Data Types and Control Flow

C# Programming

Casting and Type Conversions

- C# is statically-typed at compile time, after a variable is declared.
- It cannot be declared again or used to store values of another type unless that type is convertible to the variable's type.
- In C#, allows the following conversions:
  - Implicit conversions
  - Explicit conversions
  - User-defined conversions
  - Conversions with helper classes

Casting

- Sometimes we need to copy a value into a variable or method parameter of another type, such operations are called Type Conversions or Casting.
- If Right Hand Side expression and Left Hand Side variable are not of same data type then casting is required. So that the data can be converted from one form to another form and it is called as casting.
- In general there are two types of casting:
  1. Implicit
  2. Explicit

Implicit / Explicit

- Implicit Casting: If every possible value of RHS expression is valid for a variable on LHS variable the casting is done implicitly.
  - int n=10;
  - byte b=2;
  - n = b; // Valid implicit casting is done
  - b = n; // invalid required explicit casting.
- Explicit Casting: If a RHS expression is assigned to LHS and if there is a possibility of data loss then explicit casting is required.

What can convert implicitly?

- Unlike to 'C' here casting is done based on range, not based on size of the data type
  - sbyte → short, int, long, float, double, decimal
  - char →int, long, float, double, decimal, ushort, uint, ulong
  - byte → short, ushort, int, uint, long, ulong, float, double, decimal
  - short → int, long, float, double, decimal
  - ushort → int, uint, long, ulong, float, double, decimal
  - int → long, float, double, decimal
  - uint → long, ulong, float, double, decimal
  - ulong, long → float, double, decimal
  - float → double

User-defined conversions

- User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship.
- C# enables programmers to declare conversions on classes or structs so that classes or structs can be converted to and/or from other classes or structs, or basic types.
- Conversions are defined like operators and are named for the type to which they convert.

Presented by

Ranjan Bhatnagar

User-Defined Conversions Example

```
using System;
namepace CSProgExample
{
    struct RomanNumeral
    {
        private int value;
        public RomanNumeral(int value)  //constructor
        {
            this.value = value;
        }
        static public implicit operator RomanNumeral(int value)
        {
            return new RomanNumeral(value);
        }
        static public implicit operator RomanNumeral(BinaryNumeral binary)
        {
            return new RomanNumeral((int)binary);
        }
        static public explicit operator int(RomanNumeral roman)
        {
            return roman.value;
        }
        static public implicit operator string(RomanNumeral roman)
        {
            return ("Conversion to string is not implemented");
        }
    }
```

User-Defined Conversions Example cont.

```
struct BinaryNumeral
    {
        private int value;
        public BinaryNumeral(int value)  //constructor
        {
            this.value = value;
        }
        static public implicit operator BinaryNumeral(int value)
        {
            return new BinaryNumeral(value);
        }
        static public explicit operator int(BinaryNumeral binary)
        {
            return (binary.value);
        }
        static public implicit operator string(BinaryNumeral binary)
        {
            return ("Conversion to string is not implemented");
        }
    }
```

User-Defined Conversions Example cont.

```
class TestConversions
    {
        static void Main()
        {
            RomanNumeral roman;
            BinaryNumeral binary;
            roman = 10;
            // Perform a conversion from
            //a RomanNumeral to a BinaryNumeral:
            binary = (BinaryNumeral)(int)roman;
            // Perform a conversion from
            // a BinaryNumeral to a RomanNumeral:
            // No cast is required:
            roman = binary;
            Console.WriteLine((int)binary);
            Console.WriteLine(binary);
            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
        }
    }
}
```

Conversions with helper classes

- To convert between non-compatible types, such as **integers** and **System.DateTime** objects, or hexadecimal **strings** and **byte arrays**, you can use the **System.BitConverter** class, the **System.Convert** class, and the Parse methods of the built-in numeric types, such as **Int32.Parse**.
- Here we will see few examples for:
  - Convert a byte Array to an int
  - Convert a string to an int
  - Convert Between Hexadecimal Strings and Numeric Types

