Chapter 2

Introduction to C#

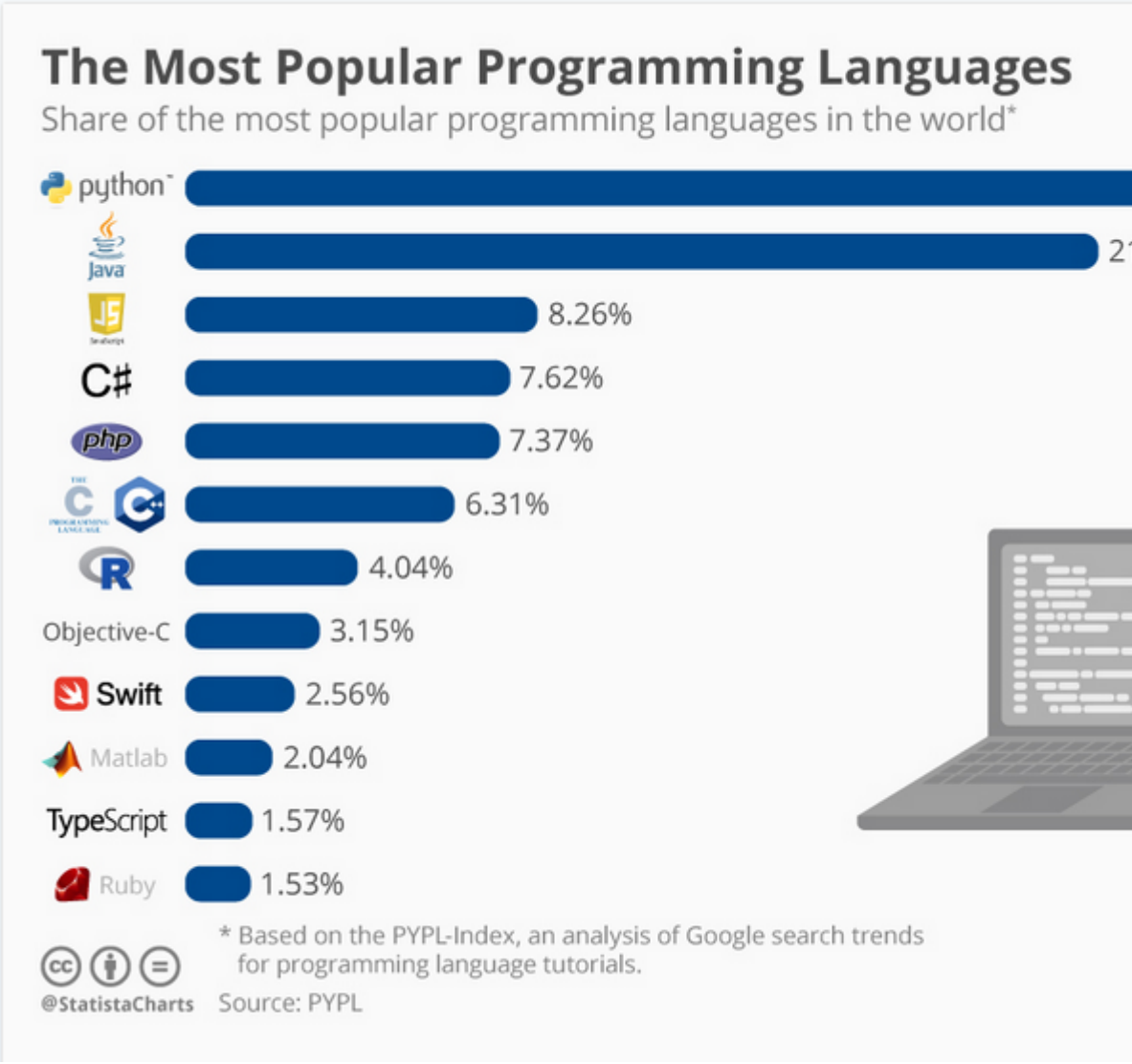
Mrs. Swati Satpute

TOP 10

Popular Programming Languages in 2020

1	Python
2	JavaScript
3	Java
4	C#
5	C
6	C++
7	GO
8	R
9	Swift
10	PHP

WWW.NORTHEASTERN.EDU/GRADUATE



C# Programming Language

- C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by **Anders Hejlsberg**.
- The first **component oriented language** in the C/C++ family
- Everything really is an Object
- Next generation robust and durable software
- Preservation of Investment

C# Programming Language

- C# is the first "**component oriented**" language in the C/C++ family

- Component concepts are first class:
  - Properties, methods, events
  - Design-time and run-time attributes
  - Integrated documentation using XML

- Enables one-stop programming
  - No header files, IDL, etc.
  - Can be embedded in web pages

C# Programming Language

- **Robust and durable software**
  - Garbage collection
    - No memory leaks and stray pointers
  - Exceptions
    - Error handling is not an afterthought
  - Type-safety
    - No uninitialized variables, unsafe casts
  - Versioning
    - Pervasive versioning considerations in all aspects of language design

Type of applications written in C#

- Winforms
  - Windows like Forms e.g. Microsoft Office
- Console - Command line Input and Output.
- Web Application
  - Web pages are written in ASP.NET but the backend code is C#.
- Web Services
- Windows Service

Hello World

```
using System;

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

- Entry point is Main() function.
- The using keyword refers to resources in the .NET Framework Class Library

C# Program Structure

- Namespaces
  - Contain types and other namespaces
- Type declarations
  - Classes, structs, interfaces, enums, and delegates
- Members
  - Constants, fields, methods, properties, indexers, events, operators, constructors, destructors
- Organization
  - No header files, code written “in-line”
  - No declaration order dependence

C# Program Structure

```
using System;

namespace System.Collections
{
   public class Stack
   {
      Entry top;

      public void Push(object data) {
         top = new Entry(top, data);
      }

      public object Pop() {
         if (top == null) throw new InvalidOperationException();
         object result = top.data;
         top = top.next;
         return result;
      }
   }
```

Namespace

- It **provides scope** for both preinstalled framework classes and custom developed classes

- To access the namespace contents "**using**" keyword is used.
- Namespace acts like **container of classes**
  - Used for storing Types and other namespaces
- Due to namespace boundary, **contents** of 2 namespaces can be **distinctly addressed**
- **Main goal of Namespace** is to **create hierarchical organization** of program
  - Where user doesn't have to worry about naming conflicts
- **System** is base namespace

Namespaces

- .Net's way of providing containers to application code i.e. code and its content

```
namespace LevelOne
{
    string nameOne;
}
```

- Qualified name contains all of its hierarchical information.
  e.g. LevelOne.nameOne
- One can define **nested namespaces**

Note:

*Only **using** is **not enough for referring namespace**, **SourceCode** of the name **has to be associated with the project.***

The Main( ) Method

- Every C# Console application must have main()
- Entry point of C# application.
- Return type of Main( ) could be void or int.

Basic C# Syntax

- C# is a **block-structured language.**
  - statements are part of a block of code. i.e. part of {}

```
{
   <code line 1, statement 1>;
   <code line 2, statement 2>;
   <code line 3, statement 3>;
}
```

- **Auto Indentation**
  - VS Editor will automatically indent the code

