CSharp string **Split** function returns an array of String containing the substrings delimited by the given System.Char array.

***string[] string.split(string[] separator)***

**Parameters:**

separator - the given delimiter

**Returns:**

An array of Strings delimited by one or more characters in separator

using System;

using System.Windows.Forms;

namespace WindowsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

string str = null;

string[] strArr = null;

int count = 0;

str = "CSharp split test";

char[] splitchar = { ' ' };

strArr = str.Split(splitchar);

for (count = 0; count < = strArr.Length - 1; count++)

{

MessageBox.Show(strArr[count]);

}

}

}

}

Collections are data structures that holds data in different ways for flexible operations . C# Collection classes are defined as part of the System.Collections or System.Collections.Generic namespace.

Most collection classes implement the same interfaces, and these interfaces may be inherited to create new collection classes that fit more specialized data storage needs. The following tutorial shows how to implement collection classes in C# programming.

ArrayList is one of the most flexible data structure from [CSharp](http://csharp.net-informations.com) Collections. ArrayList contains a simple list of values. ArrayList implements the IList interface using an array and very easily we can add , insert , delete , view etc. It is very flexible because we can add without any size information , that is it will grow dynamically and also shrink .

**Add : Add an Item in an ArrayList**

**Insert : Insert an Item in a specified position in an ArrayList**

**Remove : Remove an Item from ArrayList**

**RemoveAt: remove an item from a specified position**

**Sort : Sort Items in an ArrayList**

**How to add an Item in an ArrayList ?**

**Syntax : ArrayList.add(object)**

**object : The Item to be add the ArrayList**

**ArrayList arr;**

**arr.Add("Item1");**

**How to Insert an Item in an ArrayList ?**

**Syntax : ArrayList.insert(index,object)**

**index : The position of the item in an ArrayList**

**object : The Item to be add the ArrayList**

**ArrayList arr;**

**arr.Insert(3, "Item3");**

**How to remove an item from arrayList ?**

**Syntax : ArrayList.Remove(object)**

**object : The Item to be add the ArrayList**

**arr.Remove("item2")**

**How to remove an item in a specified position from an ArrayList ?**

**Syntax : ArrayList.RemoveAt(index)**

**index : the position of an item to remove from an ArrayList**

**ItemList.RemoveAt(2)**

**How to sort ArrayList ?**

**Syntax : ArrayList.Sort()**

using System;

using System.Collections;

using System.Windows.Forms;

namespace WindowsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

int i = 0;

ArrayList ItemList = new ArrayList();

ItemList.Add("Item4");

ItemList.Add("Item5");

ItemList.Add("Item2");

ItemList.Add("Item1");

ItemList.Add("Item3");

MessageBox.Show ("Shows Added Items");

for (i = 0; i < = ItemList.Count - 1; i++)

{

MessageBox.Show(ItemList[i].ToString());

}

//insert an item

ItemList.Insert(3, "Item6");

//sort itemms in an arraylist

ItemList.Sort();

//remove an item

ItemList.Remove("Item1");

//remove item from a specified index

ItemList.RemoveAt(3);

MessageBox.Show("Shows final Items the ArrayList");

for (i = 0; i < = ItemList.Count - 1; i++)

{

MessageBox.Show(ItemList[i].ToString());

}

}

}

}

Hashtable in C# represents a collection of key/value pairs which maps keys to value. Any non-null object can be used as a key but a value can. We can retrieve items from hashTable to provide the key . Both keys and values are Objects.

