ETH Zürich
Chair of Software Engineering

C# Programming in Depth

Prof. Dr. Bertrand Meyer

March 2007 – May 2007

Lecture 2: C# Fundamentals

Lisa (Ling) Liu

Overview

- Simple example
- Comment
- Namespace
- Class and instance
- Common Type System
- Boxing and Unboxing
- Control Statements

Example

```
//=======================================================
// File: HelloWorld.cs
// This program prints a string called "Hello, World!"
//=======================================================

using System;

namespace MyApp
{
    class HelloWorld
    {
        static void Main (string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

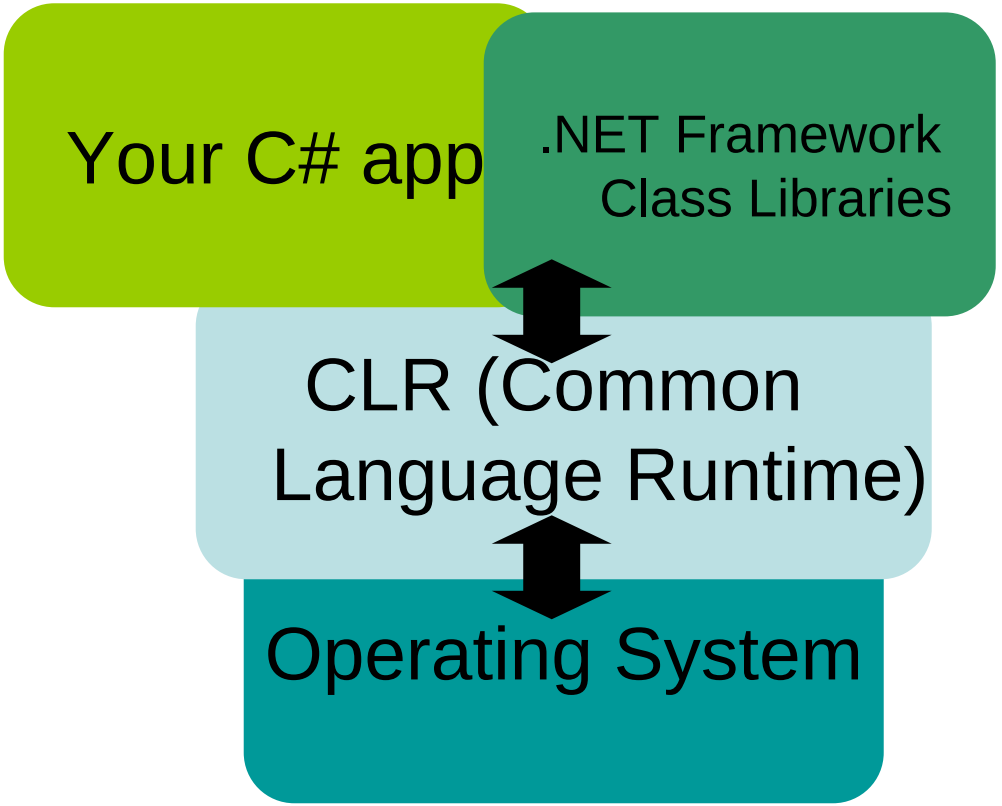
program specifications

library imports

class and namespace definitions

Enter C#

- A hybrid language incorporating features from C++ and Java (and Smalltalk, and…)
- Looks a lot like Java, with keywords from C/C++
- Object oriented
- Has a virtual machine, and garbage collection, among other Java parallels



Comment

- C/C++ comments:
  - //
  - /*...*/

```
//================
// File: HelloWorld.cs
// prints "Hello, World!"
//================
```

- Comments making use of XML elements
  - ///
  - /** ...*/

```
///<summary>
/// File: HelloWorld.cs
/// prints "Hello, World!"
///</summary>
```

- Using C# compiler to genreate document

csc /doc:XmlHello.xml HelloWorld.cs

Namespace

A namespace in C# is a collection of associated types.

- Make use of existing namespaces (packages, libraries or APIs)

