Object Oriented Programming in C#

Topics covered

Classes and Objects

Instance Variables, Methods, Constructors, Properties, Access Specifiers, Static members and methods

Inheritance

Levels of Inheritance, Constructor and Inheritance, Polymorphism, Interfaces, Abstract classes, Delegates, Indexers, Sealed Classes, Exception handling

Collections and Generics

Bounded and Unbounded Collections, Generic Programming-Generic classes, Functions, Constraints on Generic Programming

Object

- Objects encapsulate part of the application which can be a process, a chunk of data or an abstract entity
- Objects in C# are created from types with a special name in OOP called as class
- Also called as a real named instance
- The process of creation of an object of a class is instantiation

Class

- An art of systematic arranging of information and behaviour into a meaningful entity
- Class helps to produce classification
- Encapsulation supports classification
- Defines asbstract charactreristics of an object
- A template for object creation
- Classes are means to provide modularity

Namespaces

- A way to organize related classes and other types
- A logical grouping of elements
- A wrapper that wraps one or more structural elements to make them unique and differentiated from other elements
- Accpeted format is CompanyName.ProjectName.SystemSection
- Ex:Infotel.BillingApp.Customer.CustomerInfo

  (fully qualified name)

The “using” Directive

- An abbreviate a class by Prefixing a keyword specified as “using” to a class's namespace
- Ex: using System;

  using Infotel.BillingApp.Customer;
- Most widely used namespace is “System”
- A care should be taken while naming a namespace
- Microsoft recommends the format for namespace names as;

  <CompanyName>.<TechnologyName>

Namespace Aliases

- Using keyword can also be used to assign aliases
- Syntax: using alias=Namespace name;
- Ex:

```
using System
using CustData=Infotel.BillingApp.Customer;
class Test
{
  public static int main()
  {
     CustData : : CustomerDetail c1=new CustData : : CustomerDetail();
     .....
     return 0;
  }
}
```

Main() method

- An Entry point method
- Must be static method of a class or a struct
- Must have a return type either int or void
- Default access modifier is private
- main() method can also be assigned as public explicitly
- Multiple main() methods return a compile time error
- Accepts a String array argument as;

  static void main(String[] args/arg/ar/../..)

CLASS DEFINITIONS IN C#

C# uses the class keyword to define classes:

```
class MyClass
{
// Class members.
}
```

- By default, classes are declared as internal (explicit declaration is optional)

```
internal class MyClass
{
// Class members.
}
```

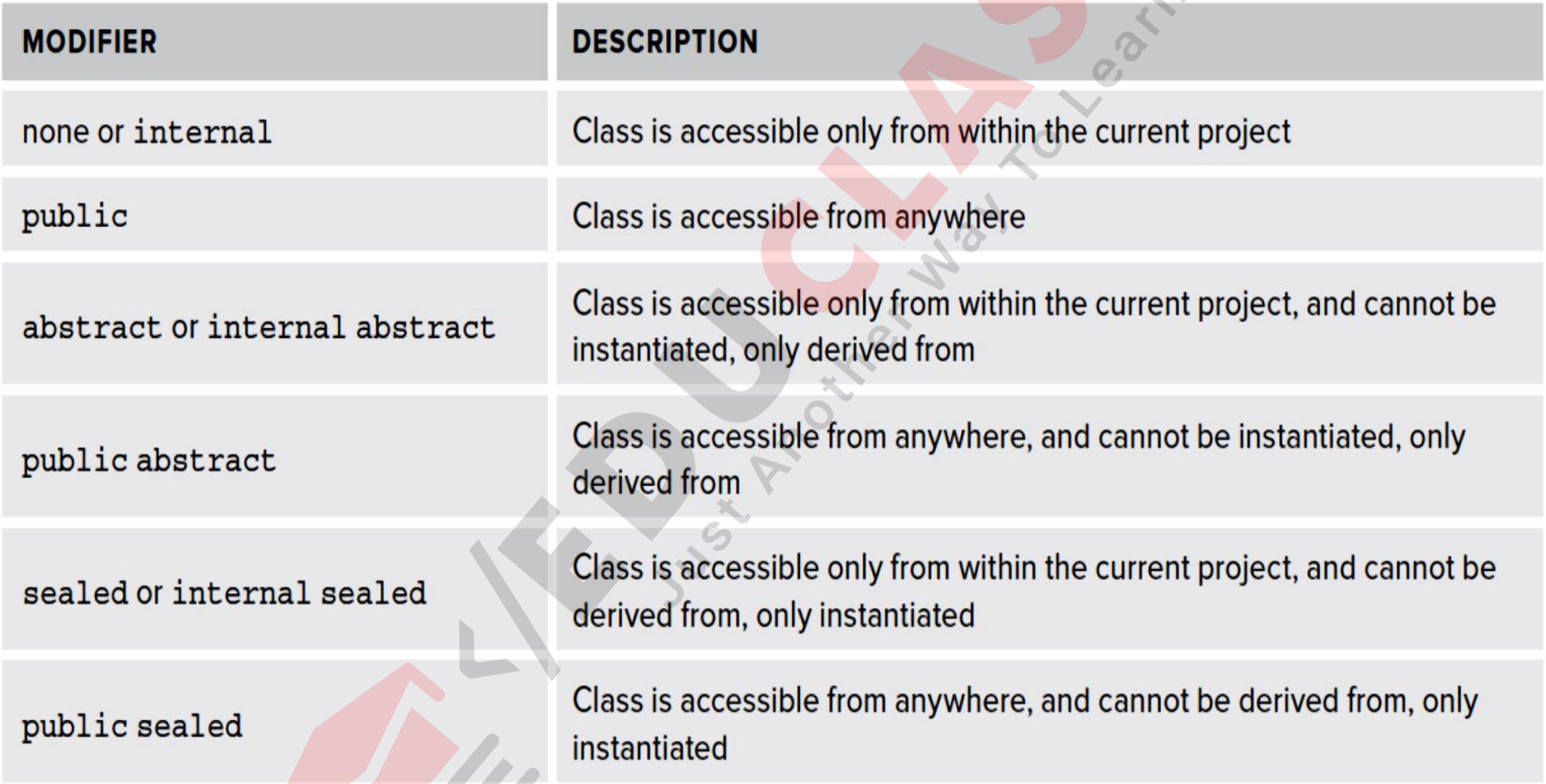
Various class access specifications

```
1) public class MyClass
    {
    // Class members.
    }
2) public abstract class MyClass
{
   // Class members, may be abstract.
}
```

Various class access specifications

```
3) public sealed class MyClass
{
// Class members.
}
4) public class MyClass : MyBase
{
// Class members.
}
```

Access Modifiers for Class Definitions



MODIFIER	DESCRIPTION
none or `internal`	Class is accessible only from within the current project
`public`	Class is accessible from anywhere
`abstract` or `internal abstract`	Class is accessible only from within the current project, and cannot be instantiated, only derived from
`public abstract`	Class is accessible from anywhere, and cannot be instantiated, only derived from
`sealed` or `internal sealed`	Class is accessible only from within the current project, and cannot be derived from, only instantiated
`public sealed`	Class is accessible from anywhere, and cannot be derived from, only instantiated

Defining and Using a Class

```
class Circle
{
    int radius;
    double Area()
    {
        return Math.PI * radius * radius;
     }
}
```

Naming and Accessibility

- Identifiers that are public should start with a capital letter. For example, Area starts with “A” (not “a”) because it’s public. This system is known as the PascalCase naming scheme.
- Identifiers that are not public (which include local variables) should start with a lowercase letter. For example, radius starts with “r” (not “R”) because it’s private. This system is known as the camelCase naming scheme

Constructors

- A *constructor* is a special method that runs automatically when you create an instance of a class
- *new* Keyword is used to construct an object at runtime
- Construction happens in 3 steps:

  1) Runtime grabs a chunk of memory from OS

  2) Filling of fields defined by class

  3) Invoke a constructor
- Has same name as class and accepts parameters
- Returning of a value is not allowed
- Every class in C# has a default Constructor

Example: Default Constructor

```
class Circle
{
        private int radius;
        public Circle() // default constructor
        {
            radius = 0;
        }
        public double Area()
        {
            return Math.PI * radius * radius;
        }
}
```

Object Creation

```
Circle c; // Create a Circle variable
c = new Circle(); // Initialize it
```

Example: Overloading Constructors

```
class Circle
{
      private int radius;
      public Circle()   // default constructor
      {
             radius = 0;
      }
      public Circle(int initialRadius) // overloaded constructor
      {
            radius = initialRadius;
      }
      public double Area()
      {
           return Math.PI * radius * radius;
      }
}
```

Destructor

- A method that is called to destroy the objects that are no longer in use
- A destructor is declared using (~) tilde sign followed by the name of destructor
- Ex:

```
class destdemo{
     static void main(String[] args) {
         Destruct obj1 = new Destruct();
     }
  }
  class Destruct{
      ~Destruct()
      {
              Console.WriteLine(“Destructor is called”);
      }
   }
```

this keyword

- Refers to the current instance of a class
- Cannot be used with static members
- “this” keyword is followed by “.” operator for accessing instance members
- Ex: public class Account{

```
                double accBalance;
                public void Balance() {
                        this.accBalance=10000;
                }
        }
```

Static Classes

- A static class or its members do not need any object to call them
- Calling is possible through direct using a class name
- Use of static keyword to define a class and its members
- Ex: static class employee{

```
            public static int id;
            public static int tele_phone;
      }
```

- Ex: Math.sqrt(25);
- Static classes are sealed (No inherit capability)

Static Constructor

- Doesn’t accept any parameters and access modifiers.
- Invokes automatically, whenever we create a first instance of class.
- Invoked by CLR so we don’t have a control on static constructor execution order in c#.
- Only one static constructor is allowed to create.

Partial Classes

circ1.cs

```
partial class Circle
{
    public Circle() // default constructor
    {
        this.radius = 0;
    }
    public Circle(int initialRadius) // overloaded constructor
    {
        this.radius = initialRadius;
    }
}
```

Partial Classes cont..

circ2.cs

```
partial class Circle
{
    private int radius;
    public double Area()
    {
        return Math.PI * this.radius * this.radius;
    }
}
```

Object Oriented Programming concepts

Encapsulation

- Process of hiding the irrelevant information of a specific object to a user
- Process of hiding internal facts
- As per OOP, encapsulation is wrapping up data and members of a class
- Restricts users from sharing & manipulating the data resulting into data protection
- Prevents data Corruption
- Binding member variables and methods into a single unit
- Increases the maintainability

Properties

- Properties are used to encapsulate the fields and data in a class
- Safer and Controlled approach as compared to field value accessing using assignment
- A property is a cross between a field and a method
- Syntactically access to Properties is same as fields access
- Access to the fields and properties is done by the operator . (dot)
- Access is achieved by keywords get and set
- Accessibility of a property can *public*, *private*, or *protected*

Syntax for Property declaration

```
AccessModifier Type PropertyName
{
    get
    {
      // read accessor code
    }
   set
   {
     // write accessor code
   }
}
```

Property Example

```
public class Person
{
    public string FirstName
    {
        get
        {
          return firstName;
        }
        set
        {
           firstName = value; }
        }
    }
}
```

Properties Example cont..

```
public class Button: Control
{
   private string caption;
   public string Caption {
      get {
         return caption;
          }
      set {
         caption = value;
         Repaint();
      }
   }
}
```

```
Button b = new Button();
b.Caption = "OK";
String s = b.Caption;
```

Indexers

- An *indexer* is a special kind of property that you can add to a class to provide array-like access
- An indexer encapsulates a set of value
- Enables objects of a class to access its members using an index notation
- Indexers can use non-numeric subscripts
- Ex: public int this [string name] { ... }
- Indexers can be overloaded whereas arrays cannot

Indexers cont..

- Access through a variable name and square bracket
- Syntax:

```
<access modifier> <Return Type> this [arg list] {
      get  {  //code for get  }
      set  {  //code for set   }
 }
```

- Declaration using this keyword:

```
public int this[string key]
{          get { return storage.Find(key); }
           set { storage.SetAt(key, value); }
 }
```

Indexers cont..

- Syntax:

```
var item = someObject["key"];
someObject["AnotherKey"] = item;
```

- Access modifier can be Private, Public, Protected, or Internal
- this keyword shows the object of current class
- Argument List is parameters passed (Atleast One Parameter required)
- Multiple type parameters are allowed (int, enum, String)
- All indexer in same class should have different signatures
- get and set portions are accessors

Inheritance

- Inheritance promotes reusability of code
- Helps to eliminate redundant code
- A class derives properties from another class
- Single Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Multiple Inheritance (only through Interface)

Polymorphism

- Allows you to invoke methods of a derived class through base class reference during runtime (Dynamic polymorphism)
- Provides different implementations of methods in a class that are called through the same name (Static polymorphism)
- Ex: Method overloading, Operator overloading, Indexer overloading

Abstraction

- Abstraction and Encapsulation are complimentary to each other
- Process of showing general information and hiding the complex information
- Managing the complexity of the code
- Decomposing complex systems into smaller components
- Ex: abstract classes and methods

Creating Interfaces

- An interface does not contain any code or data; it just specifies the methods and properties that a class that inherits from the interface
- An interface enables you to completely separate the names and signatures of the methods of a class from the method's implementation

Defining an Interface

```
interface IComparable
{
        int CompareTo(object obj);
}
```

- Never specify an access modifier (*public*, *private*, or *protected*)
- Replace the method body with a semicolon

Implementing an Interface

```
interface IntfAccountBal
{
        int getBal();
}
class ImplInter : IntfAccountBal
{
         ...
         public int getBal()
         {
              return balaAmount;
         }
}
```

Delegates

- C# handles callback functions through delegate
- A special type of object that contains details of method rather than data
- A delegate holds three pieces of information

  - The name of method on which it makes call

  - The arguments (if any)

  - The return value (if any)
- Create and Use a delegate

  1) Declare a delegate

  2) Define delegate method

  3) Creating delegate objects

  4) Invoking delegate objects

Delegates cont..

- A delegate holds reference of a method
- All delegates are implicity derived from System.Delegate class
- Declared using a delegate keyword followed by mehtod signature
- Syntax: <access modifier> delegate <return type> <delegate_name>(<parameters>)
- Ex: public delegate void Print(int value);
- Ex: public delegate void Compute(int x, int y);
- Delegate types are implicitly sealed

Defining Delegate Methods

- A method whose signature matches the delegate signature exactly
- Method can be a static or an instance
- Ex: public static void Add(int x, int y)

```
    {
        ....
    }
    public void Multiply(int x, int y)
    {
        ....
    }
```

Creating and invoking Delegate objects

- Syntax:

  Delegate-name object-name

  = new delegate-name(expression);
- expression can be a name of a method or an object of a delegate type
- Signature of method passed should be same as of delegate
- Ex: Compute cmp1

  = new Compute(DelegateTest.Add);
- Ex: DelegateTest dt = new DelegateTest();

  Compute cmp2= new Compute(dt.Multiply);
- Ex: cmp1(30,20);

  cmp2(10,15);