Convert a byte Array to an int

```
using System;
namespace CSProgExample
{
    class Program
    {
        static void Main(string[] args)
        {
            byte[] bytes = { 0, 0, 0, 25 };
            // If the system architecture is little-endian
            //(that is, little end first),
            // reverse the byte array.
            if (BitConverter.IsLittleEndian)
                Array.Reverse(bytes);
            int i = BitConverter.ToInt32(bytes, 0);
            Console.WriteLine("int: {0}", i);
            // Output: int: 25
        }
    }
}
```

Convert a string to an int

```
using System;
namespace CSProgExample
{
    class Program
    {
        static void Main(string[] args)
        {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
            int j;
            bool result = Int32.TryParse("-105", out j);
            if (true == result)
                Console.WriteLine(j);
            else
                Console.WriteLine("String could not be parsed.");
            string inputString = "abc";
            int numValue;
            bool parsed = Int32.TryParse(inputString, out numValue);
            if (!parsed)
                Console.WriteLine("Int32.TryParse could not parse
'{0}' to an int.\n", inputString);
        }
    }
}
```

Presented by

Ranjan Bhatnagar

Convert Between Hex Strings and Num Types

```
using System;
namespace CSProgExample
{
    class Program
    {
        static void Main(string[] args)
        {
            string input = "Hello World!";
            char[] values = input.ToCharArray();
            foreach (char letter in values)
            {
                int value = Convert.ToInt32(letter);
                string hexOutput = String.Format("{0:X}", value);
                Console.WriteLine("Hexadecimal value of {0} is {1}",
                                        letter, hexOutput);
            }

            byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };
            string str = BitConverter.ToString(vals);
            Console.WriteLine(str);
            str = BitConverter.ToString(vals).Replace("-", "");
            Console.WriteLine(str);
        }
    }
}
```

Type Safety

- **Initialize before use**

```
int i ;
Console.WriteLine ( i ) ;
```

- **Initial value must be within range**

```
short s = 40000 ;
```

- **Larger destination type**

```
short s = 40 ;
byte b ;
b = s ;
```

- **Boolean type is incompatible**

```
int i = 40 ;
bool b = true ;
i = b ;
b = ( bool ) i ;
```

- **sbyte and short are converted to int**

```
short s1 = 40, s2 ;
byte b = 6 ;
s2 = s2 + s1 ;
```

C# imposes some rules

- **Rule 1:** Before using any variable it must be initialized with a value. Unlike C/C++, in C# there is no concept of garbage values.
- **Rule 2:** The variable can be initialized only with a value that is within the range of its type.
- **Rule 3:** While assigning, the destination type must be larger than the source type. Narrowing conversions are not supported implicitly by C#.
- **Rule 4:** sbyte and short are converted to an int type, whereas, byte and ushort are converted to a uint type while performing arithmetic operations on them. Assigning an integer value to a short variable would result in an error.
- **Rule 5:** Boolean type is incompatible with rest of the data types. The bool type is returned by all relational operations such as a < b. The bool is also a type required by the conditional expressions that govern the control statements such as if, while and for. It solves the problem where programmers mistakenly use = instead of == resulting in unexpected behavior.

Checked v/s Unchecked

- What would be an output?

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int n = 256;
        byte b;
        b = (byte)n;
        Console.WriteLine(b);
    }
}
```

- Overflow/Underflow checks depends on project properties.

Checked Block

- Overflow/Underflow checks are done irrespective of project properties

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int n = 256;
        byte b;
        checked
        {
            b = (byte)n;
            Console.WriteLine(b);
        }
    }
}
```

Unchecked Block

- Overflow/Underflow checks not done irrespective of project properties

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int n = 256;
        byte b;
        unchecked
        {
            b = (byte)n;
            Console.WriteLine(b);
        }
    }
}
```

Presented by

Ranjan Bhatnagar

Convert

- Note that conversions of numeric value to string and vice versa, bool to string and vice versa etc. is not possible either implicitly or explicitly.
- This can be done through the methods below:
- Example:

```
/* Boolean values */
            bool ii = Convert.ToBoolean(10);
            Console.WriteLine(ii);  //True
            ii = Convert.ToBoolean(0.0);
            Console.WriteLine(ii); //False
            ii = Convert.ToBoolean("true");
            Console.WriteLine(ii); //True
            ii = Convert.ToBoolean("gg");
            Console.WriteLine(ii);
               // Runtime error called FormatException
```

More Conversions

```
/* String values */
            string s = Convert.ToString(true);
            Console.WriteLine(s); //True
            s = Convert.ToString(1.23);
            Console.WriteLine(s); //1.23
            s = Convert.ToString('A');
            Console.WriteLine(s); //A
            Console.WriteLine(Convert.ToString(null));
            // prints nothing
/* Numeric Values */
            int i = Convert.ToInt32("1009");
            short j = Convert.ToInt16("100");
            long k = Convert.ToInt64("9292929");
            byte b = Convert.ToByte("20");
            //No ToFloat method!
            double d = Convert.ToDouble("sss");
             // runtime error
```

Parse

- It is another way to convert a data type.
- Parse method converts the string representation to its equivalent using specific data type.
- Int32.Parse() converts the string representation of a number to its 32-bit signed integer equivalent.
- Int32.TryParse() converts the string representation of a number to its 32-bit signed integer equivalent. A return value indicates whether the conversion succeeded.

Example

```
using System;
public class StringParsing
{
  public static void Main()
  {
    TryToParse(null);
    TryToParse("160519");
    TryToParse("9432.0");
    TryToParse("16,667");
    TryToParse("   -322   ");
    TryToParse("+4302");
    TryToParse("(100);");
    TryToParse("01FA");
  }
```

```
  private static void
    TryToParse(string value)
{
      int number;
      bool result =
     Int32.TryParse(value, out number);
      if (result)
      {
          Console.WriteLine
         ("Converted '{0}' to {1}.",
             value, number);
      }
      else
      {
          if (value == null) value = "";
          Console.WriteLine
         ("Attempted conversion of
         '{0}' failed.", value);
      }
   }
}
```

String.Format

- Using Console.WriteLine() you can format numeric results by using the String.Format method, or through the Console.Write or Console.WriteLine method, which calls String.Format.
- The format is specified by using format strings. The table in next slide contains the supported standard format strings examples.

Examples

Character	Description	Examples	Output
C or c	Currency	Console.Write("{0:C}", 2.5); Console.Write("{0:C}", -2.5);	$2.50 ($2.50)
D or d	Decimal	Console.Write("{0:D5}", 25);	00025
E or e	Scientific	Console.Write("{0:E}", 250000);	2.500000E+005
F or f	Fixed-point	Console.Write("{0:F2}", 25); Console.Write("{0:F0}", 25);	25.00 25
G or g	General	Console.Write("{0:G}", 2.5);	2.5
N or n	Number	Console.Write("{0:N}", 2500000);	2,500,000.00
X or x	Hexadecimal	Console.Write("{0:X}", 250); Console.Write("{0:X}", 0xffff);	FA FFFF

Presented by

Ranjan Bhatnagar

Decision Control

Syntax 1:

```
if (<test>)
    <code executed if <test> is true>;
```

```
If ( a< b)
    a=10;
```

Syntax 2:

```
if (<test>)
    <code executed if <test> is true>;
else
    <code executed if <test> is false>;
```

```
If ( a< b)
    a=10;
else
    b=10;
```

Syntax 3:

```
if (<test>)
{
    <code executed if <test> is true>;
}
else
{
    <code executed if <test> is false>;
}
```

```
If ( a< b)
{
    a=10;
    b=5;
}
else
{
    a=5;
    b=10;
}
```

More on If

- Nesting in if and else can be done

```
if (var1 == 1)
{
    // Do something.
}
else if (var1 == 2)
{
    // Do something else.
}
else if (var1 == 3 || var1 == 4)
{
    // Do something else.
}
else
{
    // Do something else.
}
```

Case Control

