C# Programming

Tutorial

What is C#?

- C# is a new programming language that combines features of Java, C, and C++
- It improves on the advantages of Java
- It uses “just in time” compilation and not a virtual machine
- It has a special runtime environment
- It is meant to run on all platforms, not just windows

- Pointers are not supported except as “unsafe code”
- C# code runs in a “managed” environment
- Garbage collection is used to destroy dynamically created objects
- C# does not allow global functions or variables. Everything is in a *class*
- A rich class library is provided, the .NET FCL
- C# programs do not have header files. All definitions and declarations are together

- C# is the natural choice language for the .NET programming infrastructure
- A language independent “component” model is used
- Components written in different languages can be mixed easily in the same application

Hello World

```
using System;
class Hello
{
   static void Main() {
      Console.WriteLine("hello, world");
   }
}
```

Namespaces

- Similar to import in Java
- Replaces the use of header files
- Similar to namespaces in C++, but associated headers not used
- **using** keyword specifies the particular class library collection of related classes, e.g. **System, System.Windows.Forms**

Naming Guidelines

- Use Camel casing for class data fields.
  - public int userCount;
  - private MyClass class1;
- Do not use Microsoft MFC convention, e.g. private int m_maxsize;
- Use Pascal casing for pretty much everything else.
  - private class NetworkManager {...}

C# Types

object	The ultimate base type of all other types
string	String type; a string is a sequence of Unicode characters
sbyte	8-bit signed integral type
short	16-bit signed integral type
int	32-bit signed integral type
long	64-bit signed integral type
byte	8-bit unsigned integral type
ushort	16-bit unsigned integral type
uint	32-bit unsigned integral type
ulong	64-bit unsigned integral type
float	Single-precision floating point type

double	Double-precision floating point type
bool	Boolean type; a bool value is either true or false
char	Character type; a char value is a Unicode character
decimal	Precise decimal type with 28 significant digits

Conversions

- C# has the usual implicit conversions
- A C style cast can also be used

```
using System;
class Test
{
   static void Main() {
      long longValue = 12345;
      int intValue = (int) longValue;
      Console.WriteLine("(int) {0} = {1}", longValue,
        intValue);
   }
}
```

Values vs. References

- Scalar types are values and are instantiated when declared
- Class types are reference types and need to be instantiated with the *new* operator

```
public class Foo {...}
Foo f = new Foo();
```

- C++ programmers beware

References contd.

- A reference is a variable that is actually a hidden pointer.
- We can assign to a reference variable.

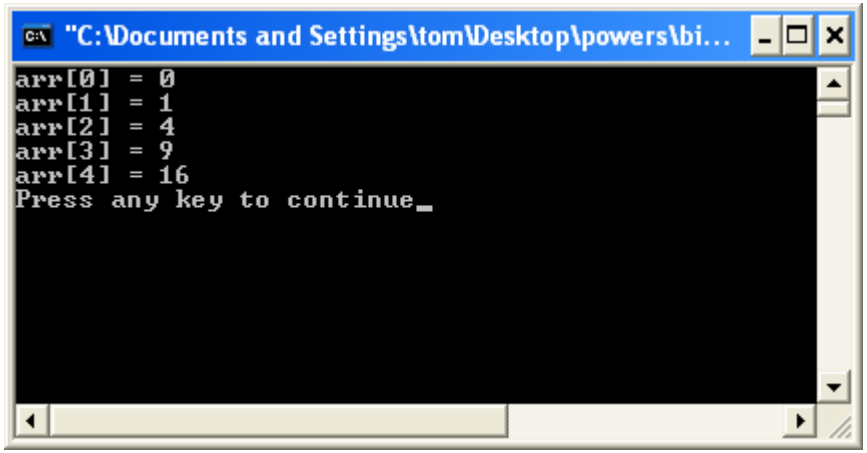
```
AClass x = new AClass();
AClass y;
y=x;    //now y and x refer to
        // the same object
```