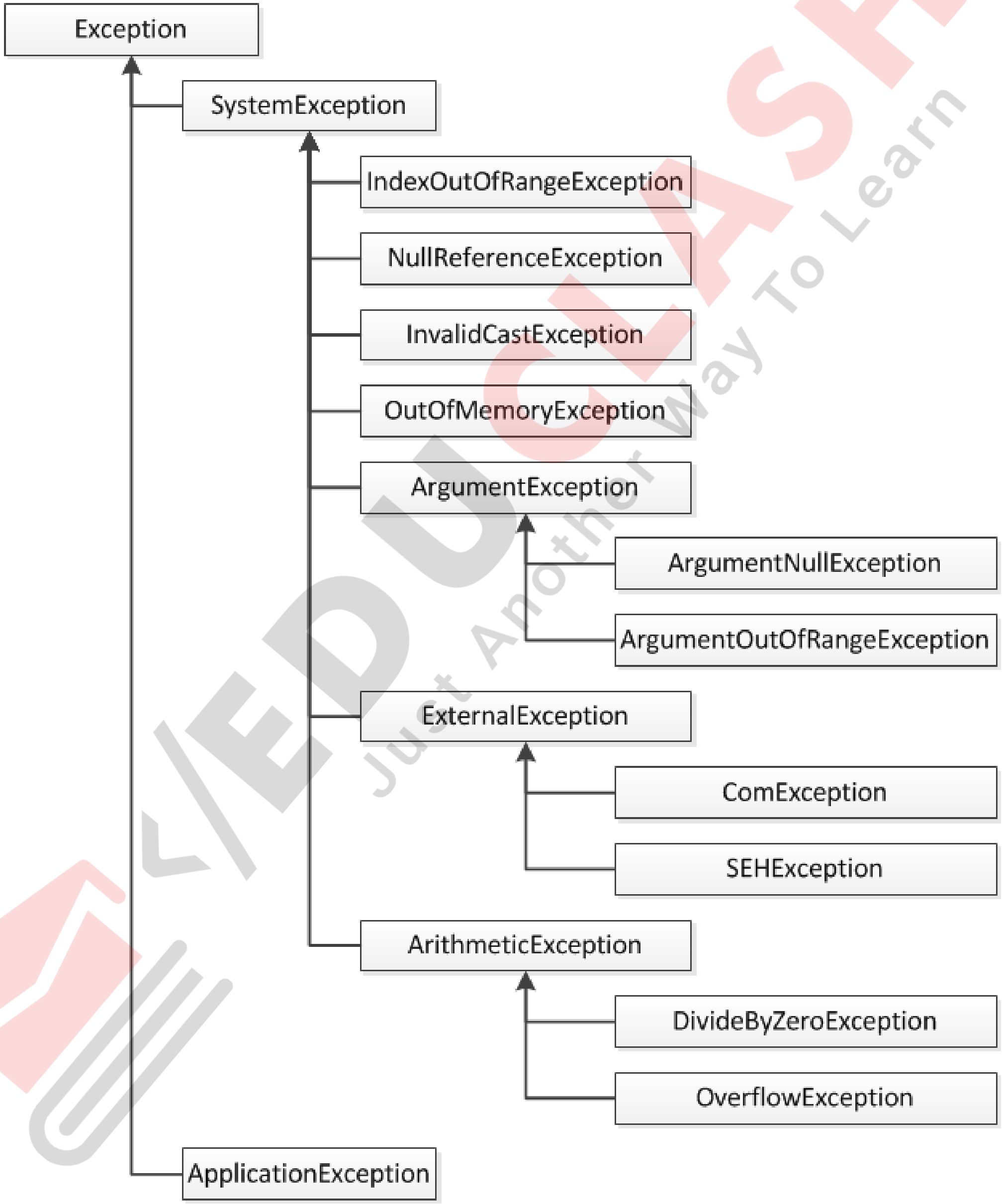
Exception Handling

- Exception is a runtime error arises due to some abnormal conditions
- Exception Handling relates to Capturing & Handling of runtime errors
- Compile time errors occur during compilation of a program
- Exceptions can be handled by using;

1) The try...catch...finally statement

2) The throw statement

Exception Hierarchy in c#



The try...catch...finally statement

- try encloses the set of statements that can cause exception
- catch block handles the occured exception
- A try block can have single or multiple catch blocks
- The finally block results into absolute execution of statements
- Only one finally block is allowed for a try block

The try...catch...finally statement

- Ex:

```
try {
    div= 100/number;
}
catch (DivideByZeroException dbze) {
    Console.WriteLine(“Exception occured”);
}
finally {
    Console.WriteLine(“Result is:”+ div);
    Console.ReadLine();
}
```

The throw statement

- throw clause is used to raise an exception in case an error occurs in a program
- throw takes only a single argument
- If a throw statement is encountered, a program terminates
- throw clause can also be used to throw an exception programatically
- throw keyword is used to throw an exception

throw an Exception explicitly

```
try {
   throw new DivideByZeroException();
}
catch(DivideByZeroException) {
   Console.WriteLine(“Exception Occured”);
}
```

Checked and Unchecked Statements

- Used to check memory overflow exceptions
- checked keyword is used to check the overflow for integral type arithmetic operations and conversions
- Ex: Value of a variable exceeds the required length
- unchecked keyword ignores the overflow-checking

Collections

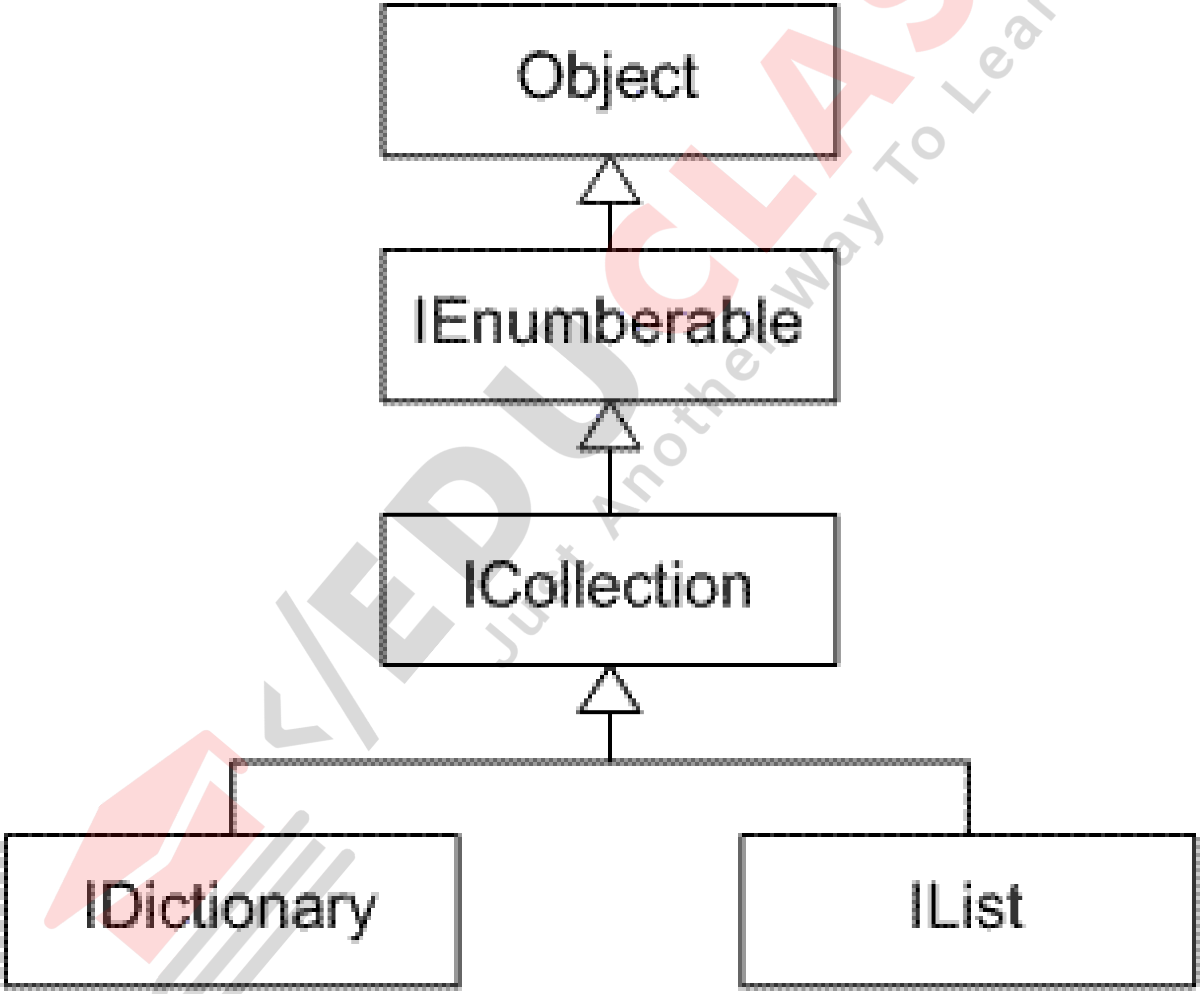
Collection Classes

- Collection classes are used for maintaining lists of objects
- A collection can store & retrieve different types of Data
- Provides automatic memory management and capacity expansion
- It is possible to create a custom collection class
- A collection is an object that simply allows you to group other objects.
- Collection based classes provide support for Stacks, Queues, Lists, and Hash Tables.