```
switch ( grade )
{
   case 'A' :
      C.W ( "In A" ) ;

   case  'B' :
      C.W ( "In B" ) ;

   case 'C' :
      C.W ( "In C" ) ;
}
```

```
switch ( grade )
{
   case 'A' :
      C.W ( "In A" ) ;
      break ;
   case  'B' :
      C.W ( "In B" ) ;
      break ;
   case 'C' :
      C.W ( "In C" ) ;
      break ;
}
```

```
switch ( grade )
{
   case 'A' :
      C.W ( "In A" ) ;
      goto s1:

   case  'B' :
      C.W ( "In B" ) ;
      break ;

   case 'C' :
      s1:
      C.W ( "In C" ) ;
      break ;
}
```

```
case 'a' :
case 'A' :
   Console.WriteLine ( "In A" ) ;
```

Example

```
using System;
class MyClass
{
    static void Main(string[] args)
    {
        int grade = Console.Read();
        switch (grade)
        {
            case 'A':
                Console.WriteLine("In A"); break;
            case 'B':
                Console.WriteLine("In B"); break;
            case 'C':
                Console.WriteLine("In C"); break;
            default:
                Console.WriteLine("In Default");
                break;
        }
    }
}
```

Loop / Repetition Control

- while statement

```
static void Main(string[] args)
{
   int i = 0;
   while (i < args.Length)
   {
      Console.WriteLine(args[i]);
      i++;
   }
}
```

```
//print 1 to 10
int i = 1;
while (i <= 10)
{
  Console.WriteLine(i++);
}
```

- do while statement

```
static void Main()
{
   string s;
   do {
      s = Console.ReadLine();
      if (s != null) Console.WriteLine(s);
   } while (s != null);
}
```

```
//print 1 to 10
int i;
do
{
  Console.WriteLine(i++);
} while (i <= 10);
```

More Loops

- For Loop

```
static void Main(string[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        Console.WriteLine(args[i]);
    }
}
```

```
//print 1 to 10
int i = 1;
for (i = 1; i <= 10; ++i)
{
       Console.WriteLine("{0}", i);
}
```

Presented by

Ranjan Bhatnagar

Break and Continue

- Break

```
static void Main()
{
   while (true)
   {
      string s = Console.ReadLine();
      if (s == null) break;
      Console.WriteLine(s);
   }
}
```

- Continue

```
static void Main(string[] args)
{
   for (int i = 0; i < args.Length; i++)
   {
      if (args[i].StartsWith("/"))
          continue;
      Console.WriteLine(args[i]);
   }
}
```

'foreach' Loop

- The foreach statement lets us iterate over elements in arrays or Collection.

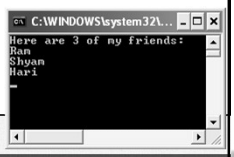
```
using System;
namespace Simple
{
    class Class1
    {
        static void Main(string[] args)
        {
            int[] arr = { 1, 2, 3, 4, 5 };
            foreach (int i in arr)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

Syntax

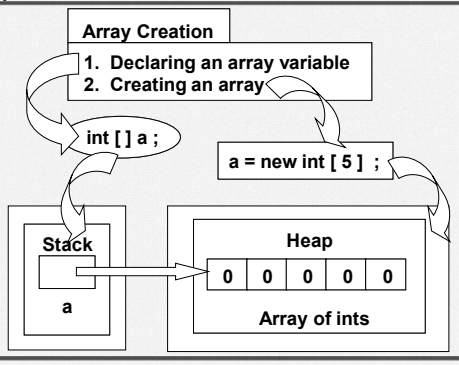
```
foreach (<baseType> <name> in <array>)
{
    // can use <name> for each element
}
```