```
{
    if (x > y)
    {
       <code line 1, statement 1>;
      <code line 2, statement 2>
    }
}
```

  - In Visual Studio 2010 onwards
    - Ctrl +k +d indent the complete page.
    - Ctrl +k +f indent the selected Code.

Basic C# Syntax - Comments

- One liner

  // This is a one liner comment.

- Multi liner

  /* This is a comment */

- A special comment

  /// A special comment

  - Single liner comment
  - Ignored by compiler
  - Can configure VS to extract the text after these comments to prepare documentation

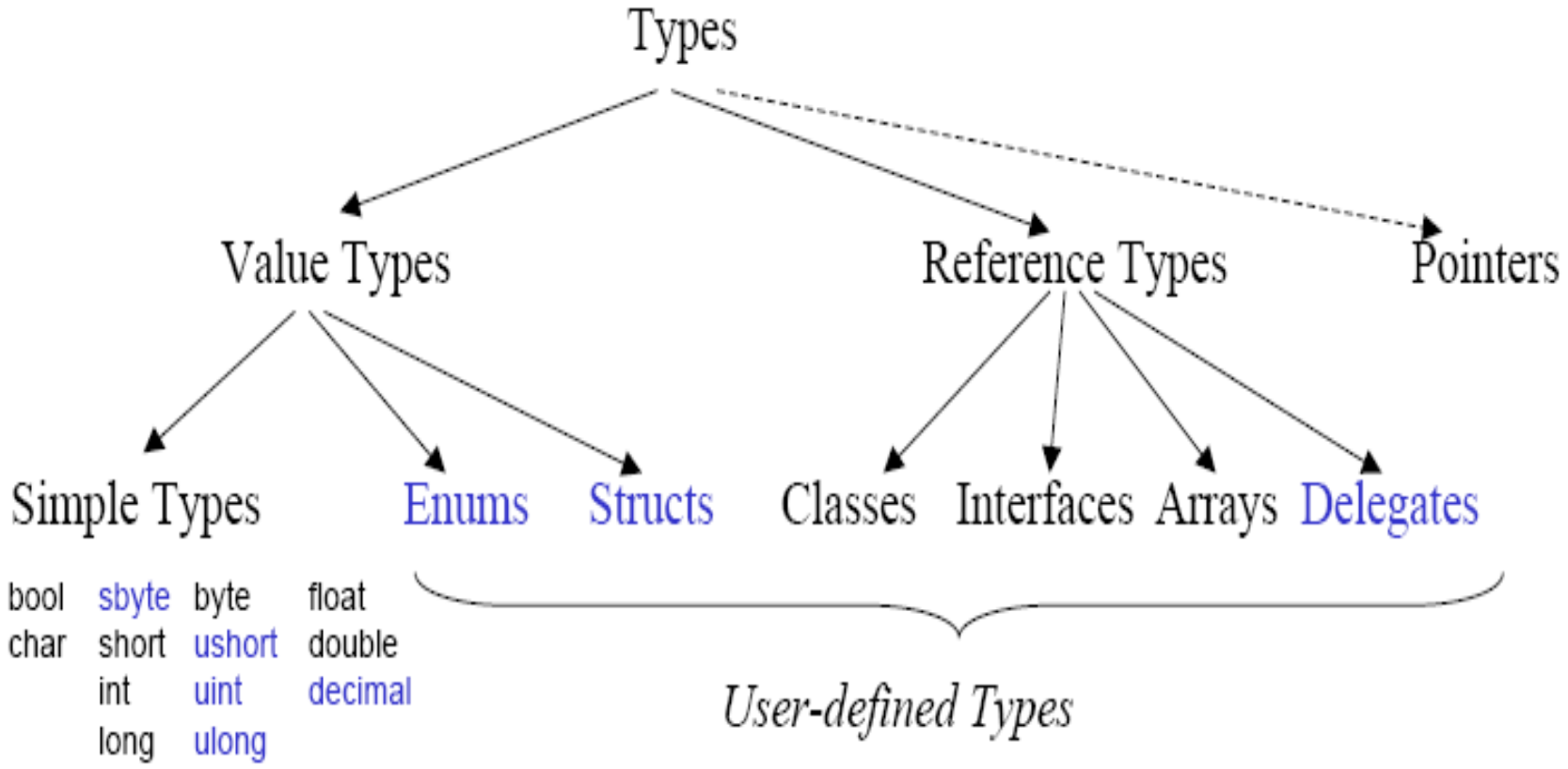
Basic C# Syntax - Region

- Regions used for code **outlining** / **indentation**
- One can use for **logical grouping** of code
- Allows user to **Expand** and **collapse** the code

e.g.

```
#region Call_Reflection
String strTypeName = "CarLibrary.Car";
            Assembly a = Assembly.LoadFrom(@"E:\CarLibrary.dll");
            Object myCar = a.CreateInstance(strTypeName);
            MethodInfo mi =
    a.GetType("CarLibrary.Car").GetMethod("StartCar");
            if (mi != null)
                mi.Invoke(myCar,null);
#endregion
```

Unified Type System

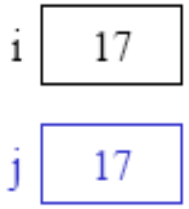


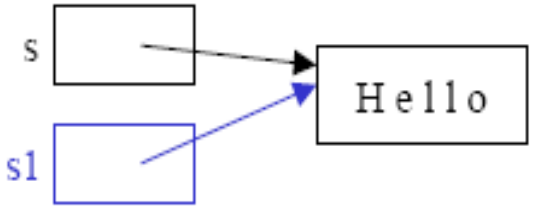
All types are compatible with *object*

- can be assigned to variables of type *object*
- all operations of type *object* are applicable to them

Value types Vs Reference Types

	Value Types	Reference Types
variable contains	value	reference
stored on	stack	heap
initialisation	0, false, '\0'	null
assignment	copies the value	copies the reference
example	int i = 17; int j = i;	string s = "Hello"; string s1 = s;

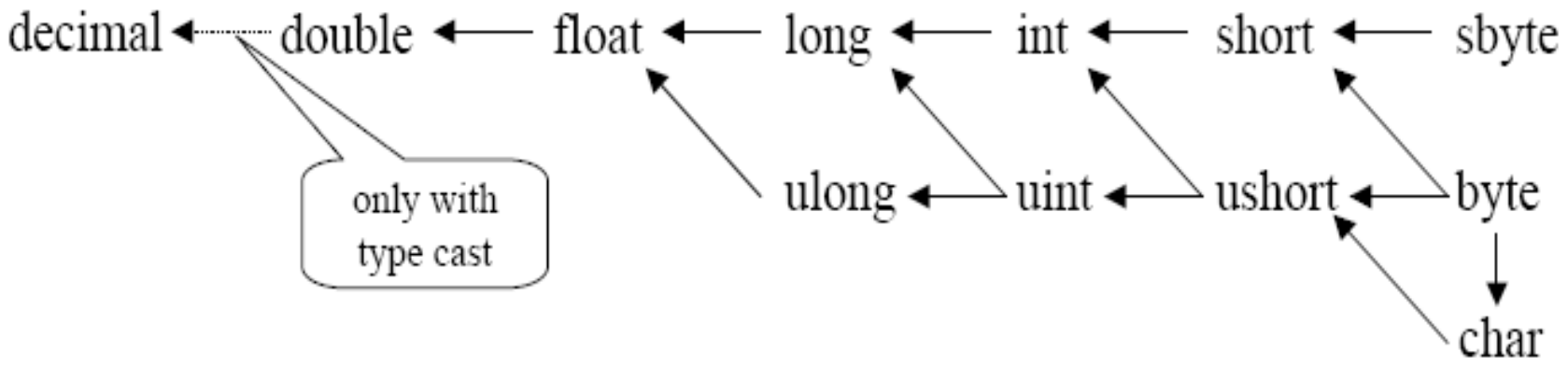




Simple Types

	Long Form	in Java	Range
sbyte	System.SByte	byte	-128 .. 127
byte	System.Byte	---	0 .. 255
short	System.Int16	short	-32768 .. 32767
ushort	System.UInt16	---	0 .. 65535
int	System.Int32	int	-2147483648 .. 2147483647
uint	System.UInt32	---	0 .. 4294967295
long	System.Int64	long	-2^{63} .. $2^{63}-1$
ulong	System.UInt64	---	0 .. $2^{64}-1$
float	System.Single	float	±1.5E-45 .. ±3.4E38 (32 Bit)
double	System.Double	double	±5E-324 .. ±1.7E308 (64 Bit)
decimal	System.Decimal	---	±1E-28 .. ±7.9E28 (128 Bit)
bool	System.Boolean	boolean	true, false
char	System.Char	char	Unicode character

Compatibility between simple Types



String vs String Builder

- A String is basically an **immutable sequence of characters**.
- Each character is a Unicode character in the range **U+0000 to U+FFFF**.
- Immutable means that its **state cannot be modified after it is created**.