Properties of Collection Classess

- Collection classes are defined as part of the System.Collections or System.Collections.Generic namespace.
- Most collection classes derive from the interfaces ICollection, IComparer, IEnumerable, IList, IDictionary, and IDictionaryEnumerator and their generic equivalents.
- Using generic collection classes provides increased type-safety and in some cases can provide better performance, especially when storing value types.

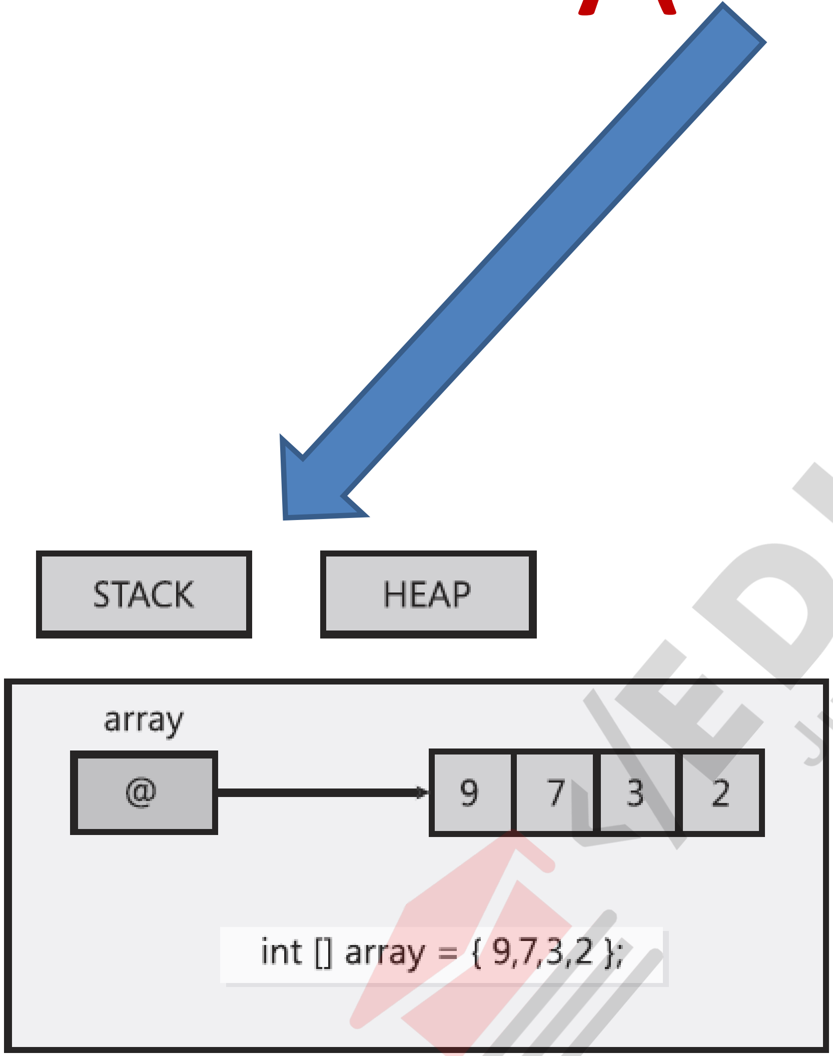
Interface Hierarchy



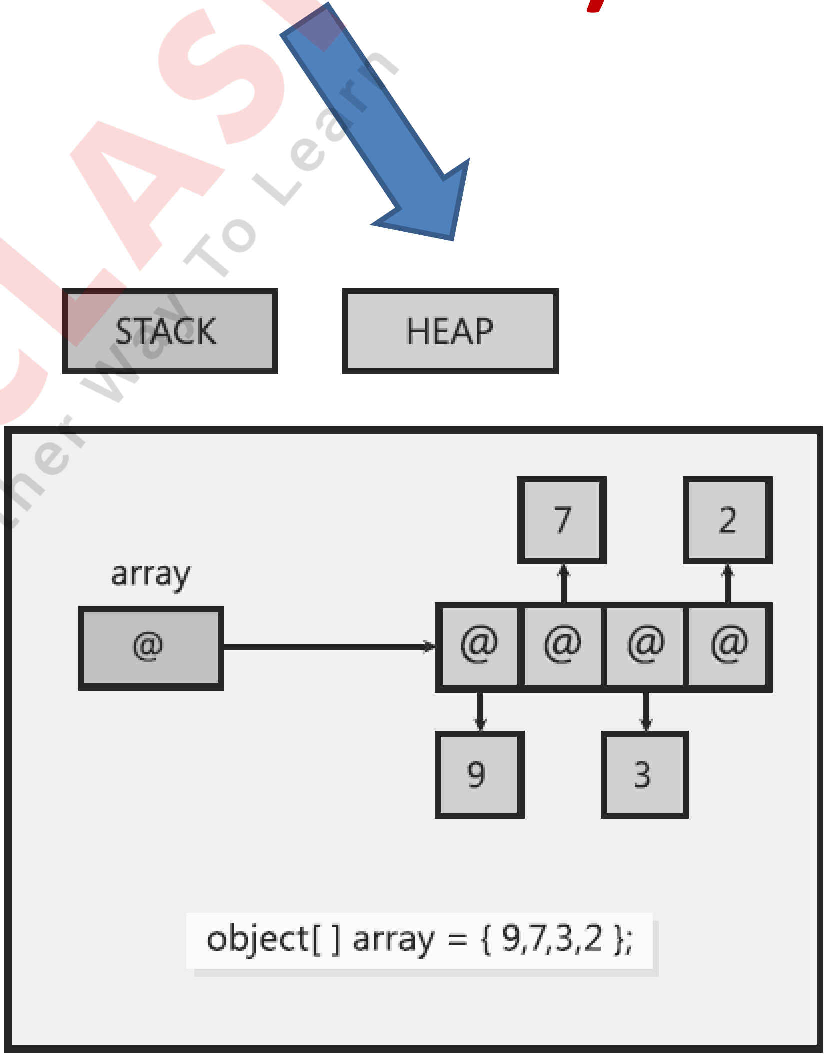
Collection classes in .Net class library

- ArrayList Class
- HashTable Class
- Queue Class
- Stack Class
- SortedList Class
- A custom collection class can be created by implementing ICollection interface.

Memory (Array Vs collection)



int [] array = { 9,7,3,2 };



object[ ] array = { 9,7,3,2 };

Dynamic Lists

- .NET Framework offers the generic class List<T>.
- List<T> class implements the IList, Icollection, IEnumerable, Ilist<T>,Icollection<T>, and Ienumerable<T> interfaces.
- EX:

public class Racer : Icomparable<Racer>, Iformattable

{//………..}

- We can create a list for above class using List<T> class
- Ex: var racers =new List<Racers>([param’s]);

ArrayList class

- Ordered collection of object indexed in individual manner
- An alternative to an array
- Dynamic manipulation is possible
- Accepts null as valid value
- **Declaration of an Array List**

  ArrayList a1 = new ArrayList();
- **Data manipulation methods**

  ArrayList.add(element);

  a1.Add(1); a1.Add("Example") ; a1.Add(true);

  a1.Remove(7);a1.RemoveAt(1); al.Sort();
- **Properties**

  count, Capacity, etc.