Another foreach example

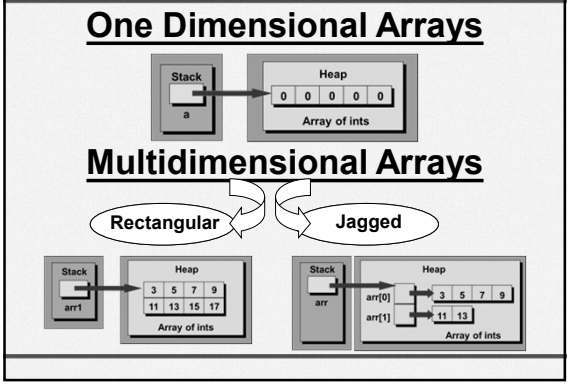
```
using System;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] frNames = { "Ram", "Shyam", "Hari" };
            Console.WriteLine("Total {0} Friends.", frNames.Length);
            foreach (string frName in frNames)
            {
                Console.WriteLine(frName);
            }
            Console.ReadKey();
        }
    }
}
```



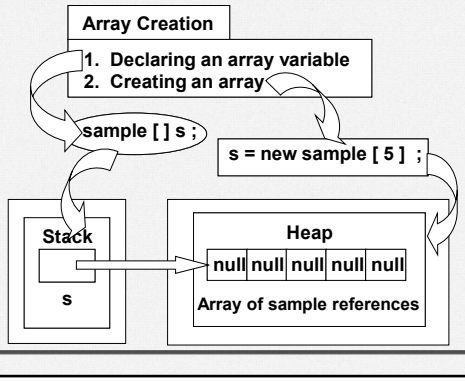
Arrays



Arrays in C#



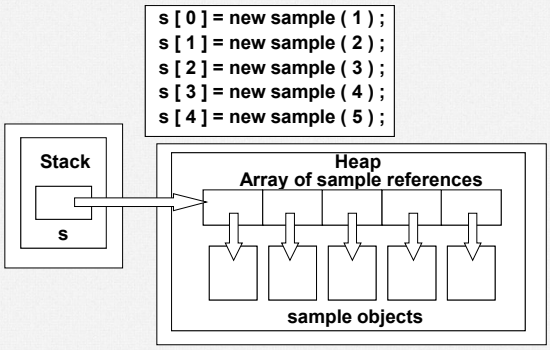
Array Of Objects



Presented by

Ranjan Bhatnagar

Creating Objects



Some Examples

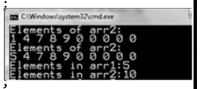
```
//One Dimensional Arrays
   int[] a;
   a = new int[5];
   int[] MyArray;
   MyArray = new int[] { 1, 2, 3, 4, 5 };
   string[] WeekDays = { "Sunday", "Monday", "Tuesday", "Wednesday",
                         "Thursday", "Friday", "Saturday" };

//Multi Dimensional Arrays
   int[,] a2 = new int[10, 5];
   int[, ,] a3 = new int[10, 5, 2];
   int[,] arr1 = new int[,] { { 3, 5, 7, 9 }, { 11, 13, 15, 17 } };
   int[,] arr2;
   arr2 = new int[,] { { 3, 5, 7, 9 }, { 11, 13, 15, 17 } };
   int[, ,] arr3 = new int[4, 2, 3];

//jagged array
   int[][] arr = new int[2][];
   arr[0] = new int[4] { 3, 5, 7, 9 };
   arr[1] = new int[2] { 11, 13 };
   int[][] jarr = new int[][]{ new int[] {1,3,5,7,9},
                               new int[] {0,2,4,6},
                               new int[] {11,22}};
```

Array Class

- The Array class is the base class for language implementations that support arrays.
- It is an abstract class.
- However, only the system and compilers can derive explicitly from the Array class. Users should employ the array constructs provided by the language.
- An array can have a maximum of 32 dimensions.

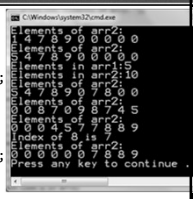
System.Arrays Class

```
using System;
class onedarray
{
    static void Main(string[] args)
    {
        int[] arr1 = new int[5] { 1, 4, 7, 8, 9 };
        int[] arr2 = new int[10];
        arr1.CopyTo(arr2, 0);
        Console.WriteLine("Elements of arr2: ");
        foreach (int i in arr2)
            Console.Write(i + " ");
        Console.WriteLine();
        arr2.SetValue(5, 0);
        Console.WriteLine("Elements of arr2: ");
        foreach (int i in arr2)
            Console.Write(i + " ");
        Console.WriteLine();
        Console.WriteLine("Elements in arr1:" + arr1.Length);
        Console.WriteLine("Elements in arr2:" + arr2.Length);
        Array.Copy(arr1, 2, arr2, 6, 2);
        Console.WriteLine("Elements of arr2: ");
```