The commonly used functions in Hashtable are :

**Add : To add a pair of value in HashTable**

**ContainsKey : Check if a specified key exist or not**

**ContainsValue : Check the specified Value exist in HashTable**

**Remove : Remove the specified Key and corresponding Value**

**Add : To add a pair of value in HashTable**

**Syntax : HashTable.Add(Key,Value)**

**Key : The Key value**

**Value : The value of corresponding key**

**Hashtable ht;**

**ht.Add("1", "Sunday");**

**ContainsKey : Check if a specified key exist or not**

**Synatx : bool HashTable.ContainsKey(key)**

**Key : The Key value for search in HahTable**

**Returns : return true if item exist else false**

**ht.Contains("1");**

**ContainsValue : Check the specified Value exist in HashTable**

**Synatx : bool HashTable.ContainsValue(Value)**

**Value : Search the specified Value in HashTable**

**Returns : return true if item exist else false**

**ht.ContainsValue("Sunday")**

**Remove : Remove the specified Key and corresponding Value**

**Syntax : HashTable.Remove(Key)**

**Key : The key of the element to remove**

**ht.Remove("1");**

The following source code shows some important operations in a HashTable

using System;

using System.Collections;

using System.Windows.Forms;

namespace WindowsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

Hashtable weeks = new Hashtable();

weeks.Add("1", "SunDay");

weeks.Add("2", "MonDay");

weeks.Add("3", "TueDay");

weeks.Add("4", "WedDay");

weeks.Add("5", "ThuDay");

weeks.Add("6", "FriDay");

weeks.Add("7", "SatDay");

//Display a single Item

MessageBox.Show(weeks["5"].ToString ());

//Search an Item

if (weeks.ContainsValue("TueDay"))

{

MessageBox.Show("Find");

}

else

{

MessageBox.Show("Not find");

}

//remove an Item

weeks.Remove("3");

//Display all key value pairs

foreach (DictionaryEntry day in weeks )

{

MessageBox.Show (day.Key + " - " + day.Value );

}

}

}

}

The Stack class represents a **last-in-first-out (LIFO)** Stack of Objects. Stack follows the push-pop operations. That is we can Push (insert) Items into Stack and Pop (retrieve) it back . Stack is implemented as a circular buffer. It follows the Last In First Out (LIFO) system. That is we can push the items into a stack and get it in reverse order. Stack returns the last item first. As elements are added to a Stack, the capacity is automatically increased as required through reallocation.

Commonly used methods :

**Push : Add (Push) an item in the Stack data structure**

**Pop : Pop return the last Item from the Stack**

**Contains: Check the object contains in the Stack**

**Push : Add (Push) an item in the Stack data structure**

**Syntax : Stack.Push(Object)**

**Object : The item to be inserted.**

**Stack days = new Stack();**

**days.Push("Sunday");**

**Pop : Pop return the item last Item from the Stack**

**Syntax : Object Stack.Pop()**

**Object : Return the last object in the Stack**

**days.Pop();**

**Contains : Check the object contains in the Stack**

**Syntax : Stack.Contains(Object)**

**Object : The specified Object to be search**

**days.Contains("Tuesday");**

The following CSharp Source code shows some of important functions in Stack Class:

using System;

using System.Collections;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

namespace WindowsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

Stack days = new Stack();

days.Push("SunDay");

days.Push("MonDay");

days.Push("TueDay");

days.Push("WedDay");

days.Push("ThuDay");

days.Push("FriDay");

days.Push("SaturDay");

if (days.Count ==7)

{

MessageBox.Show(days.Pop().ToString ());

}

else

{

MessageBox.Show("SaturDay does not exist");

}

}

}

}

The Queue works like **FIFO** system , a **first-in, first-out** collection of Objects. Objects stored in a Queue are inserted at one end and removed from the other. The Queue provide additional insertion, extraction, and inspection operations. We can **Enqueue** (add) items in Queue and we can **Dequeue** (remove from Queue ) or we can Peek (that is we will get the reference of first item ) item from Queue. Queue accepts null reference as a valid value and allows duplicate elements.