Array Vs ArrayList

Array	ArrayList
Strongly Typed	Not
Size is Fixed	Dynamic
Set or get a value of any one element at a time	Wide range of methods for Manipulation on multiple elements
Multiple Dimension	Single Dimension
Casting is not required	Casting Required

Hashtable class

- Similar to ArrayList except the accessing procedure is through a key
- Each item in Hashtable object has a key/value pair
- Each key must be unique
- Keys can be short strings or integers
- Add or Retrieve items is possible in Hashtable class
- Key cannot be null, but a value can be null
- Similar to Dictionary but lower in performance

SortedList class

- A combination of an Array and Hashtable
- Items from a list can be accessed using an index or a Key
- When using indices - object acts as ArrayList
- When using Keys - object acts as Hashtable
- SortedList is sorted by default (key wise)
- No explicit sort method available

Stack class

- A special case collection which represents a last in first out (LIFO) concept
- Process of adding to a stack is push operation
- Process of removal is pop operation
- **Declaration of the stack**

  Stack st = new Stack();
- **Adding elements to the stack**

  Stack.push(element);
- **Removing elements from the stack**

  Stack.pop();

Queue class

- Queue class follows First In First Out (FIFO) concept in data storage
- The methods that add or remove items from a Queue object are called Enqueue & Dequeue
- **Declaration**

  Queue q = new Queue();
- **Adding elements to Queue:** q.Enqueue('A');
- **Removal of elements from a Queue:**

  q.Dequeue();

Generic Programming

- A technique with which you can delay the specification of type
- labels are defined inspite of specific data type
- The label is replaced with specified datatype at run time when a generic method or class used
- Syntax: public class ClassName<T>
- Ex: List<String> l1= new List<String>();

Generic Classes

- The List<T>class
- The LinkedList<T>class
- The SortedList<TKey, TValue>class
- The Dictionary<TKey, TValue>class
- The SortedDictionary<TKey, TValue>class
- The Stack<T>class
- The Queue<T>class
- The HashSet<T>class

Constraints on Generics

- Constraints are used in Generics to restrict the types
- Constraints can be applied using the where keyword.
- Six types of constraints can be applied: class, struct, new(), base class name, interface and derived type.
- Multiple constraints also can be applied

Types of Generic Constraints

Constraint	Description
where T : class	Type must be reference type.
where T: struct	Type must be value type
where T: new()	Type must have public parameterless constructor
where T: <base class name>	Type must be or derive from the specified base class
where T: <interface name>	Type must be or implement the specified interface
where T: U	Type supplied for T must be or derive from the argument supplied for U

Generic Class with Constraints Example

```
public class GenericClass<T> where T: class
{
  public T EmpName;
  public void genericMethod(T EmpDept, T EmpSkill)
  {
    Console.WriteLine(“Emp Name: ” + EmpName);
    Console.WriteLine(“Emp Dept: “ + EmpDept);
    Console.WriteLine(“Emp Skill: “ + EmpSkill);
  }
}
```

Methods

- Declaring of Class Method

```
// Method definition
[<modifiers>] [<return_type>]
            <method_name>([<parameters_list>])
{
    // ... Method's body ...
    [<return_statement>];
}
```

Example – Method Declaration

```
int Add(int number1, int number2)
{
        int result = number1 + number2;
        return result;
}
```

Constants

- Once declared and initialized constants always have the same value for all objects of a particular type
- In C# constants are of two types:

1. Constants the values of which are extracted during the compilation of the program (compile-time constants).
2. Constants the values of which are extracted during the execution of the program (run-time constants)

Constants cont..

- Compile-Time Constants

  Ex:

  public const double PI = 3.14159265358979323 85;

- Run-time Constants

  Ex: public readonly double Size; (Allocate a value at runtime)

String implementation in C#

Strings

- Strings are sequences of characters stored in a certain address in memory
- Declared by the keyword string
- Default value is null
- Strings are enclosed in quotation marks
- Used for performing various text processing operations

The System.String Class

- Example of **declaring a string:**

  string greeting = "Hello, C#";

- Representation:

H	e	l	l	o	,		C	#

- Alternative: Creating an array of characters naming it as char[] and fill the elements with characters one by one
- Character Array creation Disadvantages:

  - Filling happens one by one

  - Length of text should be known

  - The text processing is manual

String class

- String as a class compiles as per Object Oriented Programming principles
- Values of String class are stored in dynamic memory (Managed Heap)
- String Variables hold reference of the object in the Heap
- Character sequences stored in string variable are never changing
- Accessibility through an Indexer (Only Read)

String Escaping

- Displaying special characters in source code is called as escaping
- Use of a back slash before quotes
- Ex: string quote = "Book's title is \"Intro to C#\"";
- Escaped Quotes are discarded by Compiler

Creating and Initialising a string

- Instaintiation of a declared string variable
- Un-Initialised strings are not empty
- String are stored in Heap
- Attempt to manipulate a null string will generate a NullRefernceException
- Ways of Initialising variables:
  1. By assigning a string literal.
  2. By assigning the value of another string.
  3. By passing the value of an operation which returns a string.

Creating and Initialising a string

1. Setting a string Literal

```
string website = "http://www.vegfood.org";
```

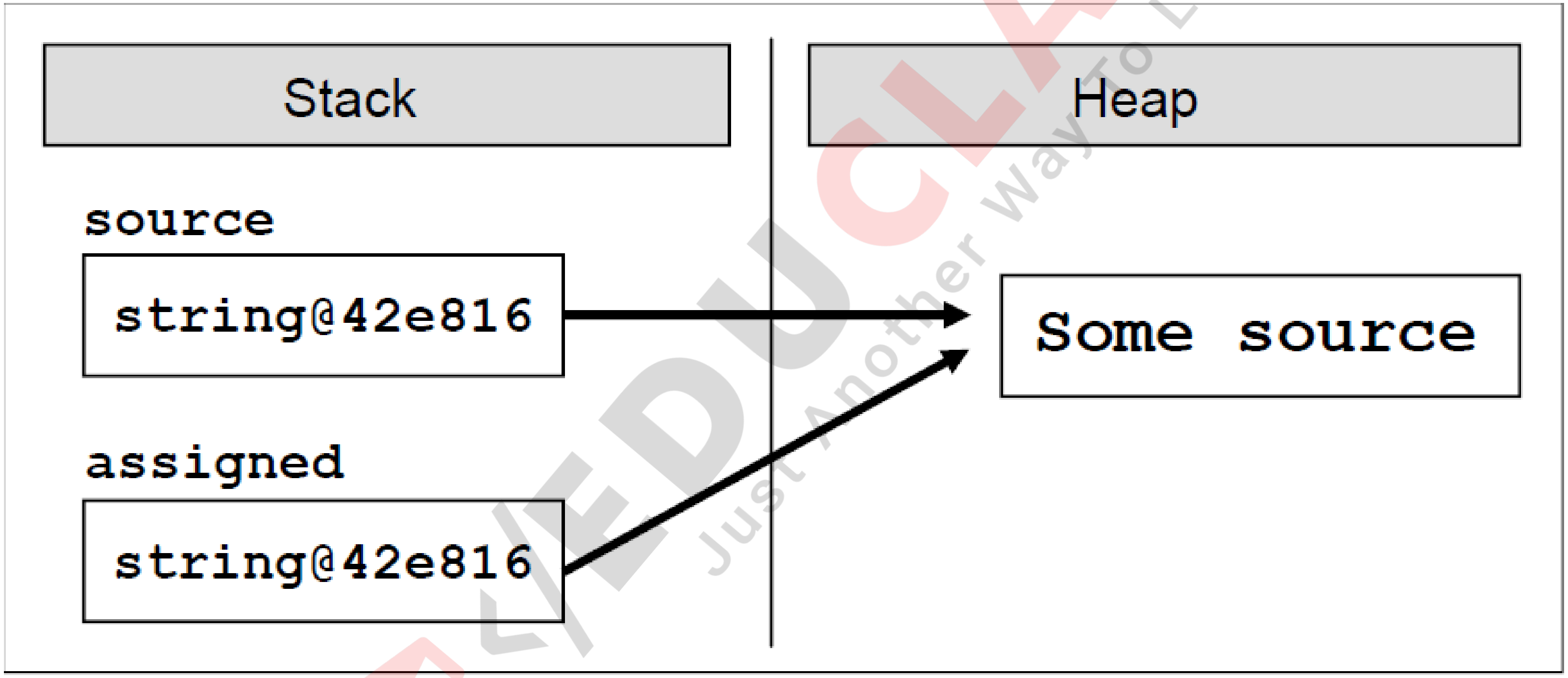
2. Assigning the value of another string

```
string source = "Some source";
string assigned = source;
```