```
String string1 = "Coding";
String string2 = "Sonata";
String string3 = "";
string3 = string1 + string2;
```

4 memory addresses to be allocated.
Coding, Sonata, **[EmptyString]** , and the **Concatenation** between Coding and Sonata

This will generate a big issue when in a loop with lots of iterations, where a string is adding (Concatenating) to itself another string or value, **it is just constantly reallocating.**
In other words, it is **copying to a new memory** location when the memory manager can't expand the requested amount in place.

String Builder

- The StringBuilder class **allows manipulation** of **a mutable string of characters.**
- It can be **used** when you want **to modify a string without creating a new object**, thus eliminating the overhead or bottleneck issue of the normal String concatenation.

- **Append**: Appends information to the end of the current StringBuilder.
- **AppendFormat**: Replaces a format specifier passed in a string with formatted text.
- **Insert**: Inserts a string or object into the specified index of the current StringBuilder.
- **Remove**: Removes a specified number of characters from the current StringBuilder.
- **Replace**: Replaces a specified character at a specified index.

When to use String and StringBuilder

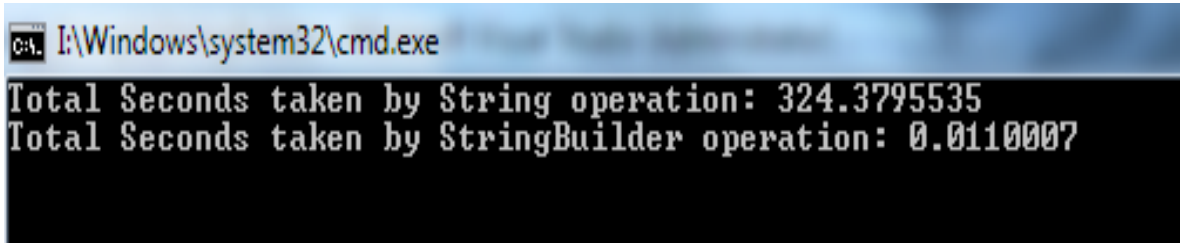
- Use **StringBuilder** when you're **concatenating** strings in a very long loop or in a loop within an unknown size – especially if you don't know for sure (at compile time) how many iterations you'll make through the loop.
  - For example, reading a file a character at a time, building up a string as you go.

- Use **String Concatenation operator** when you can specify everything which needs to be concatenated in one statement.
  - Use String.Concat explicitly – or String.Join
  - Avoid using the (+=) or the normal (+) for strings concatenation.

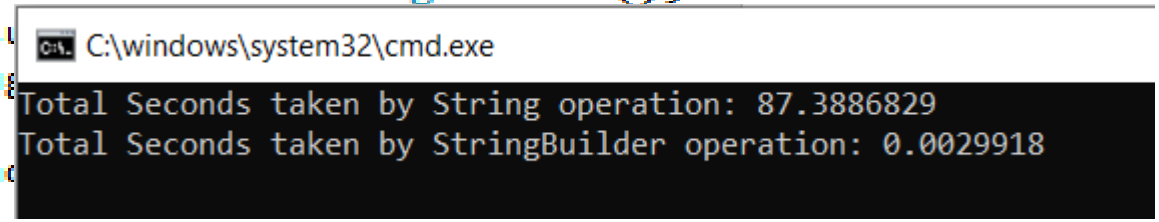
- In case **intermediate results of the concatenation** are needed for something other than feeding the next iteration of concatenation use String class

Performance using String class

```
List<String> hugeList = Enumerable.Range(1000, 200000).Select(n => n.ToString()).ToList();
String concatResult = "";
foreach (String value in hugeList)
{
    concatResult += value;
}
```




```
StringBuilder stringBuilder = new StringBuilder();
String stringBuilderResu
foreach (String s in hug
{
    stringBuilder.Append
}

stringBuilderResult = stringBuilder.ToString();
```



Lab Assignment

- Write a console based application, which performs following operations on data using StringBuilder.
  - **Append** data to existing string (i.e. OrgString) till user wants to continue.
  - Accept a sub string from user and **replace** it in original string if sub string is present.
  - **Count** occurrences of given sub string in OrgString.
  - Implement **remove** string functionality by accepting start position and number of characters from end user

Enumerations

List of named constants

Declaration (directly in a namespace)

```
enum Color {red, blue, green}    // values: 0, 1, 2
enum Access {personal=1, group=2, all=4}
enum Access1 : byte {personal=1, group=2, all=4}
```

Use

```
Color c = Color.blue;     // enumeration constants must be qualified

Access a = Access.personal | Access.group;
if ((Access.personal & a) != 0) Console.WriteLine("access granted");
```

Operations on Enumerations

Compare	if (c == Color.red) ... if (c > Color.red && c <= Color.green) ...
+, -	c = c + 2;
++, --	c++;
&	if ((c & Color.red) == 0) ...
|	c = c | Color.blue;
~	c = ~ Color.red;

The compiler does not check if the result is a valid enumeration value.

Note

- Enumerations cannot be assigned to *int* (except after a type cast).
- Enumeration types inherit from *object* (*Equals*, *ToString*, ...).
- Class *System.Enum* provides operations on enumerations
  (*GetName*, *Format*, *GetValues*, ...).

Arrays

One-dimensional Arrays

```
int[] a = new int[3];
int[] b = new int[] {3, 4, 5};
int[] c = {3, 4, 5};
SomeClass[] d = new SomeClass[10];   // Array of references
SomeStruct[] e = new SomeStruct[10];  // Array of values (directly in the array)

int len = a.Length;   // number of elements in a
```

Jagged Array

- A Jagged array is an array of arrays.

```
int [][] scores;
```

Declaring an array, does not create the array in memory. To create the above array: Make use of **New** operator

```
int[][] scores = new int[5][];
for (int i = 0; i < scores.Length; i++)
{
   scores[i] = new int[4];
}
```

You can initialize a jagged array as:

```
int[][] scores = new int[2][]{new int[]{92,93,94},new int[]{85,66,87,88}};
```

Where, scores is an array of two arrays of integers - scores[0] is an array of 3 integers and scores[1] is an array of 4 integers.

Rectangular Array

- C# allows multidimensional arrays. Multi-dimensional arrays are also called rectangular array.

```
string [,] names;
```

- Array can be thought of as a table, which has **X** number of rows and Y number of columns

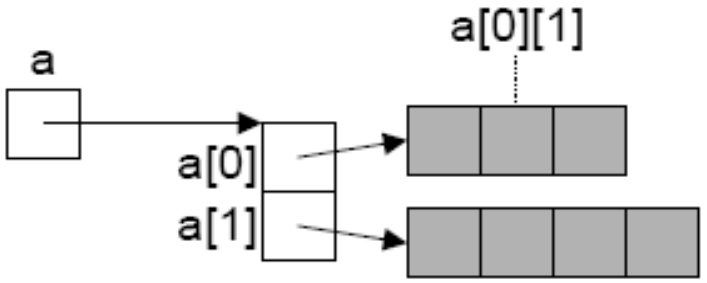
	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

```
int [,] a = new int [3,4] {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

Multidimensional Arrays

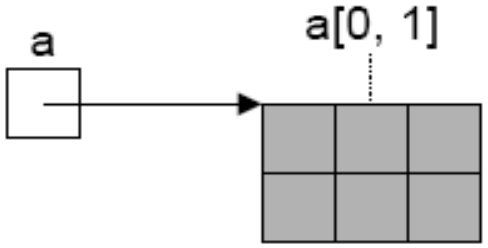
Jagged (like in Java)

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];

int x = a[0][1];
int len = a.Length;  // 2
len = a[0].Length;   // 3
```



Rectangular (more compact, more efficient access)

```
int[,] a = new int[2, 3];

int x = a[0, 1];
int len = a.Length;       // 6
len = a.GetLength(0); // 2
len = a.GetLength(1); // 3
```



Class System.String

Can be used as standard type *string*

```
string s = "Alfonso";
```

Note

- Strings are immutable (use *StringBuilder* if you want to modify strings)
- Can be concatenated with +: `"Don " + s`
- Can be indexed: `s[i]`
- String length: `s.Length`
- Strings are reference types => reference semantics in assignments
- but their values can be compared with == and != : `if (s == "Alfonso") ...`
- Class *String* defines many useful operations:
  CompareTo, IndexOf, StartsWith, Substring, ...

Structs

Declaration

```
struct Point {
    public int x, y;                                    // fields
    public Point (int x, int y) { this.x = x; this.y = y; }    // constructor
    public void MoveTo (int a, int b) { x = a; y = b; }        // methods
}
```

Use