Array Types

- Similar to C++ but allow *jagged* arrays as well

```
using System;
class Test
{
   static void Main() {
      int[] arr = new int[5];
      for (int i = 0; i < arr.Length; i++)
         arr[i] = i * i;
      for (int i = 0; i < arr.Length; i++)
         Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
   }
}
```

The Output


"C:\Documents and Settings\tom\Desktop\powers\bi...
arr[0] = 0
arr[1] = 1
arr[2] = 4
arr[3] = 9
arr[4] = 16
Press any key to continue

Object

- Object is the base class of ALL objects in C#.
- Value types have equivalent types derived from Object, e.g., Int32.
- Note - even the scalar types are actually objects and have methods.
- Example:

```
int i = 1234;
string s = i.ToString();
```

Boxing and Unboxing

- A value type can be converted to an *object* type

```
class Test
{
   static void Main() {
      int i = 123;
      object o = i;       // boxing
      int j = (int) o;    // unboxing
   }
}
```

Local Variables

- Local variables are supported as in Java or C++.
- Local variables are assigned storage on the stack. Except for unboxed scalars, objects themselves are always on the heap. Their references may be on the stack or class members.
- The C# compiler checks that a variable is initialized or assigned prior to use. If not, and error is generated

```
int i;
System.Console.WriteLine("i= {0}", i);
```

- An error is generated since i has never been initialized.

Operators and Expressions

- C# supports C++ operators and expressions.
- Java programmers should not have problems, but should check the documentation.
- Note - integer types do not evaluate to boolean as in C or C++.
- Conditional statements such as **if** require boolean expressions. You can't use an integer. C/C++ programmers beware.

Statements

- Statements are pretty much equivalent to C/C++ and Java
- There is a *foreach* statement that is new

```
static void Main(string[] args) {
    foreach (string s in args)
        Console.WriteLine(s);
}
```

- This also demonstrates the arguments to the main function.

Switch

- Switch in C# does not support fall through except for a case label with no statements, e.g.,

```
switch (i)
{
  case 1:
          <statement>;
          break;
  case 2:
  case 3:
          <statement>;
          break;
  default:
          <statement>;
};
```

Switch - contd.

```
switch (i)
{
   case 1:
        <statement>;
   case 2:
        <statement>;
}
```

- The above is NOT legal!
- No fall through is allowed.

Class Access

public	Access not limited
protected	Access limited to the containing class or types derived from the containing class
internal	Access limited to this program
protected internal	Access limited to this program or types derived from the containing class
private	Access limited to the containing type

- Default access is **private** for members and **internal** for classes.

Constructors

- Constructors are similar to Java and C++.
- Constructors can be overloaded.
- The base class constructor is invoked with the keyword "base"

```
public class Foo
{
   Foo(int i): base(i)
   {
   }
}
```

Static Members

- The **static** keyword is used to define a field (data member) or method.
- Static members exist for all instances of the class (zero or more)
- Static methods can only access other static members
- Use "dot" notation to access static members, e.g. classname.member.

Value Member Initialization

- Value members are initialized to zero unless an initializer is present or a constructor is used.
- Initializers must be constants.

```
public class Foo
{
   public int width=640;
   public int height=480;
}
```

- C++ constructor like initializations are <u>not</u> permitted for data fields.

Const Fields

- The **const** keyword allows a field to be initialized, but not subsequently modified.

```
class Foo
{
  public const double pi=3.1415;
}
```

Read-Only Fields

- Const only works if the initial value is known at compile time.
- The **readonly** keyword allows a field to be initialized by the constructor, but not subsequently modified.

```
class Foo
{
   public readonly int numUsers;
   Foo()
   {
     numUsers = <some value obtained at run-time>;
   }
}
```

Properties

- Properties make access to members safer.
- A function body is executed when a property is accessed.
- We can validate a value, limit it to bounds, or even change its internal data type all transparent to the user.