System.Arrays Class

```
        foreach (int i in arr2)
            Console.Write(i + " ");
        Console.WriteLine();
        Array.Reverse(arr2);
        Console.WriteLine("Elements of arr2: ");
        foreach (int i in arr2)
            Console.Write(i + " ");
        Console.WriteLine();
        Array.Sort(arr2);
        Console.WriteLine("Elements of arr2: ");
        foreach (int i in arr2)
            Console.Write(i + " ");
        Console.WriteLine();
        Console.WriteLine("Index of 8 is "
        + Array.IndexOf(arr2, 8));
        Array.Clear(arr2, 3, 3);
        Console.WriteLine("Elements of arr2: ");
        foreach (int i in arr2)
            Console.Write(i + " ");
        Console.WriteLine();
    }
}
```



Implicitly Typed Arrays

- C# 3.0 added the ability to declare implicitly typed variables by using the var keyword based on the type of the initializing expression.
- In same fashion it is possible to create an implicitly typed array.
- An implicitly typed array is declared using the keyword var, but you do not follow var with [ ].
- Here is an example of an implicitly typed array:

```
var vals = new[] { 1, 2, 3, 4, 5 };
var names = new[] { "Ram", "Shyam", "Hari" };
```

- Following example creates a two-dimensional array of double

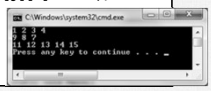
```
var vals = new[,] { { 1.1, 2.2 }, { 3.3, 4.4 }, { 5.5, 6.6 } };
```

Presented by

Ranjan Bhatnagar

Implicitly Typed Jagged Arrays

```
using System;
namespace CSProgExample
{
    class Jagged
    {
        static void Main()
        {
            var jagged = new[] {
                                new[] { 1, 2, 3, 4 },
                                new[] { 9, 8, 7 },
                                new[] { 11, 12, 13, 14, 15 }
                            };
            for (int j = 0; j < jagged.Length; j++)
            {
                for (int i = 0; i < jagged[j].Length; i++)
                    Console.Write(jagged[j][i] + " ");
                Console.WriteLine();
            }
        }
    }
}
```



Strings

- A string is an object of type System.String whose value is text.
- Internally, the text is stored as a sequential read-only collection of System.Char objects.
- There is no null-terminating character at the end of a C# string.
- String is a Unicode characters collection.
- Even though string is a reference type, the equality operators (== and !=) are defined to compare the values of string objects, not references.

Strings

```
using System;
namespace sample
{
    class Class1
    {
        static void Main(string[] args)
        {
            string s1 = "Good Morning";
            string s2;
            s2 = s1;
            s1 = "Wake Up";
            Console.WriteLine(s1);
            Console.WriteLine(s2);
            string s = "hello";
            string s1 = "h";
            s1 += "ello";
            Console.WriteLine(s == s1);  // True
            string str = "hello";
            char x = str[1]; // x = 'e';
        }
    }
}
```

System.String Class Example

```
using System;
namespace sample
{
    class Class1
    {
        static void Main(string[] args)
        {
            string s1 = "Delhi";
            string s2 = "Kolkata";
            Console.WriteLine("Char at 3rd position: " + s1[2]);

            string s3 = string.Concat(s1, s2);
            Console.WriteLine(s3);
            Console.WriteLine("Length of s3: " + s3.Length);

            s3 = s3.Replace('h', 'H');
            Console.WriteLine(s3);
            s3 = string.Copy(s2);
            Console.WriteLine(s3);
```

System.String Class Example

```
            int c = s2.CompareTo(s3);
            if (c < 0)
                Console.WriteLine("s2 is less than s3");
            if (c == 0)
                Console.WriteLine("s2 is equal to s3");
            if (c > 0)
                Console.WriteLine("s2 is greater than s3");

            if (s1 == s3)
                Console.WriteLine("s1 is equal to s3");
            else
                Console.WriteLine("s1 is not equal to s3");
```