```
using System;
```

- Define custom namespaces

```
namespace MyClasses
{
    class MyClass1
    {
        ...
    }
}
```

Declare a class

Assume a class "C" is defined in namespace "N":

- unqualified form
  using N;
  C object_c;

- qualified form
  N.C object_c;

Class and instance

- Define Classes
  - A class is a definition for a user-defined type (UDT)

- Create Instances
  - use "new" keyword

```
MyClass c = new MyClass();
```

Access modifier

C# Access Modifier	Meaning in Life
public	Marks a member as accessible from an object variable as well as any derived classes
private	Marks a method as accessible only by the class that defined the method. In C#, all members are private by default.
protected	Marks a member as usable by the defining class, as well as any derived classes. Protected methods, however, are not accessible from an object variable.
internal	Defines a method that is accessible by any type in the same assembly, but not outside the assembly.
protected internal	Defines a method whose access is limited to the current assembly or types derived from the defining class in the current assembly.

Constructors

- public default constructor
  - provided automatically
  - no arguments
  - ensure all member data is set to an appropriate default value (contrast to C++, where uninitialized state data points to garbage)
  - once you define a custom constructor , the free constructor is removed!

```
class MyClass
{
        string myMsg;
        public MyClass (string msg)
        {
            myMsg = msg;
        }

}
```

```
class MyApp
{
        MyClass c;
        public Main (string[] args)
        {
            c = new MyClass();
        }

}
```

error 1: No overload for method MyClass takes ‘0’ argument.

Is that a Memory Leak?

- never destroy a managed object explicitly
- .NET garbage collector frees the allocated memory automatically
- C# does not support a delete keyword

Constructor definition

- named identically to the class under construction
- never provide a return value (not even void)
- can provide access modifier

```
class HelloClass
{
        HelloClass()
        {
            Console.WriteLine("Default");
        }
        ...
}
```

private constructor

- It is commonly used in classes that contain static members only.

- If a class has one or more private constructors and no public constructors, then other classes (except nested classes) are not allowed to create instances of this class.

```
public class Counter
{
        private Counter() { }
        public static int currentCount;
        public static int IncrementCount()
        {
                return ++currentCount;
        }
}

class TestCounter
{
        static void Main()
        {
                Counter.currentCount = 100;
                Counter.IncrementCount();
                System.Console.WriteLine("New count: {0}",
                Counter.currentCount);
        }
}
```

Class-leve and Instance-level members

- Class-Level Members (Defined using static keyword)
  - Class Fields
  - Class Methods
  - Class Constructors
- Instance-Level Members
  - Instance Fields
  - Instance Methods
  - Instance Constructors
- static methods can operate only on static class members

static members (class-level members)

- declaring a field or method with the static key word, tells the compiler that the field or method is associated with the class itself, not with instances of the class.

- static or "class" fields and methods are global variables and methods that you can access using the class name.

```
class TestCounter
{
        static void Main()
        {
                Counter.currentCount = 100;
                Counter.IncrementCount();
                System.Console.WriteLine("New count: {0}",
                Counter.currentCount);
        }
}
```

static members ...

- There is only one copy of the static fields and methods in memory, shared by all instances of the class
- static fields are useful when you want to store state related to all instances of a class
- static methods are useful when you have behavior that is global to the class and not specific to an instance of a class

```
class A
{
    public          int x;
    public void Increase()
    {
        x = x+1;
    }
}
```

```
class program
{
    static void Main (string[] args)
    {
        A a1 = new A();
        a1.Increase;
        A a2 = new A();
        a2.Increase;
        Console.WriteLine(a1.x);
    }
}
```

```
class A
{
    public static int x;
    public void Increase()
    {
        x = x+1;
    }
}
```