Properties

```
public class AbsVal
{
   private int ival;
   public int Val {
      get {
         return ival;
      }
      set {
         if (value<0) ival = -value;
            else ival = value;
      }
   }
}
```

Example

```
AbsVal myval;
myval.Val = -500;
Console.Writeline("Value is now {0}", myval.Val);

//output
Value is now 500
```

Method Parameters

- C# supports call by value and reference for value types.
- Reference types can only be passed by reference.
- The **ref** and **out** keywords are used to specify a value parameter is to be passed by reference
- The difference is that an **out** parameter does not have to have an initial value when the method is called

Method Parameters - contd.

```
using System;
class Foo
{
   public void MyMethod(out int outParam, ref int inAndOutParam)
   {
         outParam = 123;
         inAndOutParam += 10;
   }
   public static void Main()
   {
         int outParam;
         int inAndOutParam=10;
         Foo f = new Foo();
         f.MyMethod(out outParam, ref inAndOutParam);
   }
}
```

Method Overloading

- Methods can be overloaded as in C++
- An overloaded method must differ in the type and/or number of arguments.
- A difference in only the return type is not valid.
- Constructors can be overloaded.

Polymorphism

- Virtual methods must be declared as such or they are statically bound at compile time.
- The use of the **virtual** keyword differs from C++.
- **virtual** is used only in the base class.
- Derived classes must use the **override** keyword. C++ programmers beware!

Polymorphism - contd.

```
public class Foo
{
   virtual public void f()
   {
   }
}

public class Derived: Foo
{
   override public void f()
   {
   }
}
```

New

- If you don't want to implement an override for a virtual function you can use the *new* keyword. The function called is determined by the declaration of the reference variable and *late binding* is not used.

```
class bclass
{
   public virtual void F() {....}
}
class Foo: bclass
{
   public new void F(){....}
}
```

Abstract Classes

- These work somewhat like they do in C++
- A class with at least one abstract method can't be instantiated.
- Derived classes must implement the abstract methods.

Abstract Classes - Example

```
abstract public class bclass
{
   abstract public void F();
}

public class MyClass: bclass
{
   public override void F() {......}
}
-------
MyClass mc = new MyClass();
mc.F();
```

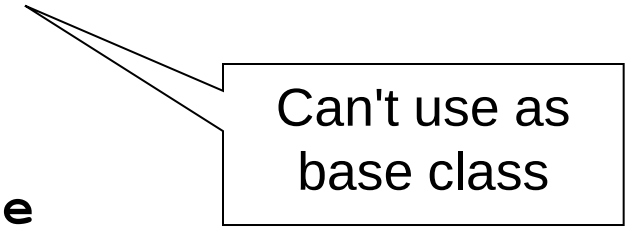
Sealed Classes

- Use the *sealed* keyword to prevent a class from being used as a base class.

```
sealed public class MyClass
{......}

public class NewClass: MyClass
{......}

//This will fail to compile
```



Nested Classes

- Class declarations can be nested.
- In other words, a class can be declared inside another class.
- This merely restricts the scope of the class and not its functionality in any way.
- The outer class name is used with dot notation to reference the inner class.
- See documentation or a book for examples.
- I don't regard this as an important feature although you will see it occasionally.

Structs

- Structs are similar to classes.
- Structs do not support inheritance.
- A struct is a **value** type.
- Use structs for types that are small, simple, and similar to built in types, e.g., a Point object.