```
Point p = new Point(3, 4);      // constructor initializes object on the stack
p.MoveTo(10, 20);               // method call
```

Features of C# Structures

- Structures can have **methods, fields, indexers, properties, operator methods, and events.**

- Structures can have **defined constructors**, but **not destructors**.
- Structures cannot define a default constructor. The default constructor is automatically defined and **cannot be changed**.

- Structures **cannot inherit** other structures or classes.

- A structure can **implement one or more interfaces**.

- Structure **members cannot be** specified as **abstract, virtual, or protected.**

- When you create a struct object using the **New** operator, it gets created and the appropriate constructor is called.

- Structs can be instantiated **without using the New operator**. But the fields remain unassigned and the object cannot be used until all the fields are initialized.

Structure Function

```
struct customerName
{
    public string firstname, lastname;

    public string Name()
    {
        return firstName + " " + lastName;
    }
}
```

- **Overloading of Functions is supported in C#**

Classes

Declaration

```
class Rectangle {
        Point origin;
        public int width, height;
        public Rectangle() { origin = new Point(0,0); width = height = 0; }
        public Rectangle (Point p, int w, int h) { origin = p; width = w; height = h; }
        public void MoveTo (Point p) { origin = p; }
}
```

Use

```
Rectangle r = new Rectangle(new Point(10, 20), 5, 5);
int area = r.width * r.height;
r.MoveTo(new Point(3, 3));
```

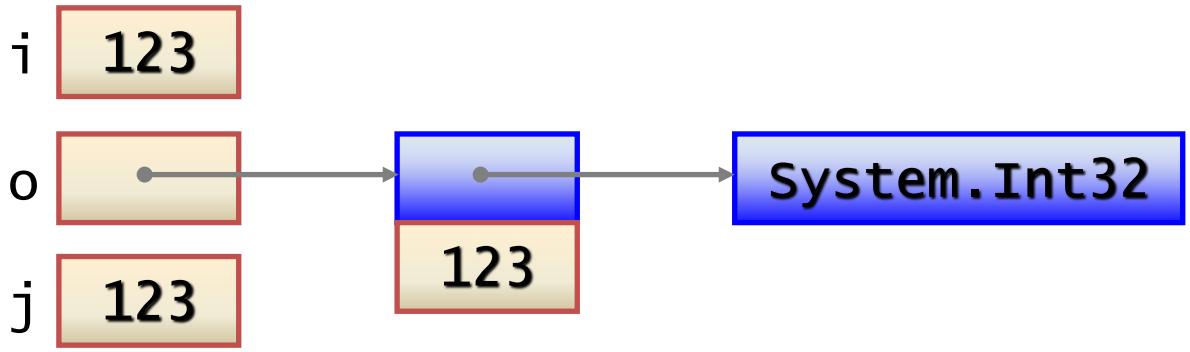
Difference between Classes and Structs

Classes	Structs
Reference Types (objects stored on the heap)	Value Types (objects stored on the stack)
support inheritance (all classes are derived from *object*)	no inheritance (but compatible with *object*)
can implement interfaces	can implement interfaces
may have a destructor	no destructors allowed
Programmer can specify default constructor i.e. parameterless	Programmer **cannot** specify default constructor i.e. parameterless

Boxing and Unboxing

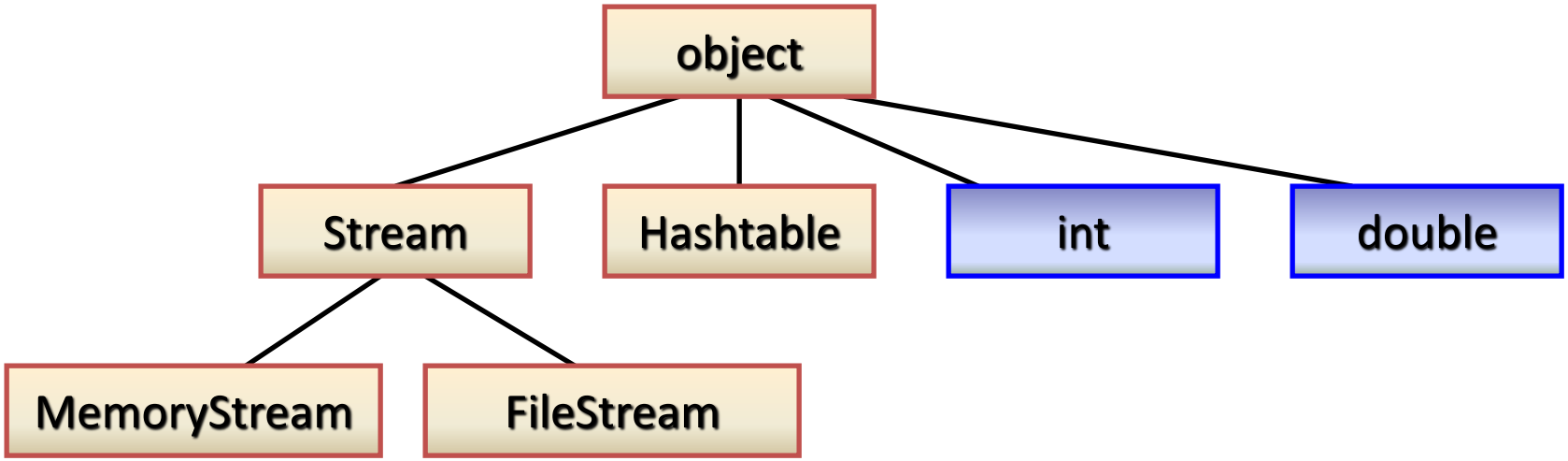
- Boxing
  - Allocates box, copies value into it
- Unboxing
  - Checks type of box, copies value out
  - Requires explicit conversion

```
int i = 123;
object o = i;
int j = (int)o;
```



System.Object Class

- **Object** class is root class of Framework class Library
- All types ultimately inherit from object
- **Any** piece of **data** can be **stored**, **transported**, and **manipulated** with **no extra work**



Methods on Object Class

- ***public bool Equals (object o)***
  - Determines whether two object instances are equal
- ***protected void Finalize ( )***
  - ***Allows an object to free up resources***
- ***public int GetHashCode ( )***
  - A hash code for the current object.
- ***public System.Type GetType ( )***
  - Gets the type of the current instance
- ***protected object MemberwiseClone ( )***
  - Creates a **shallow copy** of the current object
- ***public string ToString ( )***
  - Returns a String that represents the current object

Type Conversion

To convert values from one type to other type

Two forms of Type conversion

- **Implicit Conversion**
  - It occurs **automatically** and there is **no information loss**.
  - Where rules of conversion are simple enough .. User can rely on **compiler**

- **Explicit Conversion**
  - It **requires casting** and it **may not succeed.**
  - **Information** (precision) might be **lost.**

Implicit conversion

char to ushort conversion

```
char sourceVar = 'a';
ushort destVar;

destVar = sourceVar;

Console.WriteLine("SourceVar val: {0}", sourceVar );
Console.WriteLine("DestVar val: {0}", destVar );
```

```
SourceVar val: a
DestVar val: 97
```

- **char type represents** a character in Unicode char set **using a number**
- char - Number is stored the **same way as ushort**
- i.e. both of them store number between 0 – 65535

Note: Even though two types store the same information, they are **interpreted in diff way**, based on type

Implicit conversion

- **bool and string have no implicit conversions**
- Numeric types have few conversions as follows

Type	Can Safely Be Converted To
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, 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
long	float, double, decimal
ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Basic Rule:
Conversion of Type A to B is possible only when **all the possible ranges** of Type A **fits inside** the range of possible values of B

Explicit Conversions

- Occurs when explicitly asked compiler to convert

```
short sourceVar = 7;
byte destinationVar;

destinationVar = sourceVar;
Console.WriteLine("SourceVar val: {0}", sourceVar);
Console.WriteLine("DestinationVar val: {0}", destinationVar);
```

Conversion error will be flashed.
"are you missing a cast?"

- Casting is required to force data conversion

```
destinationVar = (byte)sourceVar;
```

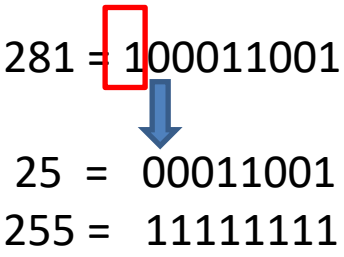
SourceVar val: 7
DestinationVar val: 7

Overflow i.e. Data Lost

