Unboxing

- Unboxing is a mechanism in which reference type is converted into value.
- It is explicit conversion process.

Example

```
int      i, j;
object  obj;
string   s;
i  = 32;
obj = i;      // boxed copy!
i   = 19;
j   = (int) obj; // unboxed!
s = j.ToString();  // boxed!
s = 99.ToString(); // boxed!
```

Difference Between Int32.Parse(), Convert.ToInt32(), and Int32.TryParse(),(int)

Int32.Parse (string s) method converts the string representation of a number to its 32-bit signed integer equivalent.

- When s is a null reference, it will throw ArgumentNullException.
- If s is other than integer value, it will throw FormatException.
- When s represents a number less than MinValue or greater than MaxValue, it will throw OverflowException.

- (int) will only convert types that can be represented as an integer (ie double, long, float, etc) although some data loss may occur.
- string s1 = "1234";
- string s2 = "1234.65";
- string s3 = null;
- string s4 =123456789123456789123456789123456789123456789";
- int result;
- bool success;
- result = Int32.Parse(s1); *//-- 1234*
- result = Int32.Parse(s2); *//-- FormatException*
- result = Int32.Parse(s3); *//-- ArgumentNullException*
- result = Int32.Parse(s4); *//-- OverflowException*

Convert.ToInt32(string):This method converts the specified string representation of 32-bit signed integer equivalent. This calls in turn **Int32.Parse** () method.

- When s is a null reference, it will return 0 rather than throw ArgumentNullException.
- If s is other than integer value, it will throwFormatException.
- When s represents a number less than MinValue or greater than MaxValue, it will throw OverflowException. For example:

- result = Convert.ToInt32(s1); *//-- 1234*
- result = Convert.ToInt32(s2); *//-- FormatException*
- result = Convert.ToInt32(s3); *//-- 0*
- result = Convert.ToInt32(s4); *//-- OverflowException*

Int32.TryParse(string, out int): This method converts the specified string representation of 32-bit signed integer equivalent to out variable, and returns true if it is parsed successfully, false otherwise.

  - When s is a null reference, it will return 0 rather than throwArgumentNullException.
  - If s is other than an integer value, the out variable will have 0 rather thanFormatException.
  - When s represents a number less than MinValue or greater than MaxValue, the outvariable will have 0 rather than OverflowException.
  - success = Int32.TryParse(s1, out result); *//-- success => true; result => 1234*
- success = Int32.TryParse(s2, out result); *//-- success => false; result => 0*
- success = Int32.TryParse(s3, out result); *//-- success => false; result => 0*
- success = Int32.TryParse(s4, out result); *//-- success => false; result => 0*
- Convert.ToInt32 is better than **Int32.Parse** since it returns 0 rather than an exception. But again, according to the requirement, this can be used. TryParse will be the best since it always handles exceptions by itself.

Unit-III
C# Using Libraries

3.1 Namespace- System

System Namespace in fundamental namespace for c# application. It contain all the fundamental classes and base classes which are required in simple C# application. These classes and sub classes defines reference data type, method and interfaces. Some classes provide some other feature like data type conversion, mathematical function.

Some functionality provided by System namespace

- Commonly-used value
- Mathematics
- Remote and local program invocation
- Application environment management
- Reference data types
- Events and event handlers
- Interfaces Attributes Processing exceptions
- Data type conversion
- Method parameter manipulation

Some Classes provide by System namespace

- AccessViolationException
- Array
- ArgumentNullException
- AttributeUsageAttribute
- Buffer
- Console
- Convert
- Delegate
- Exception
- InvalidCastException

Some interfaces provided by System namespace

- Public interface ICloneable
- Public interface IComparable
- Public interface IComparable<T>
- Public interface IConvertible
- Public interface ICustomFormatter
- Public interface IDisposable
- Public interface IEquatable<T>
- Public interface

IFormatProvider MATH EXAMPLE

```
using System; class
Pythagorean { static
void Main() {
double s1;
double s2;
double hypot;
string str;
Console.WriteLine("Enter length of first side:
"); str = Console.ReadLine();
s1 = Double.Parse(str);
Console.WriteLine("Enter length of second side: ");
str = Console.ReadLine();
s2 = Double.Parse(str);
hypot = Math.Sqrt(s1*s1 + s2*s2);
Console.WriteLine("Hypotenuse is " + hypot);
}
}
```

Sorting and Searching,Reverse,Copy Arrays

Using Sort( ), you can sort an entire array, a range within an array, or a pair of arrays that contain corresponding key/value pairs. Once an array has been sorted, you can efficiently search it using BinarySearch( ).

```
using System; class
SortDemo { static
void Main() {
int[] nums = { 5, 4, 6, 3, 14, 9, 8, 17, 1, 24, -1, 0
}; Console.Write("Original order: ");
foreach(int i in nums)
Console.Write(i + " ");  Console.WriteLine();
Array.Sort(nums);
Console.Write("Sorted order: ");
foreach(int i in nums)
Console.Write(i + " ");   Console.WriteLine();
int idx = Array.BinarySearch(nums, 14);
Console.WriteLine("Index of 14 is " + idx); } }
```