Struct Example

```
//Example 07-01: Creating a struct

using System;

public struct Location
{
    public Location(int xCoordinate, int yCoordinate)
    {
        xVal = xCoordinate;
        yVal = yCoordinate;
    }
    public int x
    {
        get
```

```
{
            return xVal;
        }
        set
        {
            xVal = value;
        }
    }
    public int y
    {
        get
        {
            return yVal;
        }
        set
        {
```

```
 yVal = value;
        }
    }

    public override string ToString()
    {
        return (String.Format("{0}, {1}", xVal,yVal));
    }

    private int xVal;
    private int yVal;
}

public class Tester
{
    public void myFunc(Location loc)
```

```
{
        loc.x = 50;
        loc.y = 100;
        Console.WriteLine("Loc1 location: {0}", loc);
    }
    static void Main()
    {
        Location loc1 = new Location(200,300);
        Console.WriteLine("Loc1 location: {0}", loc1);
        Tester t = new Tester();
        t.myFunc(loc1);
        Console.WriteLine("Loc1 location: {0}", loc1);
    }
}
```

Struct Differences

- No custom default constructor
- Default constructor zeros fields.
- No initializations. Use a constructor that takes parameters.
- The *new* operator does not need to be used.

```
Location l;
l.x = 100;
l.y = 150;
Console.WriteLine(l);
```

Interfaces

- An interface is an alternative to an abstract base class.
- An interface is a *contract* to implement methods, properties, etc.
- A class can inherit from more than one interface (a type of multiple inheritance).
- The use of interfaces is central to the .NET framework as well as Microsoft's COM technology.

Interface Example

```
//Example 08-01: Using a simple interface

using System;

// declare the interface
interface IStorable
{
   // no access modifiers, methods are public
   // no implementation
   void Read(  );
   void Write(object obj);
   int Status { get; set; }
}
```

```
// create a class which implements the IStorable interface
public class Document : IStorable
{
   public Document(string s)
   {
      Console.WriteLine("Creating document with: {0}", s);
   }

   // implement the Read method
   public void Read(  )
   {
      Console.WriteLine(
         "Implementing the Read Method for IStorable");
   }
   // implement the Write method
   public void Write(object o)
   {
```

```
Console.WriteLine(
        "Implementing the Write Method for IStorable");
  }

  // implement the property
  public int Status

  {
     get

     {

        return status;

     }

     set

     {

        status = value;

     }

  }


  // store the value for the property

  private int status = 0;
```

```
}

// Take our interface out for a spin
public class Tester
{

   static void Main(  )
   {
      // access the methods in the Document object
      Document doc = new Document("Test Document");
      doc.Status = -1;
      doc.Read(  );
      Console.WriteLine("Document Status: {0}", doc.Status);
   }
}
```