```
short sourceVar = 281;
byte destinationVar;
destinationVar = (byte) sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

sourceVar val: **281**
destinationVar val: **25**

- Conversion is from **short** ➔ **byte**
- Maximum allowed value in Byte is 255



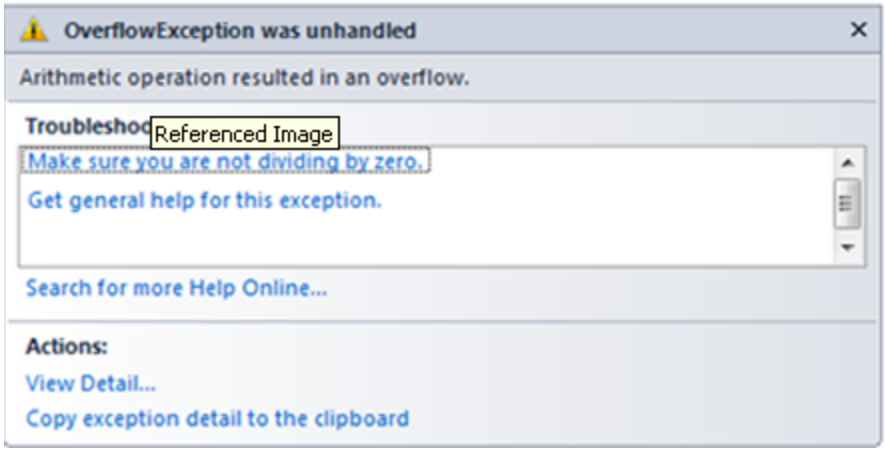
leftmost bit of the source data has been lost

Overflow will occur in case, when **value** to be stored is **bigger than Destination's capacity**

Overflow Checking

destinationVar = checked((byte)sourceVar);

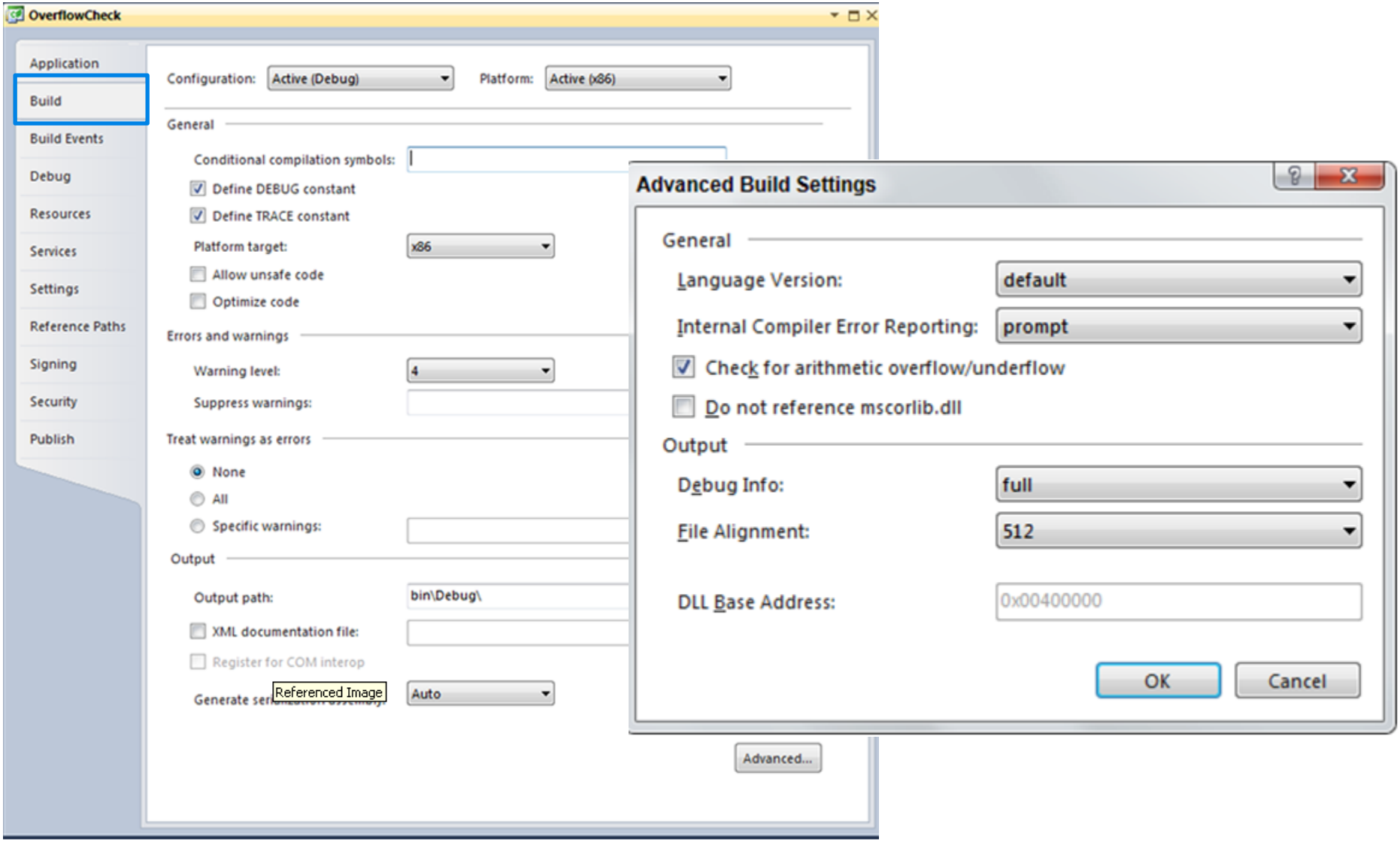
- check **ensures containers ability to store value**.
- In **case of overflow**, system will crash and show error



destinationVar = **unchecked**((byte)sourceVar);

- unchecked will ignore the overflow checks
- May result into invalid results
- **Default settings**

Application wide configuration



Explicit conversion using Convert

Command	Result
Convert.ToBoolean(val)	val converted to bool
Convert.ToByte(val)	val converted to byte
Convert.ToChar(val)	val converted to char
Convert.ToDecimal(val)	val converted to decimal
Convert.ToDouble(val)	val converted to double
Convert.ToInt16(val)	val converted to short
Convert.ToInt32(val)	val converted to int
Convert.ToInt64(val)	val converted to long
Convert.ToSByte(val)	val converted to sbyte
Convert.ToSingle(val)	val converted to float
Convert.ToString(val)	val converted to string
Convert.ToUInt16(val)	val converted to ushort
Convert.ToUInt32(val)	val converted to uint
Convert.ToUInt64(val)	val converted to ulong

Conversions are always **overflow - checked**

Conversion Assignment – Demo

```
short sh_Val = 4, sh_result = 0;
int int_val = 67;
long ln_Result;
float fl_Val = 10.5F;
double db_Result, db_Val = 99.999;

string str_Result, str_Val = "23";
bool bl_Val = true;
// float to double
db_Result = fl_Val * sh_Val;
Console.WriteLine("Implicit conversion: float= {0} short= {1} result_double=
{2}",fl_Val, sh_Val, db_Result);
 // float to short
sh_result = (short)fl_Val;
Console.WriteLine("Explicit conversion: float= {0} result short= {1}", fl_Val,
sh_result);
// boolean to string
str_Result = Convert.ToString(bl_Val);
Console.WriteLine("Convert bool {0} to String {1}: ",bl_Val, str_Result);
```

Operators & their precedence

Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
Unary	+ - ~ ! ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is as
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	c?x:y
Assignment	= += -= *= /= %= <<= >>= &= ^= |=

Operators on the same level are evaluated from left to right

Variable Naming

Basic rules:

1. The **first character** of a variable name **must be** either a **letter**, an **underscore** character (_), or the **at symbol** (@).