```
class program
{
    static void Main (string[] args)
    {
        A a1 = new A();
        a1.Increase;
        A a2 = new A();
        a2.Increase;
        Console.WriteLine(A.x);
    }
}
```

How to initialize the values of static fields?

static constructor

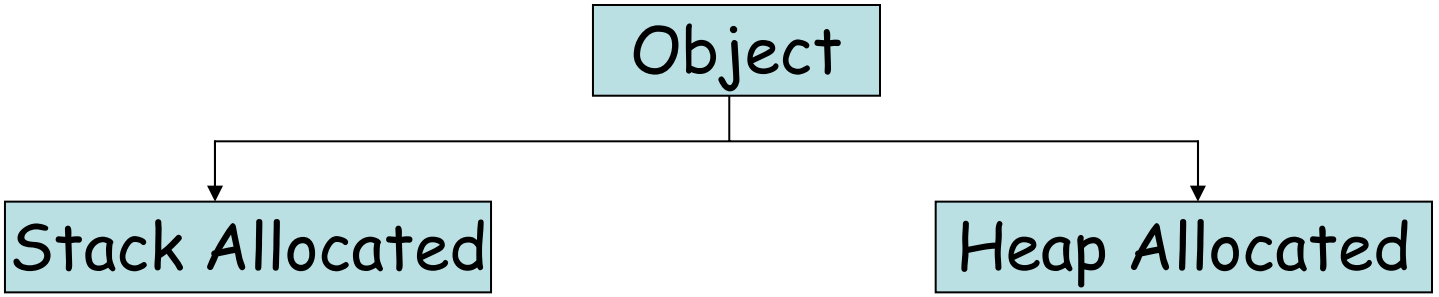
```
class A
{
        public static  int x;
        static A ( )
        {
                x = 0;
        }
        public void Increase()
        {
                x = x+1;
         }
}
```

How to use static constructor?

Notes regarding Static Constructor

- A given class (or structure) may define only a single static constructor.
- A static constructor executes exactly one time, regardless of how many objects of the type are created
- A static constructor does not take an access modifier and cannot take any parameters.
- The runtime invokes the static constructor when it creates an instance of the class or before accessing the first static member invoked by the caller.
- The static constructor executes before any instance-level constructors.

Data Types



- Value Types
  - Primitives
  - Enumerations
  - Structures
- Deallocated when defining blocks exits

- Reference Types
  - Classes
  - Interfaces
  - Arrays
  - Delegates
  - String
- Garbage collected

C# Primitive Types and System Types

C# Primitive Type	System Type	CLS Compliant
sbyte	System.SByte	No
byte	Sytem.Byte	Yes
short	System.Int16	Yes
ushort	System.UInt16	No
int	System.Int32	Yes
uint	System.UInt32	No
long	System.Int64	Yes
ulong	System.UInt64	No
char	System.Char	Yes
float	System.Single	Yes
double	System.Double	Yes
bool	System.Boolean	Yes
decimal	System.Decimal	Yes
string	System.String	Yes
object	System.Object	Yes

Default values of variables

- Class member variables:
  - bool: false
  - Numeric type: 0 or 0.0
  - string: null
  - char: '\0'
  - Reference type: null
- Local variables:
  - local variables must be initialized by using them

Struct

- Structs are defined using the struct keyword
- A struct type is a value type that is suitable for representing lightweight objects such as Point, Rectangle, and Color
- Structs can declare constructors, but they must take parameters
- Structs can implement an interface but they cannot inherit from another struct. For that reason, struct members cannot be declared as **protected**
- Structs can also contain constructors, constants, fields, methods, properties, indexers, operators, events, and nested types, although if several such members are required, you should consider making your type a class instead

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

```
class TestCoOrds
{
    static void Main()
    {
        CoOrds coords1 = new CoOrds();
        CoOrds coords2 = new CoOrds(10, 10);
        CoOrds coords3;
        coords3.x = 10;
        coords3.y = 20;
    }
}
```

Enumerations

```
// A custom enumeration

enum EmpType

{

        Manager,        // = 0
        Grunt,          // = 1
        Contractor,     // = 2
        VP              // = 3

}
```

```
// Begin numbering at 102.

enum EmpType

{

        Manager = 102,
        Grunt,          // = 103
        Contractor,     // = 104
        VP              // = 105

}
```

```
// Elements of an enumeration need
//  not be sequential

enum EmpType : byte

{

        Manager = 10,
        Grunt = 1,
        Contractor = 100,
        VP = 9

}
```

By default, the storage type for each item in an enumeration maps to System.Int32

Enumerations – bit fields

- A enumeration type can be treated as a set of bit fields with attribute FlagsAttribute
- Bit fields are generally used for lists of elements that might occur in combination, whereas enumeration constants are generally used for lists of mutually exclusive elements.