Combining Interfaces

```
//Example 08-02: Extending and combining interfaces

using System;

interface IStorable
{
   void Read();
   void Write(object obj);
   int Status { get; set; }

}

// here's the new interface
interface ICompressible
```

```
{
   void Compress();
   void Decompress();
}

// Extend the interface
interface ILoggedCompressible : ICompressible
{
   void LogSavedBytes();
}

// Combine Interfaces
interface IStorableCompressible : IStorable, ILoggedCompressible
{
   void LogOriginalSize();
}
```

```
// yet another interface
interface IEncryptable
{
   void Encrypt();
   void Decrypt();
}

public class Document : IStorableCompressible, IEncryptable
{
   // the document constructor
   public Document(string s)
   {
      Console.WriteLine("Creating document with: {0}", s);

   }

   // implement IStorable
```

```
public void Read()
   {
      Console.WriteLine(
         "Implementing the Read Method for IStorable");
   }

   public void Write(object o)
   {
      Console.WriteLine(
         "Implementing the Write Method for IStorable");
   }
   public int Status
   {
      get
      {
         return status;
      }
```

```
set
      {
         status = value;
      }
   }

   // implement ICompressible
   public void Compress()
   {
      Console.WriteLine("Implementing Compress");
   }
   public void Decompress()
   {
      Console.WriteLine("Implementing Decompress");
   }
```

```
// implement ILoggedCompressible
   public void LogSavedBytes()
   {
      Console.WriteLine("Implementing LogSavedBytes");
   }

   // implement IStorableCompressible
   public void LogOriginalSize()
   {
      Console.WriteLine("Implementing LogOriginalSize");
   }
   // implement IEncryptable
   public void Encrypt()
   {
      Console.WriteLine("Implementing Encrypt");

   }
```

```
 public void Decrypt()
    {
       Console.WriteLine("Implementing Decrypt");

    }

    // hold the data for IStorable's Status property
    private int status = 0;
}

public class Tester
{
    static void Main()
    {
       // create a document object
       Document doc = new Document("Test Document");
```

```
// cast the document to the various interfaces
      IStorable isDoc = doc as IStorable;
      if (isDoc != null)
      {
         isDoc.Read();
      }
      else
         Console.WriteLine("IStorable not supported");
      ICompressible icDoc = doc as ICompressible;
      if (icDoc != null)
      {
         icDoc.Compress();
      }
      else
         Console.WriteLine("Compressible not supported");
      ILoggedCompressible ilcDoc = doc as ILoggedCompressible;
```

```
if (ilcDoc != null)
     {
        ilcDoc.LogSavedBytes();
        ilcDoc.Compress();
        // ilcDoc.Read();
     }
     else
        Console.WriteLine("LoggedCompressible not supported");

     IStorableCompressible isc = doc as IStorableCompressible;
     if (isc != null)
     {
        isc.LogOriginalSize();  // IStorableCompressible
        isc.LogSavedBytes();    // ILoggedCompressible
        isc.Compress();         // ICompressible
        isc.Read();             // IStorable
     }
```

```
else
        {
           Console.WriteLine("StorableCompressible not supported");
        }

        IEncryptable ie = doc as IEncryptable;
        if (ie != null)
        {
           ie.Encrypt();
        }
        else
           Console.WriteLine("Encryptable not supported");
    }
}
```

Accessing Interface Methods

- It is common practice to cast a class to one of its interfaces and then invoke the methods of the interface.

```
public class Foo: ITry
{...}

Foo myclass =  new Foo();
ITry mytry = (ITry) myclass;
mytry.InterfaceMethod();
```

Dangers of Casting

- If the class does not implement the interface the cast will NOT generate a compiler error.
- When the method is called an exception will be thrown.
- This is not desirable in most cases

Is and As Operators

- Is can be used to determine if a class implements an interface.
- As combines a test and a cast.
- If the class does not implement the interface then the *as* operator results in a null reference.

Examples

```
MyClass c = new MyClass();
IMyinterface mi;
if (c is IMyinterface) {
   mi = (IMyinterface) c;
   mi.IMethod();
}

mi = c as IMyinterface;
if(mi != null) mi.IMethod();
```

Interfaces - Advanced Features

- Overriding interface implementations.
- Explicit implementations. Required when two or more interfaces have a name conflict for one or more methods.
- Member hiding.
- Use of interfaces with value types.
- Consult a good text for examples of these advanced features.

Indexers

- An indexer is a technique to make a class act like an array.
- A similar technique in C++ is to overload the [] operator.
- Consider a class that encapsulates an array of some simple type and adds functionality to make it appear as a more sophisticated type, e.g., a sparse array class.