Some important functions in the Queue Class are follows :

**Enqueue : Add an Item in Queue**

**Dequeue : Remove the oldest item from Queue**

**Peek : Get the reference of the oldest item**

**Enqueue : Add an Item in Queue**

**Syntax : Queue.Enqueue(Object)**

**Object : The item to add in Queue**

**days.Enqueue("Sunday");**

**Dequeue : Remove the oldest item from Queue (we don't get the item later)**

**Syntax : Object Queue.Dequeue()**

**Returns : Remove the oldest item and return.**

**days.Dequeue();**

**Peek : Get the reference of the oldest item (it is not removed permanently)**

**Syntax : Object Queue.Peek()**

**returns : Get the reference of the oldest item in the Queue**

**days.peek();**

The following CSharp Source code shows some of commonly used functions :

using System;

using System.Collections;

using System.Windows.Forms;

namespace WindowsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

Queue days = new Queue();

days.Enqueue("Sunday");

days.Enqueue("Monday");

days.Enqueue("Tuesday");

days.Enqueue("Wednsday");

days.Enqueue("Thursday");

days.Enqueue("Friday");

days.Enqueue("Saturday");

MessageBox.Show (days.Dequeue().ToString ());

if (days.Contains("Monday"))

{

MessageBox.Show("The queue contains Monday");

}

else

{

MessageBox.Show("Does not match any entries");

}

}

}

}

**NameValueCollection** is used to store a collection of associated String ***keys*** and String ***values*** that can be accessed either with the key or with the index. It is very similar to C# HashTable, HashTable also stores data in Key , value format .

NameValueCollection can hold multiple string values under a single key. As elements are added to a NameValueCollection, the capacity is automatically increased as required through reallocation. The one important thing is that you have to import ***System.Collections.Specialized*** Class in your program for using NameValueCollection.

**Adding new pairs**

**NameValueCollection.Add(name,value)**

**NameValueCollection pair = new NameValueCollection();**

**pair.Add("High", "80");**

**Get the value of corresponding Key**

**string[] NameValueCollection.GetValues(index);**

**NameValueCollection pair = new NameValueCollection();**

**pair.Add("High", "80");**

**string[] vals = pair.GetValues(1);**

using System;

using System.Collections;

using System.Windows.Forms;

using System.Collections.Specialized;

namespace WindowsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

NameValueCollection markStatus = new NameValueCollection();

string[] values = null;

markStatus.Add("Very High", "80");

markStatus.Add("High", "60");

markStatus.Add("medium", "50");

markStatus.Add("Pass", "40");

foreach (string key in markStatus.Keys)

{

values = markStatus.GetValues(key);

foreach (string value in values)

{

MessageBox.Show (key + " - " + value);

}

}

}

}

}

|  |
| --- |
| **params keyword** **by: [kirk](http://www.csharpfriends.com/Forums/User/UserProfile.aspx?UserName=kirk)** |
| |  | | --- | |  |   params keyword   The params keyword allows methods to have a variable length parameter list.   For example, the following class defines a method called "MultiPrint", which can have any number of string's passed to it.   |  | | --- | | using [System](http://www.csharpfriends.com/Articles/getArticle.aspx?articleID=49);  public class MyApplication  {  public static void MultiPrint(params string[] list)  {  for ( int i = 0 ; i < list.Length ; i++ )  Console.WriteLine(list[i]);  }  public static void Main()  {  MultiPrint("First", "Second", "Third");  MultiPrint("Fourth");  MultiPrint("Fifth", "Sixth");  }  } | |

You have strings you are using and need to put them in a **string Dictionary** for fast lookups. This matches keys and values and is an associative array. Here we use string Dictionaries, which have excellent performance and are easy to create, using the C# programming language.

**Using string Dictionary**

Here we see an example of how you can create a new Dictionary with **string** keys and string values. String Dictionaries are the most common in many cases and are very useful. The example shows how the string Dictionary is populated and tested.

**~~~ Program that uses Dictionary of strings (C#) ~~~**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// A. Create a new Dictionary with two keys and two values.

Dictionary<string, string> example = new **Dictionary**<string, string>();

example.Add("string1", "C#");

example.Add("string2", "Perl");

// B. Lookup a string in the Dictionary.

string value;

if (example.TryGetValue("string2", out value))

{

Console.WriteLine("Found {0}", value);

}

// C. See if it contains this key.