```
enum SingleHue : short
{
        Black = 0,
        Red = 1,
        Green = 2,
        Blue = 4
 };
// Define an Enum with FlagsAttribute.
[FlagsAttribute]
enum MultiHue : short
{
      Black = 0,
      Red = 1,
      Green = 2,
      Blue = 4
 };
```

```
static void Main( )
 {
        Console.WriteLine(
           "\nAll possible combinations of values of an \n" +
           "Enum without FlagsAttribute:\n" );

        // Display all possible combinations of values.
        for( int val = 0; val <= 8; val++ )
           Console.WriteLine( "{0,3} - {1}",
              val, ( (SingleHue)val ).ToString( ) );

        Console.WriteLine(
           "\nAll possible combinations of values of an \n" +
           "Enum with FlagsAttribute:\n" );

        // Display all possible combinations of values.
        // Also display an invalid value.
        for( int val = 0; val <= 8; val++ )
           Console.WriteLine( "{0,3} - {1}",
              val, ( (MultiHue)val ).ToString( ) );
}
```

System.Enum base class

- .NET enumerations are implicitly derived from System.Enum
- A .NET enumeration type is a value type
- Selected static members of System.Enum
  - Format
  - GetName
  - GetNames
  - GetValues
  - IsDefined
  - Parse

System.Object

- All classes in the .NET Framework are derived from **Object**, every method defined in the **Object** class is available in all objects in the system, including:
  - Equals - Supports comparisons between objects.
  - Finalize - Performs cleanup operations before an object is automatically reclaimed.
  - GetHashCode - Generates a number corresponding to the value of the object to support the use of a hash table.
  - ToString - Manufactures a human-readable text string that describes an instance of the class.

Note: if you override Equals () you should also override GetHashCode ()

```
using System;

// The Point class is derived from System.Object.
class Point
{
    public int x, y;
    public Point (int x, int y)
    {
                ...
    }

    public override bool Equals (object obj)
    {
        // If this and obj do not refer to the same type, then they are not equal.
        if (obj.GetType() != this.GetType()) return false;

        // Return true if  x and y fields match.
        Point other = (Point) obj;
        return (this.x == other.x) && (this.y == other.y);
    }

    // Return the XOR of the x and y fields.
    public override int GetHashCode()
    {
        return x ^ y;
    }

    // Return the point's value as a string.
    public override String ToString()
    {
        return String.Format("({0}, {1})", x, y);
    }
}
```

System.String

- string: shorthand for System.String
- Even though string is a reference type, the equality operators "==" and "!=" are defined to compare the value with the string objects
- The value of a string cannot be modified once established. Thus modifying a string in fact return a new object containing the modification

.NET data types provide the ability to parse a string to corresponding value

```
Bool myBool = bool.Parse ("True");
int myInt = int.Parse ("8");
char myChar = char.Parse ("w");
```

Escape characters

Character	Meaning
\'	Insert a single quote
\"	Insert a double quote
\\	Insert a backslash
\a	Triggers a system alert (beep)
\n	Insert a new line
\r	Insert a carriage return
\t	Insert a horizontal tab

Verbatim strings

- The @-prefixed string is called verbatim string, which is used to disable the processing of escaped characters in the string

For example:

```
Console.WrteLine (@"c:\My Documents\My Videos");
Console.WriteLine (@"This is a ""value-type"" variable!");
```

System.Text.StringBuilder

- This class represents a string-like object whose value is a mutable sequence of characters