2. Subsequent characters may be letters, underscore characters, or numbers.

e.g. myCount

Naming convention

1. Hungarian Notation (popular)

   Type specific lowercase prefix on all variables e.g. i_cnt, str_temp

2. Conventions used in .net Framework namespaces
   a) PascalCase: e.g. Age, LastName
   b) camelCase: age, firstName

Flow Control

- Order of execution
  - Branching i.e. conditional execution

    e.g. count < 10
  - Looping i.e. repeated execution

- Goto statement

  Labels lines of code and jumps straight to them using goto statement

Flow Control - Branching

- Ternary operator
- IF statement
- Switch
  - Can check for **integer**, **constant** and **strings**

Switch Statement

```
switch (country) {
      case "Germany": case "Austria": case "Switzerland":
            language = "German";
            break;
      case "England": case "USA":
            language = "English";
            break;
      case null:
            Console.WriteLine("no country specified");
            break;
      default:
            Console.WriteLine("don't know language of {0}", country);
            break;
}
```

If no case label matches ➔ default

If no default specified ➔ continuation after the switch statement

Flow Control – Loops

while

```
while (i < n) {
        sum += i;
        i++;
}
```

do while

```
do {
        sum += a[i];
        i--;
} while (i > 0);
```

for

```
for (int i = 0; i < n; i++)
        sum += i;
```

Interrupting loops

break;

continue

goto

return

foreach statement

For iterating over collections and arrays

```
int[] a = {3, 17, 4, 8, 2, 29};
foreach (int x in a) sum += x;
```

```
string s = "Hello";
foreach (char ch in s) Console.WriteLine(ch);
```

```
Queue q = new Queue();
q.Enqueue("John"); q.Enqueue("Alice"); ...
foreach (string s in q) Console.WriteLine(s);
```

Functions

Syntax

<visibility> <return type> <name>(<parameters>)

{ <function code> }

e.g.

```
public void DoStuff(int i)
{
    Console.WriteLine(“Value received is: {0}“, i);
}
```

Static Methods

Operations on class data (static fields)

```
class Rectangle {
        static Color defaultColor;

        public static void ResetColor() {
                defaultColor = Color.white;
        }
}
```

Access within the class

```
ResetColor();
```

Access from other classes

```
Rectangle.ResetColor();
```

Parameters

Value Parameters (input values)

```
void Inc(int x) {x = x + 1;}
void f() {
    int val = 3;
    Inc(val); // val == 3
}
```

- "call by value"
- formal parameter is a copy of the actual parameter
- actual parameter is an expression

ref Parameters (transition values)

```
void Inc(ref int x) { x = x + 1; }
void f() {
    int val = 3;
    Inc(ref val); // val == 4
}
```

- "call by reference"
- formal parameter is an alias for the actual parameter
  (address of actual parameter is passed)
- actual parameter must be a variable

out Parameters (output values)

```
void Read (out int first, out int next) {
    first = Console.Read(); next = Console.Read();
}
void f() {
    int first, next;
    Read(out first, out next);
}
```

- similar to ref parameters
  but no value is passed by the caller.
- must not be used in the method before it got a value.

params Parameter

- params keyword can be used to specify a method parameter that **takes a variable number of arguments.**
- You can send a **comma-separated list of arguments of the type** specified in the parameter declaration or **an array of arguments** of the specified type.
- If **no arguments** are sent, the **length** of the params list is **zero**.
- **No additional parameters are permitted after the params keyword** in a method declaration
- **Only one params keyword** is permitted in a method declaration

params Parameter

```
public class MyClass
{
   public static void UseParams(params int[] list)
   {
      for (int i = 0; i < list.Length; i++)
      {
         Console.Write(list[i] + " ");
      }
      Console.WriteLine();
   }

   public static void UseParams2(params object[]
list)
   {
      for (int i = 0; i < list.Length; i++)
      {
         Console.Write(list[i] + " ");
      }
      Console.WriteLine();
   }
```

comma-separated list of arguments of the specified type.

```
UseParams(1, 2, 3, 4);
UseParams2(1, 'a', "test");
```

Zero Parameters

```
UseParams2();
```

Passing Array

```
int[] myIntArray = { 5, 6, 7, 8, 9 };
UseParams(myIntArray);

object[] myObjArray = { 2, 'b', "test", "again" }
UseParams2(myObjArray);
```

Assignment

Write a program **Swap** function using Ref parameter and Out parameter

Debugging

Two types of program execution mode

1. Debug
2. Release (Compact)

Debug mode

- Contains **symbolic information** about application
  - This helps VS to understand everything as each line executed
- Symbolic Information
  - Keeping track of uncompiled member, which matched up with values in compile code
- **.pdb** (**Program Data Base**) file is generated which stores all this info.

PDB

PDB enables user to perform following:

- Outputting debugging info to VS
- Looking and editing values at run time
- Pausing and restarting programs
- Automatically halting execution at certain points
- Stepping through code line by line
- Modifying variables at run time

Outputting Debug information

- Debugging can be performed by
  - Interrupting program execution
  - Generating notes for later analysis

- Output Window in VS IDE can be used to view debug notes

```
Debug.WriteLine("Inside Debug“)  // works only in debug mode
Trace.WriteLine("Inside Trace");    // works in both mode
```

- Use **System.Diagnostics** namespace

Other Commands

1. Debug.WriteLineIF();
2. Trace.WriteLineIF();
3. Debug.WriteIF();
4. Trace.WriteIF();

Error Handling

- An Exception is an **abnormal/exceptional/unexpected** condition that **disrupts the normal execution** of an application.

- An exception is an instance of a class which inherits from the System.Exception base class.

- Many different types of exception class are provided by the .NET Framework

- One can create own exception classes.

Error Handling

- Each type extends the basic functionality of the System.Exception class **by allowing further access to information about the specific type** of error that has occurred.

- **An instance of an Exception class** is created and **thrown** when the .NET Framework **encounters an error** condition.

- You can deal with exceptions by using the Try, Catch. Throw, Finally construct.

Error Handling

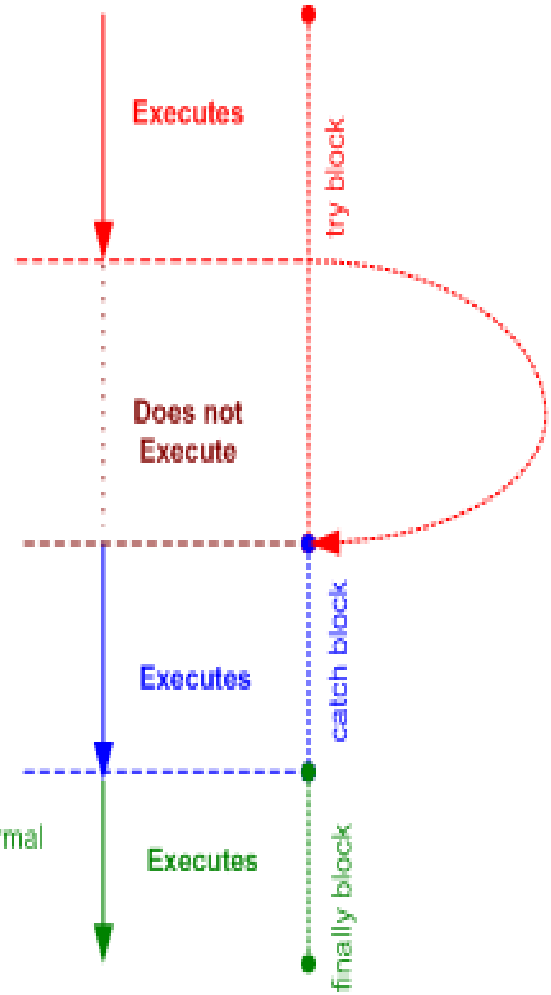
- The CLR infrastructure is also designed for **cross-language exception handling**

- What this means is :
  - that if a **VB.NET component throws** back **an exception** to a C# application that's consuming the component, the **C# application** will be able to **catch the error** and obtain rich error information on the error that has occurred.