Console.WriteLine(example.ContainsKey("string3"));

// D. Add this key if it isn't there.

if (!example.ContainsKey("string3"))

{

example.Add("string3", "VB.NET");

}

// E. Enumerate the keys

foreach (var pair in example)

{

Console.WriteLine("Key = {0}, Value = {1}", pair.Key, pair.Value);

}

}

}

**~~~ Output of the program ~~~**

Key = string1, Value = C#

Key = string2, Value = Perl

Key = string3, Value = VB.NET

**Steps in control flow.** Part A of the example above simply instantiates a new empty string Dictionary, and then adds two keys and two values. Part B shows how you can use the TryGetValue method, which often enhances performance by reducing key lookups. The code here will reach the Console.WriteLine call.

**Further steps.** Part C shows the result of the ContainsKey call. The key is not found, and thus the result printed is "False". No exception is raised. Part D shows the tester-doer pattern. The if conditional tests that the key doesn't exist. Then it adds it. If you add a key that is already there, you get an exception. Finally, it enumerates all the KeyValuePairs in the Dictionary's internal layout.

**More usages**

The majority of programs the author has seen use strings as the keys in Dictionaries. This is because it is very challenging to map strings to indexes in a performant way without a hash key.

**Avoiding string Dictionary.** When you need to add key/value pairs where the key is an int or other numeric type, it is usually better to use an int Dictionary. This is because there is less chance of errors. For example, the keys "-1" and " -1" are different as strings, but mean the same thing as numbers.

**Utilizing the StringComparer class.** In the C# programming language, you can provide a StringComparer class to the parameter list of the Dictionary object constructor. If you are using a Dictionary that does not require globalization features, you can apply the StringComparer.Ordinal class instance to the parameter list to achieve a substantial performance boost on all string key lookups.

[**(See Dictionary StringComparer Tip.)**](http://dotnetperls.com/dictionary-stringcomparer)

**Summary**

Here we saw how you can use the **string Dictionary** constructed type in the C# language. String Dictionaries are an exceedingly useful and powerful mechanism in C# and .NET. They are ideal for storing key/value pairs where the key is a name, word, or other complex data that is best represented textually.

 Remarks

Use [CreateSection](http://msdn.microsoft.com/en-us/library/system.collections.specialized.bitvector32.createsection(VS.80).aspx) to define a new section. A **BitVector32.Section** is a window into the [BitVector32](http://msdn.microsoft.com/en-us/library/system.collections.specialized.bitvector32(VS.80).aspx) and is composed of the smallest number of consecutive bits that can contain the maximum value specified in **CreateSection**. For example, a section with a maximum value of 1 is composed of only one bit, whereas a section with a maximum value of 5 is composed of three bits. You can create a **BitVector32.Section** with a maximum value of 1 to serve as a Boolean, thereby allowing you to store integers and Booleans in the same **BitVector32**.

using System;

using System.Collections.Specialized;

public class SamplesBitVector32 {

public static void Main() {

// Creates and initializes a BitVector32.

BitVector32 myBV = new BitVector32( 0 );

// Creates four sections in the BitVector32 with maximum values 6, 3, 1, and 15.

// mySect3, which uses exactly one bit, can also be used as a bit flag.

BitVector32.Section mySect1 = BitVector32.CreateSection( 6 );

BitVector32.Section mySect2 = BitVector32.CreateSection( 3, mySect1 );

BitVector32.Section mySect3 = BitVector32.CreateSection( 1, mySect2 );

BitVector32.Section mySect4 = BitVector32.CreateSection( 15, mySect3 );

// Displays the values of the sections.

Console.WriteLine( "Initial values:" );

Console.WriteLine( "\tmySect1: {0}", myBV[mySect1] );

Console.WriteLine( "\tmySect2: {0}", myBV[mySect2] );

Console.WriteLine( "\tmySect3: {0}", myBV[mySect3] );

Console.WriteLine( "\tmySect4: {0}", myBV[mySect4] );

// Sets each section to a new value and displays the value of the BitVector32 at each step.