The IComparable and IComparable<T> Interfaces

- Many classes will need to implement either the IComparable or IComparable<T> interface because they enable one object to be compared to another (for the purpose of ordering) by various methods defined by the .NET Framework
- **IComparable is especially easy to implement because it consists of just this one method:**
- int CompareTo(object *obj)*
- This method compares the invoking object against the value in *obj. It returns greater than zero* if the invoking object is greater than *obj, zero if the two objects are equal, and less than* zero if the invoking object is less than *obj.*

StringBuilder in C#

Once created a string cannot be changed. A StringBuilder can be changed as many times as necessary. It yields astonishing performance improvements. It eliminates millions of string copies. Many C# programs append or replace strings in loops. There the StringBuilder type becomes a necessary optimization. It uses the new keyword for StringBuilder. Use the new keyword to make your StringBuilder. This is different from regular strings. StringBuilder has many overloaded constructors. continuing on it calls the instance Append method. This method adds the contents of its arguments to the buffer in the StringBuilder. Every argument to StringBuilder will automatically have its ToString method called. It calls AppendLine, which does the exact same thing as Append, except with a new line on the end. Next, Append and Append Line call themselves. This shows terse syntax with StringBuilder. Finally ToString returns the buffer. You will almost always want ToString. It will return the contents as a string.

Example

```
using System;
using System.Text;
class Program
{
   static void Main()
   {
      StringBuilder builder = new StringBuilder();
      // Append to StringBuilder.
      for (int i = 0; i < 10; i++)
      {
         builder.Append(i).Append(" ");
      }
      Console.WriteLine(builder);
   }
}
```

3.2 Input-Output

C# programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ. Thus, the I/O classes and methods can be applied to many types of devices. For example, the same methods that you use to write to the console can also be used to write to a disk file.

Byte Streams and Character Streams

At the lowest level, all C# I/O operates on bytes. This makes sense because many devices are byte oriented when it comes to I/O operations. Frequently, though, we humans prefer to communicate using characters. Recall that in C#, **char** is a 16-bit type, and **byte** is an 8-bit type. If you are using the ASCII character set, then it is easy to convert between **char** and **byte**; just ignore the high-order byte of the **char** value. But this won't work for the rest of the Unicode characters, which need both bytes (and possibly more). Thus, byte streams are not perfectly suited to handling character-based I/O. To solve this problem, the .NET Framework defines several classes that convert a byte stream into a character stream, handling the translation of **byte**-to-**char** and **char**-to-**byte** for you automatically.

The Predefined Streams

Three predefined streams, which are exposed by the properties called **Console.In**, **Console.Out**, and **Console.Error**, are available to all programs that use the **System** namespace. **Console.Out** refers to the standard output stream. By default, this is the console. When you call **Console.WriteLine( )**, for example, it automatically sends information to **Console.Out**. **Console.In** refers to standard input, which is, by default, the keyboard. **Console.Error** refers to the standard error stream, which is also the console by default. However, these streams can be redirected to any compatible I/O device. The standard streams are character streams. Thus, these streams read and write characters.

System.IO Namespace

- **BinaryReader Class:** Reads primitive data types as binary values in a specific encoding.
- **BinaryWriter Class :** Writes primitive types in binary to a stream and supports writing strings in a specific encoding.
- **BufferedStream Class :** Adds a buffering layer to read and write operations on another stream. This class cannot be inherited.

- **Directory Class:** Exposes static methods for creating, moving, and enumerating through directories and subdirectories. This class cannot be inherited.
- **DirectoryInfo Class:** Exposes instance methods for creating, moving, and enumerating through directories and subdirectories. This class cannot be inherited.
- **File Class:**Provides static methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects.
- **FileInfo :**Provides properties and instance methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects. This class cannot be inherited.
- **FileStream**: Exposes a Stream around a file, supporting both synchronous and asynchronous read and write operations.
- **IOException** Class
- **Path** Class
- **Stream** Class: Provides a generic view of a sequence of bytes.
- **StreamReader:** Implements a TextReader that reads characters from a byte stream in a particular encoding.
- **StreamWriter :** Implements a TextWriter for writing characters to a stream in a particular encoding.
- **StringReader :**Implements a TextReader that reads from a string.
- **StringWriter:** Implements a TextWriter for writing information to a string. The information is stored in an underlying StringBuilder.
- **TextReader:** Represents a reader that can read a sequential series of characters.
- **TextWriter:**Represents a writer that can write a sequential series of characters. This class is abstract.

Creating and writing text on a File

```
namespace IOTest{
 class Program  {
       static void Main(string[] args)      {
           StreamWriter sw;
          sw= File.CreateText("d:/workspace/Hello.txt");
         sw.WriteLine("Hello Mca Students This is your basic
         IO"); //sw.Flush();
          //sw.Close();
          Console.WriteLine("Please show the file in d drive ");
       } } }

class TextFileWriter  {
       static void Main(string[] args)      {
           TextWriter tw = new StreamWriter("date.txt");
           tw.WriteLine(DateTime.Now);
          tw.Close();
       }  }
class TextFileReader  {
       static void Main(string[] args)      {
           Textreader tr = new StreamReader("date.txt");
           Console.WriteLine(tr.ReadLine()); tr.Close();


       }
   }
```