Error Handling

Execution flow when an exception occurs

```
try
{
   // Code that can encounter errors and raise exceptions

   throw new  BankBalanceEmptyException();


   // Rest of the code goes here ……


}
catch
{
   // Code that handles errors and exceptions

}
finally
{
   // Code that performs cleanup and executes both on a normal
   // execution path as well as when an error occurs

}
```



Exceptions

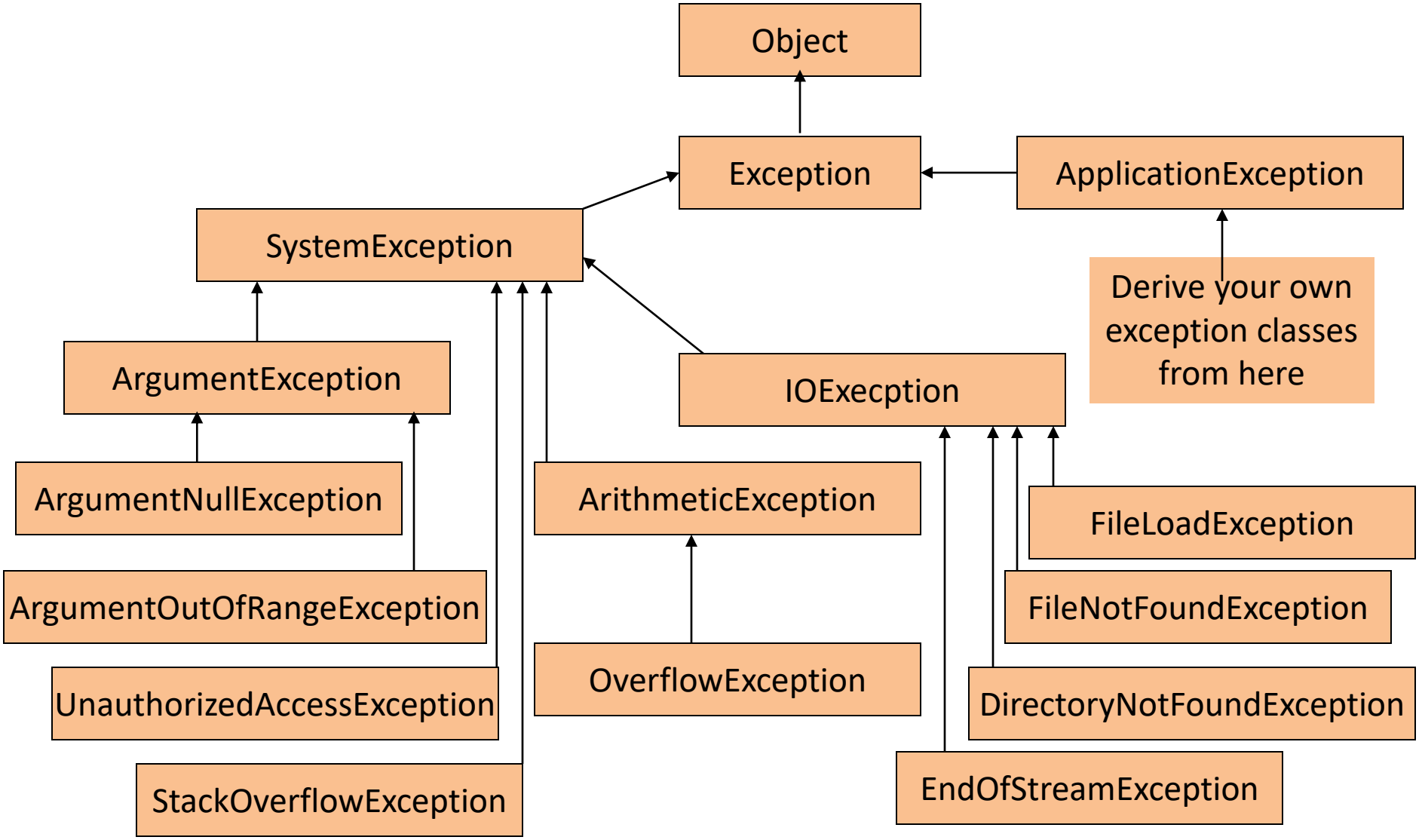
- Exception objects is created when particular exceptional condition occurs
- **Object contains exception details** which helps to solve the problem

Exception Categories

- **System.SystemException**
  - All exception classes are directly or indirectly derived from this class.
  - Contains information for the cause of the error / unusual situation
    - `Message` – text description of the exception
    - `StackTrace` – the snapshot of the stack at the moment of exception throwing.

- **System.ApplicationException**
  - This is intended base for implementing user defined exceptions

Exception Classes



Common Exception Types

- The following exception types are used widely throughout the CLR and .NET Framework. You can throw these yourself or use them as base classes for deriving custom exception types.

- System.ArgumentException

  Thrown when a function is called with a **bogus argument**. This generally indicates a program bug

- System.ArgumentNullException

  Subclass of ArgumentException that's thrown when a function **argument is (unexpectedly) null.**

- System.ArgumentOutOfRangeException

  Subclass of ArgumentException that's thrown when a (usually numeric) **argument is too big or too small**.

  e.g. this is thrown when passing a negative number into a function that accepts only positive values.

Common Exception Types

- System.InvalidOperationException

  Thrown when the **state of an object is unsuitable for a method** to successfully execute, regardless of any particular argument values.

  E.g. Reading an unopened file or getting the next element from an enumerator where the underlying list has been modified partway through the iteration.

- System.NotSupportedException

  Thrown to indicate that a particular functionality is not supported.

  e.g. Calling the **Add** method on a collection for which **IsReadOnly** returns true

- System.NotImplementedException

  Thrown to indicate that a function has not yet been implemented.

- System.ObjectDisposedException
- Thrown when the object upon which the function is called has been disposed.

Catching Exception

- *try* blocks: contains code of the normal operation of program,
  - but **which might encounter** some **serious error** conditions.

- *catch* blocks: contains code that **deals with the various error conditions** that your code **might have encountered** by working through any of the code in the accompanying *try* block

- *finally* blocks: contains code that **cleans up any resources** or **takes any other action** that you will normally want to done at the end of a *try* or *catch* block.

Catching Exception

```
try
{
    // Code for normal execution
}
catch(Exception e)
{
    // Error handling
}
finally
{
    // Clean up
}
```

Throwing Exception

When a method needs to notify the calling method than an error

has occurred, it uses the **throw keyword** in following manner :

```
throw [expression];
```

Creating Your Own Exception

```
public class myExceptionClass: Exception
{       public myExceptionClass():base()
        {       // Your Code            }
        public myExceptionClass(string message) : base(message)
        {       // Your Code            }
}
public class myClass
{       static void Main(string[] args)
        {       try{            ….
                                throw new myExceptionClass(“Error”);
                }
                catch(myExceptionClass e){
                                Console.WriteLine(e.Message);
                }
        }
}
```

Assignment Implement Invalid age range Exception

Object Oriented Programming

- Class:
  - blue print of object
  - Classes describe the ***type*** of object

- Object:
  - building block of an OOP
  - objects are usable instances of classes.

- Class Members

  a) **Properties**:
  describe class data

  b) **Method**:
  define class behavior.

  c) **Events**:
  provide communication between different classes and objects

Class Members

- Fields/Data Members
- Methods/Members Functions
- Properties
- Constructors
- Finalizers
- Operators
- Indexers
- Constants
- Events

Access Modifiers

- public
- private
- protected
- Internal
  - The **type or member** can be **accessed** by any code in the **same assembly**, but not from another assembly
- protected internal
  - Protected **OR** Internal
  - The type or member can be accessed
    - by any code **in** the **same assembly**
    - or from **within a derived class** in another assembly.

Constructors

- **Implicit method** gets invoked during object creation
- **Member initialization** is feature of constructor
- More than one constructor could be written
  - Default Constructor
  - Parameterized Constructor
  - Private Constructor
  - Static Constructor

Constructors for Classes

Example

```
class Rectangle {
    int x, y, width, height;
    public Rectangle (int x, int y, int w, int h) {this.x = x; this.y = y; width = x; height = h; }
    public Rectangle (int w, int h) : this(0, 0, w, h) {}
    public Rectangle () : this(0, 0, 0, 0) {}
    ...
}
```

```
Rectangle r1 = new Rectangle();
Rectangle r2 = new Rectangle(2, 5);
Rectangle r3 = new Rectangle(2, 2, 10, 5);
```

- Constructors can be overloaded.
- A construcor may call another constructor with *this*
  (specified in the constructor head, not in its body as in Java!).
- Before a construcor is called, fields are possibly initialized.

Constructor calling another constructor

```
public class RecursiveConstructor
{
   //When this constructor is called
   public RecursiveConstructor():this(One(), Two())
   {
       Console.WriteLine("Constructor one. Basic.");
   }

   public RecursiveConstructor(int i, int j)
   {
       Console.WriteLine("Constructor two.");
       Console.WriteLine("Total = " + (i+j));
   }

   public static int One()
   {
       return 1;
   }

   public static int Two()
   {
       return 2;
   }
}
```

The calling method

```
public class RecursiveConstructorTest
{
    public static void Main()
    {
        RecursiveConstructor recursiveConstructor = new RecursiveConstructor();

        Console.ReadKey();
    }
}
```

The Result

Constructor two.