Console.WriteLine( "Changing the values of each section:" );

Console.WriteLine( "\tInitial: \t{0}", myBV.ToString() );

myBV[mySect1] = 5;

Console.WriteLine( "\tmySect1 = 5:\t{0}", myBV.ToString() );

myBV[mySect2] = 3;

Console.WriteLine( "\tmySect2 = 3:\t{0}", myBV.ToString() );

myBV[mySect3] = 1;

Console.WriteLine( "\tmySect3 = 1:\t{0}", myBV.ToString() );

myBV[mySect4] = 9;

Console.WriteLine( "\tmySect4 = 9:\t{0}", myBV.ToString() );

// Displays the values of the sections.

Console.WriteLine( "New values:" );

Console.WriteLine( "\tmySect1: {0}", myBV[mySect1] );

Console.WriteLine( "\tmySect2: {0}", myBV[mySect2] );

Console.WriteLine( "\tmySect3: {0}", myBV[mySect3] );

Console.WriteLine( "\tmySect4: {0}", myBV[mySect4] );

}

}

/\*

This code produces the following output.

Initial values:

mySect1: 0

mySect2: 0

mySect3: 0

mySect4: 0

Changing the values of each section:

Initial: BitVector32{00000000000000000000000000000000}

mySect1 = 5: BitVector32{00000000000000000000000000000101}

mySect2 = 3: BitVector32{00000000000000000000000000011101}

mySect3 = 1: BitVector32{00000000000000000000000000111101}

mySect4 = 9: BitVector32{00000000000000000000001001111101}

New values:

mySect1: 5

mySect2: 3

mySect3: 1

mySect4: 9

\*/

String Builder *- So you want to learn about the StringBuilder eh?*   
  
In Classic [ASP](http://www.csharpfriends.com/Articles/getArticle.aspx?articleID=13) we were accustomed to forming [sql](http://www.csharpfriends.com/Articles/getArticle.aspx?articleID=13) strings or any other string by concatenating values using '&' that would sometimes span many lines. A fast and efficient way of performing this same task in C# is so much cleaner and a pleasure to work with.   
  
You will find this class in [System](http://www.csharpfriends.com/Articles/getArticle.aspx?articleID=13).Text (namespace). StringBuilder is ideal for appending, removing, replacing, or inserting characters. This size of your string can be modified dynamically, no need to set a size during declaration.   
  
Common usage of StringBuilder would be concatenating string values, replacing values in a string e.g Replacing all occurances of ' with ''

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | <%@ Page language="c#"%> |   <script language="C#" runat="[server](http://www.csharpfriends.com/Articles/getArticle.aspx?articleID=13)">  private void Page\_Load(object sender, System.EventArgs e)  {  //Strinbuilder class in found in System.Text namespace  System.Text.StringBuilder sb = new System.Text.StringBuilder();  sb.Append("Hel");  sb.Append("la,");  sb.Append(" War");  sb.Append("ld!");    Response.Write("<br><h1>Capacity of sb is: " + sb.Capacity);  Response.Write("<br><br>Length is: " + sb.Length + "<br>");  for(int i = 0; i < sb.Length; i++)  Response.Write(sb[i]);    Response.Write("<br>");    sb.Replace("a","o");    for(int i = 0; i < sb.Length; i++)  Response.Write(sb[i]);  Response.Write("</h1>");  }  </script>  <html>  <head>  <title>StringBuilder</title>  </head>  <body>  </body>  </html> |

**Stringbuilder :** [**http://msdn.microsoft.com/en-us/library/2839d5h5(VS.71).aspx**](http://msdn.microsoft.com/en-us/library/2839d5h5(VS.71).aspx)

**Difference between StringBuilder and StringBuffer**

[**http://kaioa.com/node/59**](http://kaioa.com/node/59)

What is Boxing and UnBoxing in C#?

