccount a = new SavingsAccount(0.05);
Console.WriteLine(a.m_amount);
Console.WriteLine(a.m_interest_rate);
Console.WriteLine(a.GetInfo());
```

Then the output is:

```
100
0.05
Basic Account and Savings Account
```

Interfaces

An interface in C# is much like an interface in Java. An interface states what an object can do, but not how it is done. It looks like a class definition but we cannot implement any methods in the interface nor include any variables. Here is a sample interface:

```
public interface IDrivable {
        void Start();
        void Stop();
        void Turn();
}
```

We can define classes that implement interfaces, for example:

```
public class SportsCar : IDriveable {
        void Start() {
                // Code here to implement start
        }
        void Stop() {
                // Code here to implement stop
        }
        void Turn() {
                // Code here to implement turn
        }
}
```

We can declare methods that take as input an interface, which accept any object that implements the interface, for example:

```
void GoForward(IDrivable d)
{
        d.Start();
        // wait
        d.Stop();
}
```

Getting Input

To input data, we must read it as a string and then convert it to the desired type.
Console.ReadLine() will return a line of input text as a string.
We can then use type.Parse(string) to convert the string to the desired type. For example:

```
string s;
int i;

s = Console.ReadLine();
i = int.Parse(s);
```

we can also use double.Parse(s); float.Parse(s); etc.
There is also a useful Convert class, with methods such as Convert.ToDouble(val); Convert.ToBoolean(val); Convert.ToDateTime(val); etc.

Procedural Code

We also have our familiar procedural constructs:

Arithmetic, relational, Boolean operators: all the same as Java/C++

For, While, Do, If : all the same as Java/C++

Switch statements: Like Java, except forces a break after a case. Code is not allowed to "fall through" to the next case, but several case labels may mark the same location.

Math class: Math.Sin(), Math.Cos(), etc.

Random class:

```
Random r = new Random();
r.NextDouble();          // Returns random double between 0-1
r.Next(10,20);           // Random int, 10 ≤ int < 20
```

Passing Parameters

If we pass a value parameter to a method then by default we get the pass by value behavior, just like Java. For example:

```
public static void foo(int a)
{
      a=1;
}

static void Main(string[] args)
{
      int x=3;
      foo(x);
      Console.WriteLine(x);
}
```

This outputs the value of 3 because x is passed by value to method foo, which gets a copy of x's value under the variable name of a.

However, C# allows a ref keyword to pass value types by reference:

```
public static void foo(int ref a)
{
      a=1;
}

static void Main(string[] args)
{
      int x=3;
      foo(ref x);
      Console.WriteLine(x);
}
```

The ref keyword must be used in both the parameter declaration of the method and also when invoked, so it is clear what parameters are passed by reference and may be changed. In this case, the program outputs the value of 1 since variable a in foo is really a reference to where x is stored in Main.

If we pass a reference parameter (Objects, strings, etc.) to a method, we get the same behavior as in Java. Changes to the contents of the object are reflected in the caller, since there is only one copy of the actual object in memory and merely multiple references to that object.

Once in a while it is useful to pass reference objects by reference. Consider the following:

```
public static void foo(string s)
{
      s = "cow";
}

static void Main(string[] args)
{
      string str = "moo";
      foo(str);
      Console.WriteLine(str);
}
```

This program outputs “moo” since inside method foo, the local reference parameter s is set to a new object in memory with the value “cow”. The original reference in str remains untouched.

If we instead pass the reference parameter itself by reference we see the changes reflected in the caller:

```
public static void foo(string ref s)
{
      s = "cow";
}

static void Main(string[] args)
{
      string str = "moo";
      foo(ref str);
      Console.WriteLine(str);
}
```

This program outputs “cow” since foo is passed a reference to the reference str, which changes what str points to.

Arrays

Arrays in C# are quite similar to Java arrays. Arrays are always created off the heap and we have a reference to the array data. The format is just like Java:

Type arrayname = new Type[size];

For example:

```
int arr = new int[100];
```

This allocates a chunk of data off the heap large enough to store the array, and arr references this chunk of data.

We can get the size of the array dynamically through the Length property:

```
Console.WriteLine(arr.Length);          // Outputs 100 for above declaration
```

If we want to declare a method parameter to be of type array we would use:

```
public void foo(int[] data)
```

To return an array we can use:

```
public int[] foo()
```

Just like in Java, if we have two array variables and want to copy one to the other we can't do it with just an assignment. This would assign the reference, not make a copy of the array. To copy the array we must copy each element one at a time, or use the Clone() method to make a copy of the data and set a new reference to it (and garbage collect the old array values).

Multidimensional Arrays, Vectors, foreach

We have two ways to declare multidimensional arrays. One is to create arrays of arrays, as Java does. Another is to create a true multidimensional array.

The following defines a 30 x 3 array:

```
int[,] arr = new int[30][3];
```

Here we put a comma inside the [] to indicate two dimensions. This allocates a single chunk of memory of size 30*3*sizeof(int) and creates a reference to it. We use the formulas for row major order to access each element of the array.

The following defines a 30 x 3 array using an array of arrays:

```
int[][] arr = new int[30][3];
```

To an end user this looks much like the previous declaration, but it creates an array of 30 elements, where each element is an array of 3 elements. This gives us the possibility of creating ragged arrays but is slower to access since we must dereference each array index.

Related to arrays, you may want to check out the ArrayList class defined in System.Collections. It defines a class that behaves like a Java vector in that it allows dynamic allocation of elements that can be accessed like an array or also by name using a key.

Lastly, C# provides a new loop method, called foreach. Foreach will loop through each element in an array or collection. For example:

```
string[] arr = {"hello", "world", "foo", "abracadabra"};
foreach (string x in arr) Console.WriteLine(x);
```

Will output each string in the array.

Delegates

C# uses delegates where languages such as C++ use function pointers. A delegate defines a class that describes one or more methods. Another method can use this definition, regardless of the actual code that implements it. C# uses this technique to pass the EventHandler type to different methods, where the event may be handled in different ways.

Here is an example of a delegate.

```
public delegate int CompareDelegate(string s1, string s2);

// Two different methods for comparison
public static int compare1(string s1, string s2)
{
      return (s1.CompareTo(s2));
}
public static int compare2(string s1, string s2)
{
      if (s1.Length <= s2.Length) return -1;
      else return 1;
}

// A method that uses the delegate to find the minimum
public static string FindMin(string[] arr, CompareDelegate compare)
{
      int i, minIndex=0;

      for (i=1; i<arr.Length; i++)
      {
            if (compare(arr[minIndex],arr[i])>0) minIndex=i;
      }
      return arr[minIndex];
}
```

```
static void Main(string[] args)
{
      string[] arr = {"hello", "world", "foo", "abracadabra"};
      string s;
      Console.WriteLine(FindMin(arr, new CompareDelegate(compare1)));
      Console.WriteLine(FindMin(arr, new CompareDelegate(compare2)));
}
```

The output of this code is:

abracadabra	(using compare1, alphabetic compare)
foo	(using compare2, length of string compare)

Here we have covered all of the basic constructs that exist in the C# language under the Common Language Runtime. Next we will see how to use various Windows.Forms features to create Windows applications with graphical interfaces.