System.String Class Example

```
            s3 = s1.ToUpper();
            Console.WriteLine(s3);
            s3 = s2.Insert(7, "Mumbai");
            Console.WriteLine(s3);
            s3 = s2.Remove(0, 1);
            Console.WriteLine(s3);
            int fin = s1.IndexOf('D');
            Console.WriteLine("First index of D in s1: " + fin);
            int lin = s1.LastIndexOf('l');
            Console.WriteLine("Last index of l in s1: " + lin);
            string sub = s1.Substring(fin, lin);
            Console.WriteLine("Substring: " + sub);
            int i = 10;
            float f = 9.8f;
            s3 = string.Format("Value of i:{0}\nValue of f:{1}", i, f);
            Console.WriteLine(s3);
        }
    }
}
```

Presented by

Ranjan Bhatnagar

Immutability

- String objects are immutable, that is, they cannot be changed after they have been created.
- To understand this, let us look at the code below

```
string s = "hello";
s.Substring(1);
System.Console.WriteLine(s); //prints hello
s = s.Substring(1);
System.Console.WriteLine(s);// prints ello
```

- The Substring method did not change the original object. It returned a new string object.
- All the String methods behave in the same way.
- string s = " Welcome " + " back "; This statements ends up creating 3 strings → Welcome, back and Welcome, back

StringBuilder

- If a string has to undergo a lot of manipulation, then the application will end up creating lots of strings. How can this be avoided?
- System.Text namespace defines a StringBuilder class that creates mutable string.
- The methods of this class are very similar to String class.

```
using System.Text;

StringBuilder b = new StringBuilder("Hello");
b.Remove(1, 2);
System.Console.WriteLine(b); // prints Hlo
```

StringBuilder Example

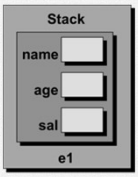
```
using System;
using System.Text;
public sealed class App
{
    static void Main()
    {
        // Create a StringBuilder that expects to hold 50 characters.
        // Initialize the StringBuilder with "ABC".
        StringBuilder sb = new StringBuilder("ABC", 50);
        // Append three characters (D, E, and F) at the end
        sb.Append(new char[] { 'D', 'E', 'F' });
        // Append a format string to the end of the StringBuilder.
        sb.AppendFormat("GHI{0}{1}", 'J', 'k');
        // Display number of characters within and its string.
        Console.WriteLine("{0} chars: {1}", sb.Length, sb.ToString());
        // Insert a string at the beginning of the StringBuilder.
        sb.Insert(0, "Alphabet: ");
        // Replace all lowercase k's with uppercase K's.
        sb.Replace('k', 'K');
        // Display the number of characters within and its string.
        Console.WriteLine("{0} chars: {1}", sb.Length, sb.ToString());
    }
}
```

Struct

- Structs or Structures are data structures are composed of element of different types.
- Structs are defined by using the struct keyword.
- Whenever we needed to group dissimilar data along with some functionality we used classes.
- Structures are similar to classes because they too contain dissimilar data along with methods.
- Note that even though we have used new, space for struct gets allocated on the stack.

Structure Example

```
using System;
struct emp
{
    string name;
    int age;
    float sal;
    public emp(string n,
               int a, float s)
    {
        name = n;
        age = a;
        sal = s;
    }
    public void show()
    {
        Console.WriteLine
          ("Name: " + name);
        Console.WriteLine
          ("Age: " + age);
       Console.WriteLine
          ("Salary: " + sal);
    }
}
```




```
class Program
{
    static void Main
                (string[] args)
    {
        emp e1;
        e1 = new
       emp("Amit", 35, 25000);
        e1.show();
    }
}
```

Classes v/s Structures

Object on heap	**Object on stack**
Reference on stack	**Variable directly refers structure**
Always passed by reference	**Always passed by value**
Can be inherited	**Cannot be inherited**
Can have protected modifier	**Cannot have protected modifier**
Provide default 0-arg constructor when no other constructor is provided	**Always provide 0-arg constructor**
We can provide our own 0-arg constructor	**We cannot provide our own 0-arg constructor**
Can have virtual methods	**Cannot have virtual methods**