**User Rating** (129 votes)

0 reviews available

**A:** Boxing and unboxing is a  essential concept in C#’s type system. With Boxing and unboxing one can link between value-types and reference-types by allowing any value of a value-type to be converted to and   
from type object. Boxing and unboxing enables a unified view of the type system wherein a value of any   
type can ultimately be treated as an object. **Converting a value type to reference type is called Boxing**. Unboxing is an explicit operation.

C# provides a unified type system. All types including value types derive from the type object. It is   
possible to call object methods on any value, even values of  primitive  types such as int.

The example

using System;  
class Test  
{  
  static void Main()  
  {  
    Console.WriteLine(3.ToString());  
  }  
}  
calls the object-defined ToString method on an integer literal.

The example

class Test  
{  
  static void Main()  
  {  
   int i = 1;  
   object o = i;    // boxing  
   int j = (int) o; // unboxing  
  }  
}

An int value can be converted to object and back again to int.

This example shows both boxing and unboxing. When a variable of a value type needs to be converted to a reference type, an object box is allocated to hold the value, and the value is copied into the box.

Unboxing is just the opposite. When an object box is cast back to its original value type, the value is   
copied out of the box and into the appropriate storage location.

### Boxing conversions

A boxing conversion permits any value-type to be implicitly converted to the type object or to any   
interface-type implemented by the value-type. Boxing a value of a value-type consists of allocating an object instance and copying the value-type   
value into that instance.

For example any value-type H, the boxing class would be declared as follows:

class vBox  
{  
  H value;  
  H\_Box(H h)  
  {  
   value = h;  
  }  
}

Boxing of a value v of type H now consists of executing the expression new H\_Box(v), and returning the   
resulting instance as a value of type object.   
Thus, the statements  
int i = 12;  
object box = i;

conceptually correspond to  
int i = 12;  
object box = new int\_Box(i);

Boxing classes like H\_Box and int\_Box above don’t actually exist and the dynamic type of a boxed value   
isn’t actually a class type. Instead, a boxed value of type G has the dynamic type G, and a dynamic type check using the is operator can simply reference type G.

For example,

int i = 12;  
object box = i;  
if (box is int)  
{  
  Console.Write("Box contains an int");  
}

will output the string  Box contains an int on the console.

A boxing conversion implies making a copy of the value being boxed. This is different from a conversion   
of a reference-type to type object, in which the value continues to reference the same instance and   
simply is regarded as the less derived type object.

For example, given the declaration

struct Point  
{  
  public int x, y;  
  public Point(int x, int y)  
  {  
    this.x = x;  
    this.y = y;  
  }  
}

the following statements  
Point p = new Point(10, 10);  
object box = p;  
p.x = 20;  
Console.Write(((Point)box).x);  
will output the value 10 on the console because the implicit boxing operation that occurs in the   
assignment of p to box causes the value of p to be copied. Had Point instead been declared a class, the   
value 20 would be output because p and box would reference the same instance.

### Unboxing conversions

An unboxing conversion permits an explicit conversion from type object to any value-type or from any   
interface-type to any value-type that implements the interface-type. An unboxing operation consists of   
first checking that the object instance is a boxed value of the given value-type, and then copying the   
value out of the instance.  
unboxing conversion of an object box to a value-type G consists of executing the expression   
((G\_Box)box).value.

Thus, the statements  
object box = 12;  
int i = (int)box;

conceptually correspond to  
object box = new int\_Box(12);  
int i = ((int\_Box)box).value;

For an unboxing conversion to a given value-type to succeed at run-time, the value of the source   
argument must be a reference to an object that was previously created by boxing a value of that   
value-type. If the source argument is null or a reference to an incompatible object, an   
InvalidCastException is thrown.

### CONCLUSION :

This type system unification provides value types with the benefits of object-ness without introducing   
unnecessary overhead.  
For programs that don’t need int values to act like objects, int values are simply 32-bit values. For   
programs that need int values to behave like objects, this capability is available on demand. This   
ability to treat value types as objects bridges the gap between value types and reference types that   
exists in most languages.

Hi There,

Simple Example.