3. By passing the value of an operation which returns a string

```
string email = "xyz@gmail.com";
string info = "My mail is: " + email;
```

Memory Allocation for a string



String Comparison Using Equals method or == operator

```
string word1 = "C#";
string word2 = "c#";
Console.WriteLine(word1.Equals("C#"));
Console.WriteLine(word1.Equals(word2));
Console.WriteLine(word1 == "C#");
Console.WriteLine(word1 == word2);

Console.WriteLine(word1.Equals(word2,
  StringComparison.CurrentCultureIgnoreCase));
```

Comparing Strings in Alphabetical Order

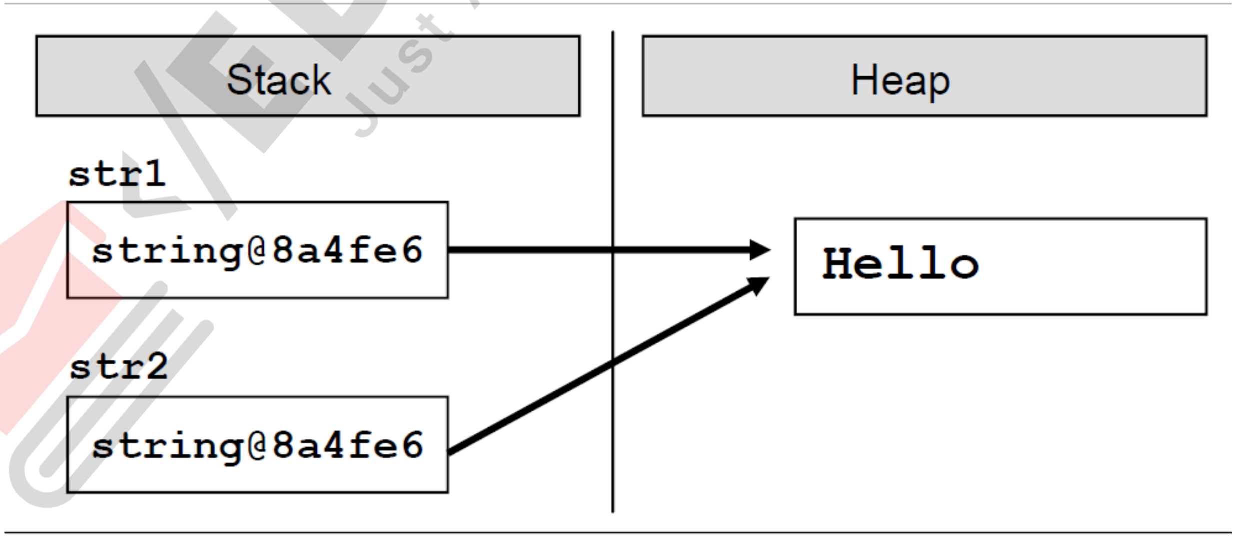
```
string score = "sCore";
string scary = "scary";
Console.WriteLine(score.CompareTo(scary));
Console.WriteLine(scary.CompareTo(score));
Console.WriteLine(scary.CompareTo(scary));
// Console output:
// 1
// -1
// 0
```

Ignore Letter Casing

```
string.Compare(string strA, string strB,
                    bool ignoreCase)
string alpha = "alpha";
string score1 = "sCorE";
string score2 = "score";
Console.WriteLine(string.Compare(alpha, score1, false));
Console.WriteLine(string.Compare(score1, score2, false));
Console.WriteLine(string.Compare(score1, score2, true));
Console.WriteLine(string.Compare(score1, score2,
                                        StringComparison.CurrentCultureIgnoreCase));
// Console output:
// -1
// 1
// 0
// 0
```

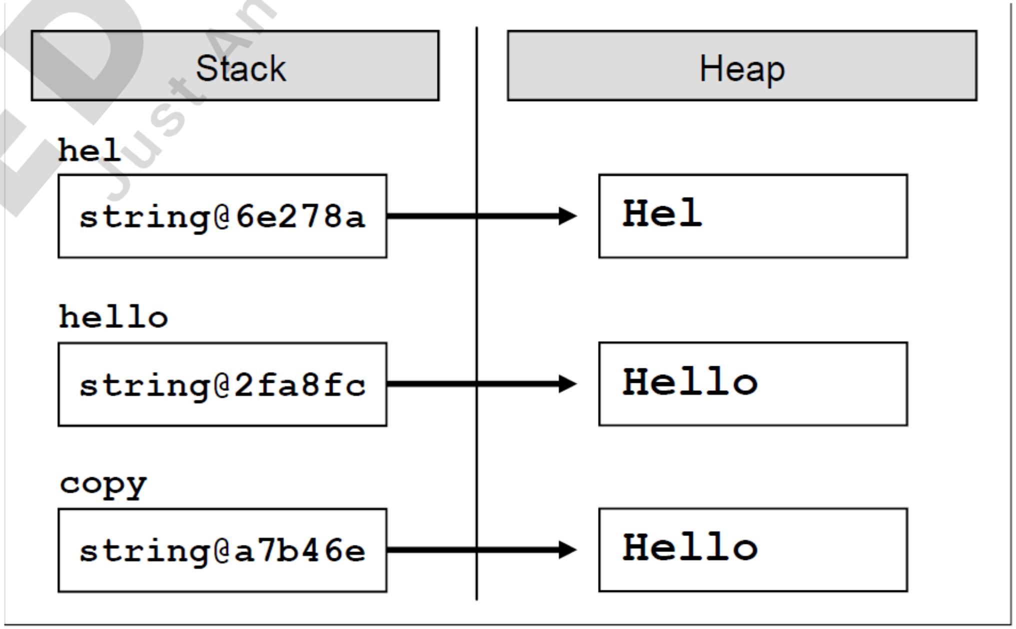
== and != Operators & Memory Usage

```
string str1 = "Hello";
string str2 = str1;
Console.WriteLine(str1 == str2);
// Console output:
// True
```