```
using System.Text;
…
StringBuilder myBuffer = new StringBuilder (“my string”);
myBuffer = myBuffer.Append (“contains some characters.”);
```

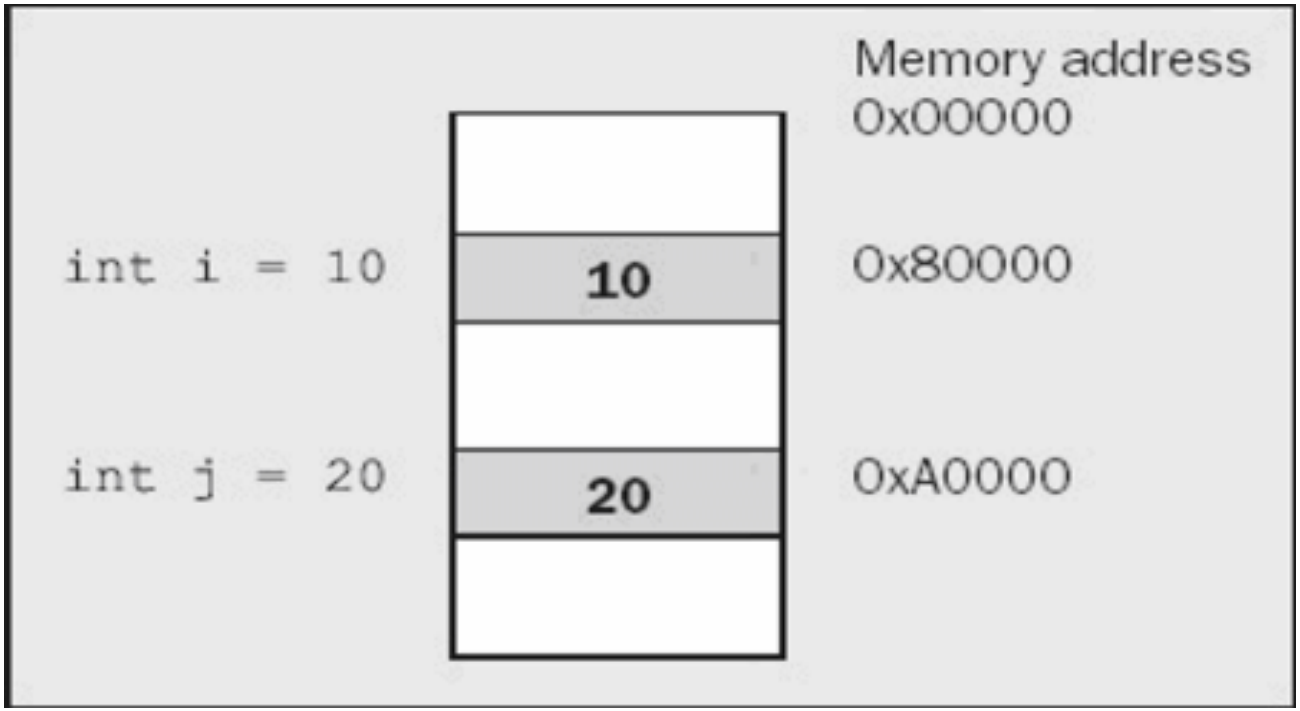
.NET Array types

- Arrays are references and derive from the common base class System.Array
- By default, arrays always have a lower bound zero
- Elements in an array are automatically set to their default values unless you indicate otherwise
- Declare an array

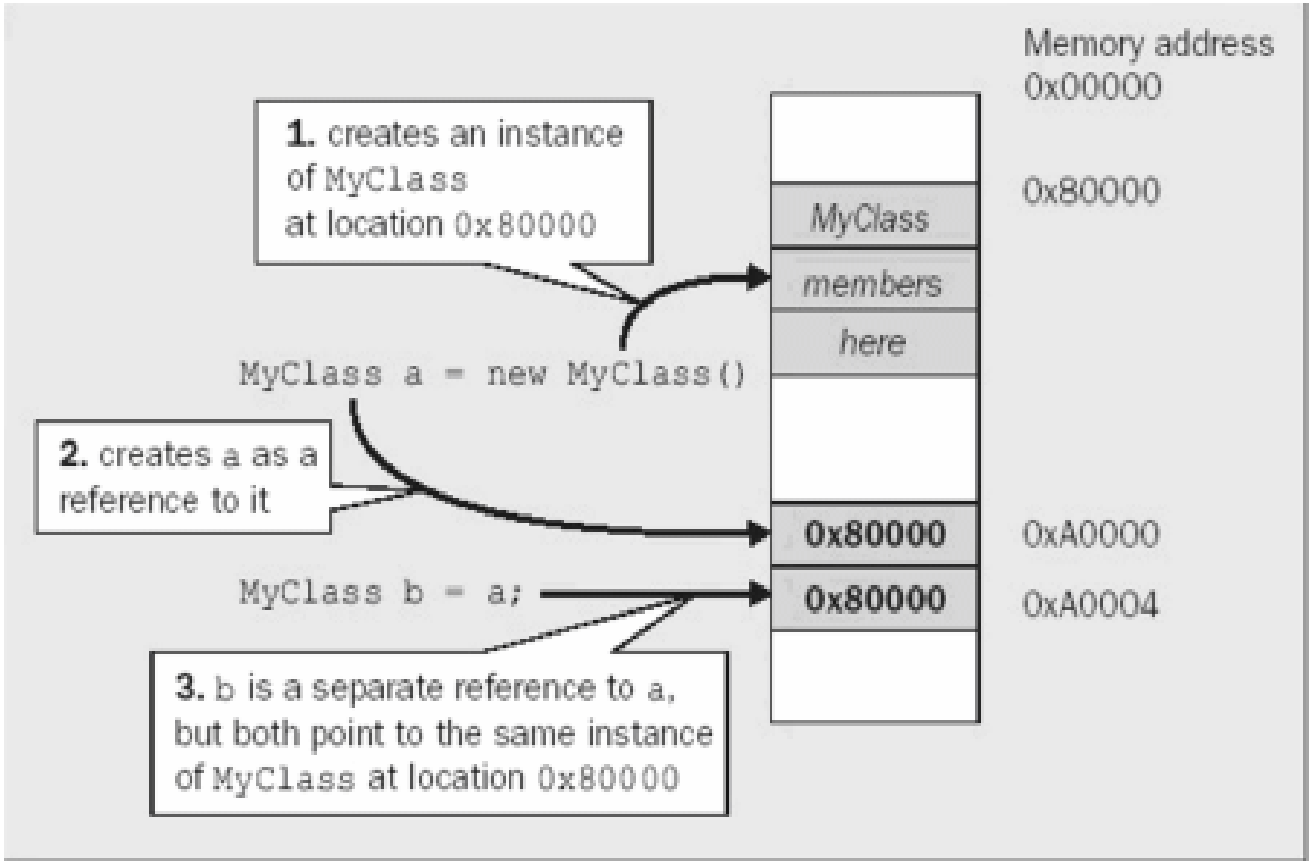
```
string[] books = new string[3];
int[] n2= new int[] {20, 22, 23, 0};
int[] n3 = {1, 2, 3, 4, 5};
int[,]  matrix = new int[5,5];