Indexers - Example

```
// Example 09-09: Using a simple indexer

namespace Programming_CSharp
{
   using System;

   // a simplified ListBox control
   public class ListBoxTest
   {
      // initialize the list box with strings
      public ListBoxTest(params string[] initialStrings)
      {
         // allocate space for the strings
         strings = new String[256];
```

```
// copy the strings passed in to the constructor
        foreach (string s in initialStrings)
        {
           strings[ctr++] = s;
        }
     }

     // add a single string to the end of the list box
     public void Add(string theString)
     {
        if (ctr >= strings.Length)
        {
           // handle bad index
        }
        else
           strings[ctr++] = theString;
     }
```

```
// allow array-like access
     public string this[int index]
     {
        get
        {
           if (index < 0 || index >= strings.Length)
           {
              // handle bad index
           }
           return strings[index];
        }
        set
        {
           // add only through the add method
           if (index >= ctr )
           {
              // handle error
           }
```

```
else
                strings[index] = value;
        }
    }

    // publish how many strings you hold
    public int GetNumEntries()
    {
        return ctr;
    }

    private string[] strings;
    private int ctr = 0;
}

public class Tester
{
```

```
static void Main()
     {
        // create a new list box and initialize
        ListBoxTest lbt =
           new ListBoxTest("Hello", "World");

        // add a few strings
        lbt.Add("Who");
        lbt.Add("Is");
        lbt.Add("John");
        lbt.Add("Galt");

        // test the access
        string subst = "Universe";
        lbt[1] = subst;
```

```
// access all the strings
          for (int i = 0;i<lbt.GetNumEntries();i++)
          {
             Console.WriteLine("lbt[{0}]: {1}",i,lbt[i]);
          }
       }
    }
}
```

Delegates

- Similar in concept to pointers to functions used with C and C++.
- Allows a generic function prototype to be declared such that an actual function can be used any place the delegate is specified.
- Interfaces can serve a similar purpose, but are less elegant and are designed for multiple functions.
- Most often used with *events*.

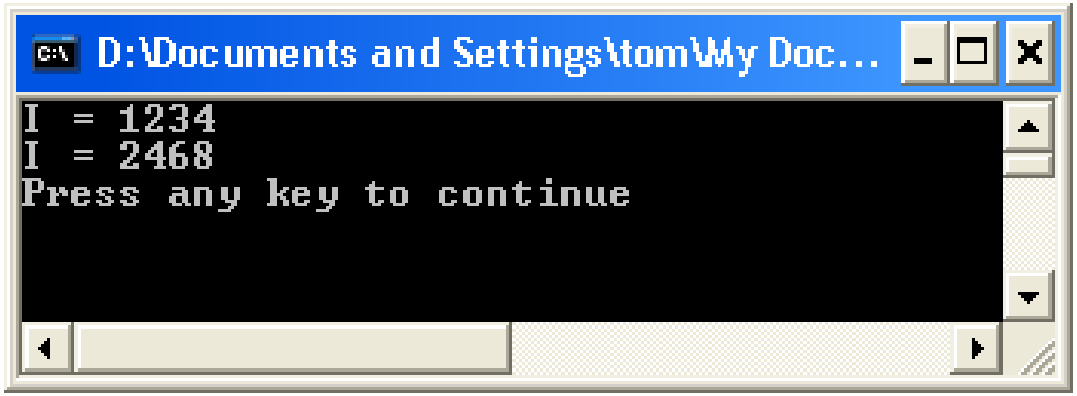
```
using System;

namespace DelegTest
{
    public delegate void Foo(int i);

    class MyTest
    {
        public static void MyFoo(int i)
        {
            Console.WriteLine("I = {0}", i);
        }
        public static void MyFooToo(int i)
        {
            Console.WriteLine("I = {0}", 2*i);
        }

        static void Main(string[] args)
        {
            MyTest t = new MyTest();
            Foo f = new Foo(MyFoo);
            f += new Foo(MyFooToo);    //adds a delegate to chain
            f(1234);
        }
    }
}
```

Output


D:\Documents and Settings\tom\My Doc...
I = 1234
I = 2468
Press any key to continue

Events

- Events are central to Windows programming.
- An event is a special type of delegate that allows one object to activate a handler in another object.
- .NET provides event delegates for the WIN32 API windows messages, e.g. WM_PAINT.

```
this.menuItem2.Index = 0;
this.menuItem2.Text = "Fire";
this.menuItem2.Click += new System.EventHandler(this.menuItem2_Click);
- - - -
   private void menuItem2_Click(object sender, System.EventArgs e)
   {
   }
```

- Click is a predefined event in the menu class associated with clicking on the menu item.
  public event EventHandler Click;
- System.EventHandler is a general purpose delegate for event handling.