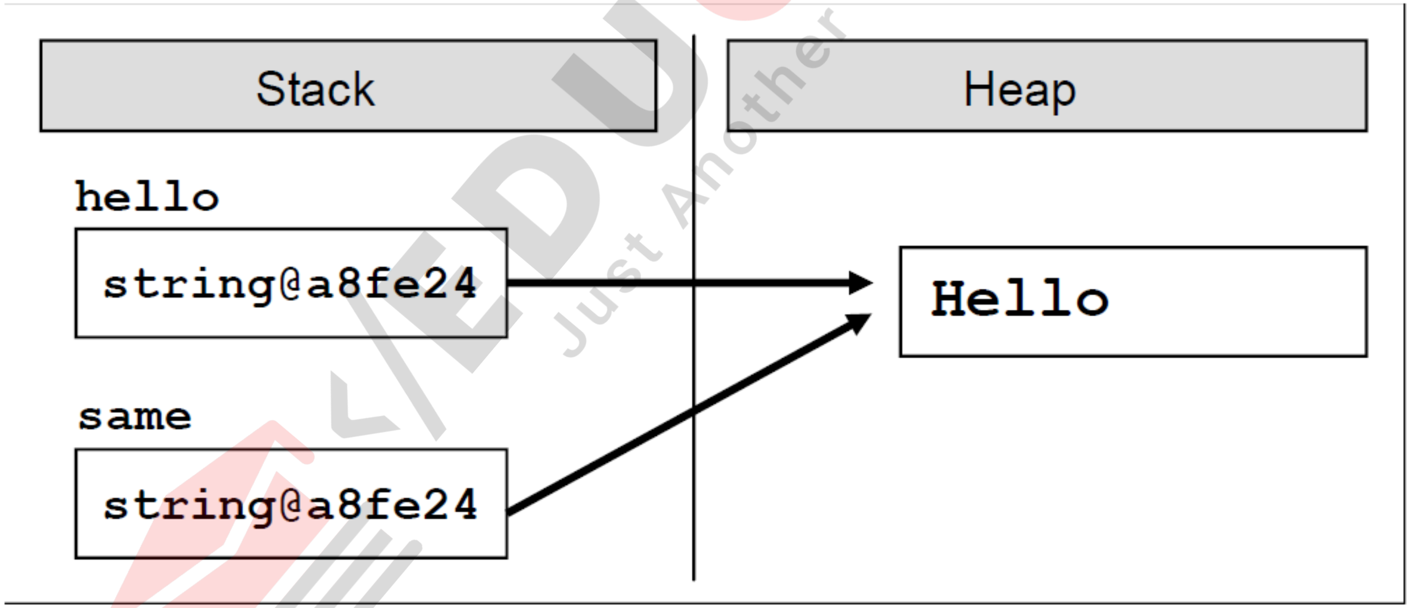
String variable and Literal

```
string hel = "Hel";
string hello = "Hello";
string copy = hel + "lo";
Console.WriteLine(copy == hello);
// True
```



Memory Optimization for Strings (Interning)

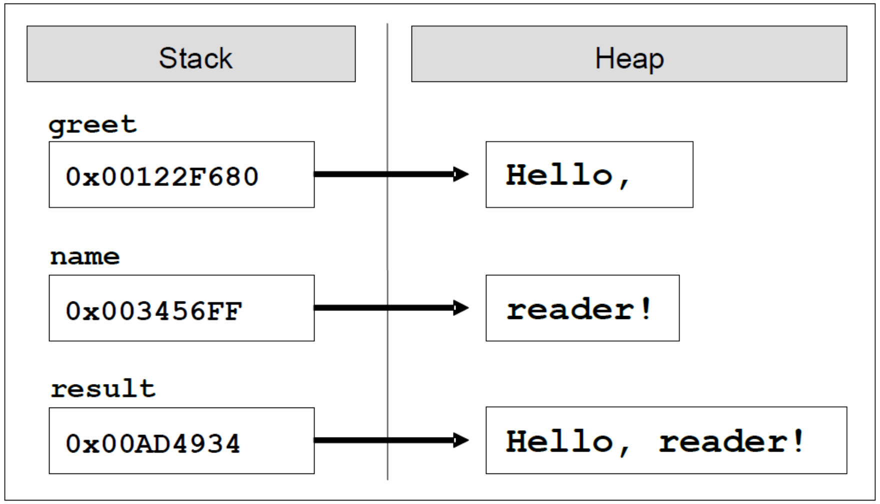
```
string hello = "Hello";
string same = "Hello";
```



Strings Concatenation

- Used to glue two strings
- Ex:

```
string greet = "Hello, ";
string name = "reader!";
string result = string.Concat(greet, name);
string result = greet + name;
```



Change the casing of a string

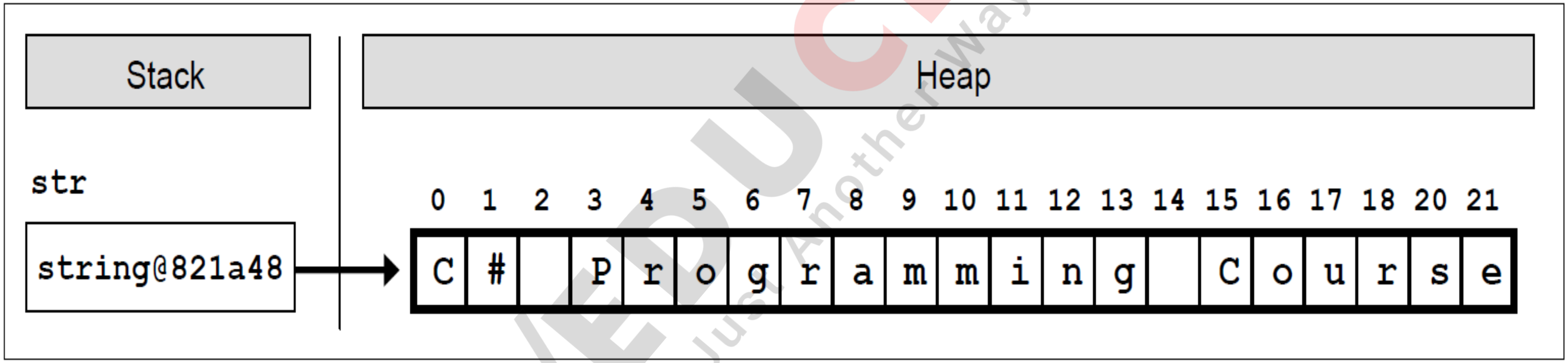
```
string text = "All Kind OF LeTTeRs";
Console.WriteLine(text.ToLower());
// all kind of letters
string pass1 = "PasswoRd";
string pass2 = "PaSSwoRD";
string pass3 = "password";
Console.WriteLine(pass1.ToUpper() == "PASSWORD");
Console.WriteLine(pass2.ToUpper() == "PASSWORD");
// Console output:
// True
// True
```

Search a String within Another String

```
string book = "Introduction to C# book";
int index = book.IndexOf("C#");
Console.WriteLine(index);
// index = 16

string str = "C# Programming Course";
int index = str.IndexOf("C#"); // index = 0
index = str.IndexOf("Course"); // index = 15
index = str.IndexOf("COURSE"); // index = -1
```

Searching a String in Memory



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	20	21
C	#		P	r	o	g	r	a	m	m	i	n	g		C	o	u	r	s	e

Finding All Occurrences of a Substring

```
string quote = "The main intent of the \"Intro C#\"" + "
book is to introduce the C# programming to newbies.";
string keyword = "C#";
int index = quote.IndexOf(keyword);
while (index != -1)
{
Console.WriteLine("{0} found at index: {1}", keyword,
index);
index = quote.IndexOf(keyword, index + 1);
}
```

Extracting a Portion of a String

```
Substring(startIndex, length);


string path = "C:\\Pics\\CoolPic.jpg";
string fileName = path.Substring(8, 7);
// fileName = "CoolPic"
```

Splitting the String by a Separator

```
string listOfBeers = "Amstel, Heineken, Tuborg, Becks";
char[] separators = new char[] {' ', ',', '.'};
string[] beersArr = listOfBeers.Split(separators);
foreach (string beer in beersArr)
{
    if (beer != "")
    {
        Console.WriteLine(beer);
    }
}
```

Replacing a Substring

```
string doc = "Hello, some@gmail.com, " +
"you have been using some@gmail.com in your
registration.";
string fixedDoc =
doc.Replace("some@gmail.com",
"john@smith.com");
Console.WriteLine(fixedDoc);
// Console output:
// Hello, john@smith.com, you have been using
// john@smith.com in your registration.
```