int[][] jagArray = new int[5][];
for (int i=0; i<jagArray.Length; i++)
   jagArray[i] = new int[i+7];
```

Memory locations for value types



Memory locations for reference type



Method parameter modifies

Parameter Modifier	Meaning in Life
(none)	If a parameter is not marked with a parameter modifier, it is assumed to be passed by value, meaning the called method receives a copy of the original data.
out	Output parameters are assigned by the method being called (and therefore passed by reference). If the called method fails to assign output parameters, you are issued a compiler error.
params	This parameter modifier allows you to send in a variable number of identically typed arguments as a single logical parameter. A method can have only a single params modifier, and it must be the final parameter of the method.
ref	The value is initially assigned by the caller, and may be optionally reassigned by the called method (as the data is also passed by reference). No compiler error is generated if the called method fails to assign a ref parameter

out Modifier

- variables passed as output variables are not required to be assigned before use.
- allows the caller to obtain multiple return values from a single method invocation.

ref Modifier

- ref parameters don't pass the values of the variables used in the function member invocation - they use the variables themselves

- difference between ref and out
  - out: actual output parameters do not need to be initialized before they are passed to the method
  - ref: actual reference parameters must be initialized before they are passed to the method

- Although ref and out are treated differently at run-time, they are treated the same at compile time