Boxing: Convert Value Type to Reference Type

e.g

C#

int i = 123;

object 0 = (object)i;                                ' Here you do boxing

VB

Dim i as Integer = 123

Dim o as Object = Ctype(i, Object)          ' Here you do boxing

UnBoxing: Vise Versa, Reference Type to Value Type

C#

object o = 123;

int i = (int)o;                                           ' Here you do unboxing

VB

Dim o as Object = 123

DIm i as Integer = CType(o, Integer)        ' Here you do unboxing

 ----------------------------------------------------------------

Example not boxing unboxing.

C#

Control c = new Control();

object o = (object)c;

VB

Dim c As New Control

Dim o As Object = CType(c, Object)

This is not boxing unboxing is because you are casting an object to an object (reference type to reference type )

Hope this clear enough

## Introduction

In this article I will explain the concepts of Boxing and UnBoxing. C# provides us with Value types and Reference Types. Value Types are stored on the stack and Reference types are stored on the heap. The conversion of value type to reference type is known as **boxing** and converting reference type back to the value type is known as **unboxing.**

Let me explain you little more about Value and Reference Types.

## Value Types

Value types are primitive types that are mapped directly to the FCL. Like Int32 maps to System.Int32, double maps to System.double. All value types are stored on stack and all the value types are derived from System.ValueType. All structures and enumerated types that are derived from System.ValueType are created on stack, hence known as ValueType.

## Reference Types

Reference Types are different from value types in such a way that memory is allocated to them from the heap. All the classes are of reference type. C# new operator returns the memory address of the object.

## Examples

Lets see some examples to have a better understanding of Value Types and Reference Types. Since we know that all ValueTypes are derived from System.Valuewe can write something like this:

System.ValueType r = 5;

So what do you think about the above line of code. Will it compile ? Yes it will compile. But wait what type is it cause I don't remember any type which is called System.ValueType since its a base class from which all value types inherit. So is it Int32, Int64,double, decimal etc. It turns out that the type for variable 'r' is System.Int32. The Question arrises why Int32 and why not Int16. Well its because it is mapped to Int32 by default depending upon the Initial value of the variable.

You cannot write something like this since System.ValueType is not a primitive type its a base class for primitive value types and these mathematical operations can be performed on primitive types.

System.ValueType r = 10;

r++;

In the above example I told you that variable 'r' will be a System.Int32 variable but if you don't believe me than you can find out yourself using the GetType() method:

System.ValueType r = 5;

Console.WriteLine(r.GetType()) // returns System.Int32;

Here are few samples you can try on your own:

|  |
| --- |
| System.ValueType r = 23.45;          Console.WriteLine(r.GetType()); // what does this print         //-------------------------------------------------------         System.ValueType r = 23.45F;          Console.WriteLine(r.GetType()); // What does this print         //-------------------------------------------------------         System.ValueType r = 2U;          Console.WriteLine(r.GetType()); // What does this print         //-------------------------------------------------------         System.ValueType r = 'c';         Console.WriteLine(r.GetType()); // What does this print         //-------------------------------------------------------         System.ValueType r = 'ac';         Console.WriteLine(r.GetType()); // tricky          //-------------------------------------------------------         System.ValueType r = "Hello World";          Console.WriteLine(r.GetType()); // tricky |

## Boxing

Lets now jump to Boxing. Sometimes we need to convert ValueTypes to Reference Types also known as boxing. Lets see a small example below. You see in the example I wrote "**implicit boxing**" which means you don't need to tell the compiler that you are boxing Int32 to object because it takes care of this itself although you can always make explicit boxing as seen below right after implicit boxing.

|  |
| --- |
| Int32 x = 10;          object o = x ;  // Implicit boxing         Console.WriteLine("The Object o = {0}",o); // prints out 10         //-----------------------------------------------------------         Int32 x = 10;          object o = (object) x; // Explicit Boxing         Console.WriteLine("The object o = {0}",o); // prints out 10 |