Triming a String

Ex1:

```
string fileData = "        David Allen        ";
string reduced = fileData.Trim();
```

Ex2:

```
string fileData = " 111 $ % David Allen ### s ";
char[] trimChars = new char[] {' ', '1', '$', '%', '#', 's'};
string reduced = fileData.Trim(trimChars);
// reduced = "David Allen"
```

Ex3: `string reduced = fileData.TrimEnd(trimChars);`

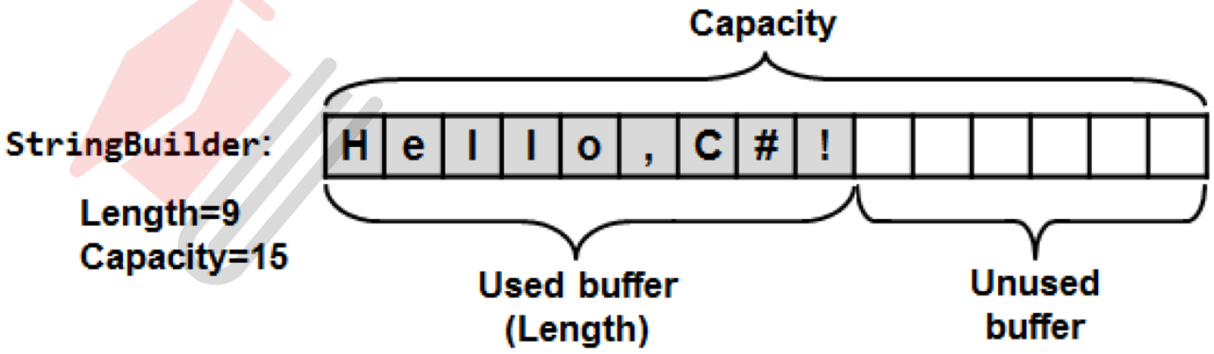
StringBuilder class

- Serves to build and change the Strings
- Used to overcome the string performance problem
- class is build in the form of array of characters
- Same buffer is used to make any changes
- Objects of StringBuilder are mutable
- StringBuilder keeps a buffer with a certain capacity (default 16 characters)

StringBuilder class cont..

- The buffer is implemented as an array of characters
- At any moment part of the characters in the buffer are used and the rest stay in reserve
- If the entire capacity of the buffer is filled, then the buffer is doubled.
- Ex: StringBuilder sb = new StringBuilder(15);

  sb.Append("Hello, C#!");



StringBuilder class Example

```
class ElegantNumbersConcatenator
{
static void Main()
{
Console.WriteLine(DateTime.Now);
StringBuilder sb = new StringBuilder();
sb.Append("Numbers: ");
for (int index = 1; index <= 200000; index++)
{
sb.Append(index);
}
Console.WriteLine(sb.ToString().Substring(0, 1024));
Console.WriteLine(DateTime.Now);
}
}
```

StringBuilder Example

```
public static string ExtractCapitals(string str)
{
         StringBuilder result = new StringBuilder();
         for (int i = 0; i < str.Length; i++)
        {
               char ch = str[i];
               if (char.IsUpper(ch))
               {
                          result.Append(ch);
               }
        }
        return result.ToString();
}
```

Parsing Data

- Converting from text to some other data type (opposite of ToString())
- Parsing Numeric Types:

```
int intValue = int.Parse(text);
bool boolValue = bool.Parse(text);
```

- Parsing Dates:

```
string text = "11/11/2001";
DateTime parsedDate = DateTime.Parse(text);
```

Reversing a String

```
public class WordReverser {
        static void Main() {
              string text = "EM edit";
              string reversed = ReverseText(text);
              Console.WriteLine(reversed);
              // Console output:
              // tide ME
         }
         static string ReverseText(string text)  {
                    StringBuilder sb = new StringBuilder();
                    for (int i = text.Length - 1; i >= 0; i--)
                    {
                              sb.Append(text[i]);
                    }
                    return sb.ToString();
         }         }
```

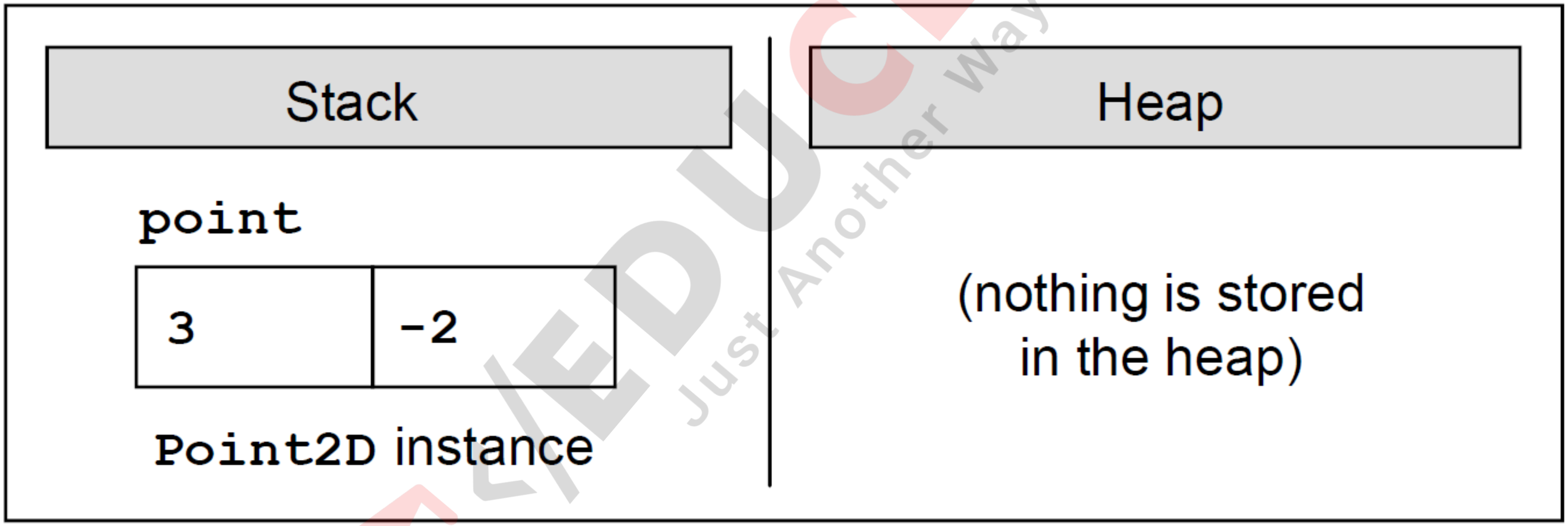
Structures

- Structures are defined through the keyword struct
- Structures (structs) are value types
- Use structures to hold simple data structures consisting of few fields that come together
- Examples are coordinates, sizes, locations, colors, etc

Structure (struct) – Example

```
struct Point2D {
     private double x;
    private double y;
    public Point2D(int x, int y) {
             this.x = x;
             this.y = y;
    }
    public double X {
         get { return this.x; }
         set { this.x = value; }
    }
   public double Y {
        get { return this.y; }
        set { this.y = value; }
   }
}
```

Sturct is a value type



Stack

point

3

-2

Point2D instance

Heap

(nothing is stored in the heap)

Enumerations

- Enumeration is a structure, which resembles a class but differs from it
- Enumerations can take values only from the constants listed in the type
- An enumerated variable cannot have value null
- Syntax:

```
[<modifiers>] enum <enum_name>
{
    constant1 [, constant2 [, [, … [, constantN]]
}
```

Enumeration - Example

```
enum Days
{
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
}
```

- Each constant, which is declared in one enumeration, is being associated with a certain integer
- Ex: int mondayValue = (int)Days.Mon;

  Console.WriteLine(mondayValue);