```
class RefOut_Example
{
    // compiler error CS0663: "cannot define overloaded
    // methods that differ only on ref and out"
    public void SampleMethod (ref int i) {  }
    public void SampleMethod (out int i) {  }
}
```

params Modifier

- a parameter that can be passed a set of identically typed arguments

```
static int Add (ref int x, int y, out int sum, params int[] a)
{
        sum = x+y;
        x = -1;
        y = -2;
        foreach (int i in a)
                sum = sum + i;
        return sum;
}
```

```
static void Main ()
{
        int param1, param2, ant;
        param1 = 100;
        param2 = 200;
        Console.WriteLine ("sum = {0}, param1 = {1}, param2={2}", Add( ref
param1, param2, out ant, 1,2,3), param1, param2);
}
```

Assignment Operator

Simple Assignment

- value types: copy value
- reference types: copy reference

Value Types Containing Reference Types

- assignment results in a copy of the references

Parameter

By default

- value type: passed by value
- reference type: passed by reference

Passing Reference Types by Value

- may change the values of the object's state
- cannot reassign the object reference

Passing Reference Types by Reference

- the callee may change the values of the object's state data as well as the object it is referencing.

Boxing and Unboxing

Boxing

- convert a value type to a reference type

Unboxing

- convert the value held in the object reference back into a corresponding value type
  - begin by verifying that the receiving data type is equivalent to the boxed type

```
int       v = 5;
object    o = b;          //Box v
int       i = (int) o;    //Unbox o; type must match
```

Practical (Un)Boxing examples

- C# compiler automatically boxes variables when appropriate
- Boxing and unboxing types takes some processing time

```
public class System.Collections.ArrayList :
...
{
    ...
    public virtual int Add (object value);
    ...
}
```

```
staic void Main ()
{
    ...
    ArrayList myInts = new ArrayList ();
    myInts.Add (88);
    myInts.Add (3.33);
    ...
    int firstItem = (int) myInts[0];
}
```

Control Statements

- Decision Costructs
  - if / else statement
  - switch statement
- Iteration Contructs
  - for loop
  - foreach loop
  - while loop
  - do/while loop

Relational and logic operaors

- Relational operators:
  - ==, !=, <, >, <=, >=
- Logical operators:
  - &&, ||, !

if / else Statement

```
string thoughtOfTheDay = “You can study
at any time!”;

if (thoughtOfTheDay.Length() != 0)
{
        ...
}

else

{

    ...

}
```

Note: no elsif

Switch Statement

```
string langChoice = Console.ReadLine();
switch (langChoice)
{                                   // can evaluate string expression
   case "C#":
      Console.WriteLine("Good choice, C# is a fine language.");
      break;
   case "VB":                       // each case must have a terminal break or goto
      Console.WriteLine("VB .NET: OOP, multi-threading and more!");
      break;
   default:
      Console.WriteLine("Well...good luck with that!");
      break;
}
```

for Loop

```
for (int i = 0; i < 10; i++)
{
        Console.WriteLine("Number is: {0} ", i);
 }
 // 'i' is not visible here.
```

foreach Loop

```
string[] books = {"Complex Algorithms",
                  "Do you Remember Classic COM?",
                  "C# and the .NET Platform"};
foreach (string s in books)
{ Console.WriteLine(s); }
```

while Loop

```
string userIsDone = "no";
while (userIsDone != "yes")
{
        Console.Write("Are you done? [yes] [no]: ");
         userIsDone = Console.ReadLine();
         Console.WriteLine("In while loop");
}
```

do / while Loop

```
string ans;
do
{
        Console.WriteLine("In do/while loop");
        Console.Write("Are you done? [yes] [no]: ");
        ans = Console.ReadLine();
} while (ans != "yes");
```

Questions?