Presented by

Ranjan Bhatnagar

Enumeration (enum)

- An enumeration (enum) is a special form of value type, which inherits from the System.Enum class.
- An Enum consists of underline data types. The underlying types must be one of the built-in signed or unsigned data types.
- We can access the enum members using the enum names followed by the dot.
- Each enum member gets initialized with a value, default value starts from zero.

enum Examples

```
enum person
{
   married,
   unmarried,
   divorced,
   spinster,
};
enum emp
{
   skilled,
   semiskilled,
   highlyskiiled,
   unskilled,
};
void Main(string[ ] args)
{
   person  p ;
   p = person.married ;
   emp  e ;
   e = emp.skilled ;
   if ( e == emp.skilled )
    C.W ( "Qualified for  promotion" );
   C.W ( ( int ) emp.skilled ) ;
   string[ ]  str = Enum.GetNames
                    ( e.GetType( ) ) ;
   foreach ( string s in str )
      Console.WriteLine ( s ) ;
}
```

Keyword

0

Returns RTTI Type reference

skilled
semiskilled
highlyskilled
unskilled

enum Tips

- Elements of an enum can't be changed

```
enum color
{
   red, green,
   blue
}
```

```
color.red = 10
```

Error

- Underlying datatypes - byte, long, short. Default - integer

```
enum color : int
{
   red, green = 5,
   blue
}
```

```
enum color : byte
{
   red, green = 500,
   blue
}
```

Error

Methods

- A method is a code block containing a series of statements.
- In C#, every executed instruction is done so in the context of a method.
- Methods are declared within a class or struct by specifying the access level, the return value, the name of the method, and any method parameters.
- Method parameters are surrounded by parentheses, and separated by commas.
- Empty parentheses indicate that the method requires no parameters.

Example

```
using System;
class Program
{
    static void Main()
    {
        Message();
    }
    static void Message()
    {
        Console.WriteLine("Hello");
    }
}
```

- In the example Message is a method that simply prints a message.
- It neither receives any argument nor it returns anything.

- Passing arguments to a method is simply a matter of providing them in the parentheses when calling a method.

```
using System;
class Program
{
    static void Main()
    {
        AddNum(10, 20);
    }
    static void AddNum(int x, int y)
    {
        int s;
        s = x + y;
        Console.WriteLine("Sum :" + s);
    }
}
```

Parameters

```
using System;
class Program
{
    static int SumVals(params int[] vals)
    {
        int sum = 0;
        foreach (int val in vals)
        {
            sum += val;
        }
        return sum;
    }
    static void Main(string[] args)
    {
        int sum = SumVals(1, 5, 2, 9, 8);
        Console.WriteLine("Summed Values = {0}", sum);
        sum = SumVals(1, 5, 2, 9, 8, 4, 7, 2);
        Console.WriteLine("Summed Values = {0}", sum);
    }
}
```

Presented by

Ranjan Bhatnagar

Parameter Types

- If arguments are passed by **value** changes made by the method are not visible to the calling code
- If arguments are passed by **reference** changes made by the method are visible to the calling code
- If arguments are passed by **out** during a call the reference of argument gets copied and changes made in calling function would affect actual argument
- If arguments are passed by **params** during a call arguments passed to function are collected in the array
- By default, arguments are passed by **value**

Parameter Types

Value Type

```
void fun ( int x )
{
}

int i = 9 ;
fun ( i ) ;
```

Reference Type

```
void fun ( ref int x )
{
}

int i = 9 ;
fun ( ref i ) ;
```

Output Type

```
void fun ( out int x )
{
}

int i = 9 ;
fun ( out i ) ;
```

Params Type

```
void fun ( params int[] x )
{
}

int i = 9, j = 4, k = 7 ;
fun ( i, j, k ) ;
```

Thanks

Presented by

Ranjan Bhatnagar