## Unboxing

Lets now see UnBoxing an object type back to value type. Here is a simple code that unbox an object back to Int32 variable. First we need to box it so that we can unbox.

|  |
| --- |
| Int32 x = 5;          object o = x; // Implicit Boxing         x = o; // Implicit UnBoxing |

   So, you see how easy it is to box and how easy it is to unbox. The above example first boxs Int32 variable to an object type and than simply unbox it to xagain. All the conversions are taking place implicitly. Everything seems right in this example there is just one small problem which is that the above code is will not compile. You cannot Implicitly convert a reference type to a value type. You must explicitly specify that you are unboxing as shown in the code below.

|  |
| --- |
| Int32 x = 5;          object o = x; // Implicit Boxing         x = (Int32)o; // Explicit UnBoxing |

Lets see another small example of unboxing.

|  |
| --- |
| Int32 x = 5; // declaring Int32         Int64 y = 0; // declaring Int64 double         object o = x; // Implicit Boxing         y = (Int64)o; // Explicit boxing to double         Console.WriteLine("y={0}",y); |

This example will not work. It will compile successfully but at runtime It will generate an exception of System.InvalidCastException. The reason is variable x is boxed as Int32 variable so it must be unboxed to Int32 variable. So, the type the variable uses to box will remain the same when unboxing the same variable. Of course you can cast it to Int64 after unboxing it as Int32 as follows:

|  |
| --- |
| Int32 x = 5; // declaring Int32         Int64 y = 0; // declaring Int64 double         object o = x; // Implicit Boxing         y = (Int64)(Int32)o; // Unboxing and than casting to double         Console.WriteLine("y={0}",y); |

I am sure that you all have grasp the basic understanding of Boxing and Unboxing. Happy Coding and practice a lot !

### Framework Class Library (FCL)

.NET Framework provides huge set of Framework (or Base) Class Library (FCL) for common, usual tasks. FCL contains thousands of classes to provide the access to Windows API and common functions like String Manipulation, Common Data Structures, IO, Streams, Threads, [Security](http://www.programmersheaven.com/2/FAQ-DOTNET-basic-components), Network Programming, Windows Programming, Web Programming, Data Access, etc. It is simply the largest standard library ever shipped with any development environment or programming language. The best part of this library is they follow extremely efficient OO design (design patterns) making their access and use very simple and predictable. You can use the classes in FCL in your program just as you use any other class and can even apply inheritance and polymorphism on these.

# The Dot Net Base Class Libraries (BCL)

|  |
| --- |
| [.Net Tutorials Page](http://www.ninestein.com/dotNet/NinesteinDotNetStudents.html) |

The Base Class Libraries (BCL) provides the fundamental building blocks for any application you develop, be it an ASP.Net application, a Windows Forms application or a Web Service. The BCL generally serves as your main point of interaction with the runtime. BCL classes include:

|  |  |
| --- | --- |
| **Namespace** | **Description** |
| **System** | This namespace includes all the essential support you need for your programming, including base types (String, Int32, DateTime, Boolean, etc.), essential environmental support, and math functions, to name a few |
| **System.CodeDom** | all the support necessary to be able to create code, and run it, on the fly |
| **System.Collections** | The System.Collections namespace contains interfaces and classes that define various containers, such as lists, queues, bit arrays, hashtables and dictionaries. |
| **System.Diagnostics** | All the classes you need to diagnose your application, including event logging, performance counters, tracing, and process management APIs. |
| **System.Globalization** | This namespace includes fundamental support for Globalization, used throughout the rest of the Framework |
| **System.IO** | Includes fundamental Stream support which can be used by anyone, and then specifically targets the FileSystem (via File and Directory manipulation classes), SerialPorts, and Decompression |
| **System.Resources** | Used to allow an application to be translated into multiple languages, and then display the appropriate text based upon the current users language selection |
| **System.Text** | This namespace includes support for encodings, and Stringbuilder |
| **System.Text.RegularExpressions** | This namespace includes regular expression support, for robust parsing and matching of string data |