Total = 3

Constructor one. Basic.

Default Constructor

If no constructor was declared in a class, the compiler generates a parameterless default constructor:

```
class C { int x; }
C c = new C();      // ok
```

The default constructor initializes all fields as follows:

numeric	0
enum	0
bool	false
char	'\0'
reference	null

If a constructor was declared, no default constructor is generated:

```
class C {
    int x;
    public C(int y) { x = y; }
}

C c1 = new C();    // compilation error
C c2 = new C(3);   // ok
```

Private Constructor

- Private constructor in c# is used to **restrict the class from being instantiated** when it contains **every member as static**.

- A private constructor **cannot be externally called.**

- If a **class has** one or more **private constructor and no public constructor** then other classes are not allowed to create instance of this class
  - **No object creation** of the class
  - **nor** it can be **inherit by other class**

Private Constructor

```
public class Counter
{
    private Counter() { }
    public static int currentCount;
    public static int IncrementCount()
    {
        return ++currentCount;
    }
}
class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter objCounter = new Counter();   // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

    }
}
// Output: New count: 101
```

Private Constructor – Singleton Pattern

```
public sealed class Test
{
   public static readonly Test Instance = new Test(); // Singleton pattern
   public int A; // Instance field
   private Test() // This is the private constructor
   {
      this.A = 5;
   }
}
 class Program
{
   static void Main()
   {  // We can access an instance of this object that was created. // ... The private constructor was used.
      Test obj_test = Test.Instance;

      // These statements show that the class is usable.
      Console.WriteLine(obj_test.A);
      obj_test.A++;
      Console.WriteLine(obj_test.A);
   }
}
```

Static Constructors

- It is used **to initialize any static data**
- To **perform a particular action** that needs to be **performed once only**.
- It is **called automatically before**
  - the **first instance is created** or **any static members are referenced**
- A static constructor **does not take access modifiers** or have parameters.
- It **cannot be called directly.**
- The **user has no control** on **when** the static constructor **is executed** in the program
- **If** a static constructor **throws an exception**, the **runtime will not invoke it a second time.**

Static Constructors

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Use Case:
Bus Department would like to record when the first bus for the day went. Next buses can have comparative time diff.

So static member can store a start time, which is initialized by static constructor. And never changed again

Static Constructors

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to {0}",
            globalStartTime.ToLongTimeString());
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }

    // Instance method.
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;
```

Constructors Demo

Constructors for Structs

Example

```
struct Complex {
        double re, im;
        public Complex(double re, double im) { this.re = re; this.im = im; }
        public Complex(double re) : this (re, 0) {}
        ...
}
```

```
Complex c0;                 // c0.re and c0.im are still uninitialized
Complex c1 = new Complex();     // c1.re == 0, c1.im == 0
Complex c2 = new Complex(5);    // c2.re == 5, c2.im == 0
Complex c3 = new Complex(10, 3);  // c3.re == 10, c3.im == 3
```

- For every struct the compiler generates a parameterless default constructor (even if there are other constructors).
  The default constructor zeroes all fields.

- Programmers must not declare a parameterless constructor for structs (for implementation reasons of the CLR).

Destructor

- Implicit method that gets invoked by CLR before object destruction.
- **Freeing up of resources** is done in destructor

```
class Car
   {
      ~Car()  // destructor
      {
         // cleanup statements...
      }
   }
```

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

The destructor implicitly calls Finalize on the base class of the object. Therefore, the previous destructor code is implicitly translated to the following code:

Destructor vs Dispose vs Finalize

Destructor:

- Destructor will be written in a class to clean the memory used by the instances of that class. **A destructor cannot be explicitly called in C#.** It will be **called by GC process** while collecting the garbage. Its more like C++ concept

- In C#, writing a destructor is the way to override the Finalize method. So there's really no difference between the two.

Dispose:

- Dispose method **Must be called explicitly** at any time just like any other method. Contains the code to **clean up the Unmanaged code** accessed by the object

Finalize

- Finalize Method is the **code to clean up the memory used by the class**. A finalize method can be called explicitly by using the "objectname.Finalize()" syntax

Fields / Data Members

- **Fields** represent information /**state that an object** contains.
- Fields are like variables , they can be read or set directly.
- Modifiers applied to fields are

static: (associated with the class as a whole)

ex. public static int count=2;

const: value is set at **compile time**
const members are **static**

ex. public const double pi = 3.1415;

readonly: Variable can be **initialized** at **declaration or in constructor.**

ex. public **readonly** string count;

Properties and Fields

- Properties have **get** and **set** procedures, which provide more **control on how values are set or returned**.

```
class SampleClass
{
    private int   _sampleCnt;
    public int SampleCount
    {
        get { return _sampleCnt; }
        set { _sampleCnt = value; } // // Store the value in the _sampleCnt
     }
}
```

Methods

- A ***method*** is **an action that an object can perform**.

```
class SampleClass
{
    public int sampleMethod (string sampleParam)
    {
        // Insert code here
    }
}
```

- Passing Parameters to Methods
  a) Value parameter
  b) ref parameters
  c) out parameters

- **Overloaded Methods**

```
public int sampleMethod (string sampleParam) {};
public int sampleMethod (int sampleParam) {}
```

Extension Methods

- Enables to Add methods to existing class **without creating new derived type.**
- Extension methods allow you to **inject additional methods without modifying, deriving or recompiling the original class**, **struct or interface**.
- Extension methods can be added to your own custom class, .NET framework classes, or third party classes or interfaces.
- Special kind of **static** method.
- They are **called as if** they were **instance methods** on the extended type

Extension Methods

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

```
using ExtensionMethods;

string s = "Hello Extension Methods";
int i = s.WordCount();
```

- Method is invoked using instance method syntax
- MSIL translates code into a call on the static method

Extension Methods

```
namespace ExtensionMethods
{
    public static class IntExtensions
    {
        public static bool IsGreaterThan(this int i, int value)
        {
            return i > value;
        }
    }
}
```

```
using ExtensionMethods;

class Program
{
    static void Main(string[] args)
    {
        int i = 10;

        bool result = i.IsGreaterThan(100);

        Console.WriteLine(result);
    }
}
```

Static Classes

- Static class contains
  - Contains only static members.
  - Cannot be instantiated.
  - Is sealed.
  - Cannot contain Instance Constructors.

e.g. Console class

Partial Classes

- A class can be spread across multiple source files using the keyword **partial**
- All source files for the class definition **are compiled as one file** with all class members
- **Access modifiers** used for defining a class **should be consistent** across all files

Object Oriented Programming Techniques

- Interfaces
- Inheritance
- Polymorphism
- Relationships between objects
- Operator Overloading
- Events
- Reference versus value types

Interfaces

- Interface is collection of **public** **methods, properties events and Indexers** that are grouped together to offer specific functionality.
- Interface looks like a class **but has no implementation**
- Class **implements** an interface.
- Use case: ImageProcessor

Interface

```
interface ISampleInterface
{
    void SampleMethod();
}
```

Class implementing Interface

```
class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    public void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        ImplementationClass obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

Interface implementing Properties

```
interface IPoint
    {
       // Property signatures:
    int x {
            get;
            set;
    }
    int y {
            get;
             set;
    }
}
```

```
class Point : IPoint
{
        private int _x,  int _y;
        public Point(int x, int y)
         {
                    _x = x;
                   _y = y;
        }

        // Property implementation:
        public int x
        {
                    get { return _x; }
                   set { _x = value; }
        }
        public int y
        {
                    get { return _y; }
                   set { _y = value; }
        }
}
```

Using Interface

```
class MainClass
{
     static void Main()
    {
        Point p = new Point(2, 3);

        Console.Write("My Point: ");
        Console.WriteLine("x={0}, y={1}", p.x, p.y);
     }
 }
```

```
static void PrintPoint(IPoint Ip)
 {
     Console.WriteLine("x={0}, y={1}", Ip.x, Ip.y);
 }

static void Main()
{
     PrintPoint(p);
 }
```

Plug n play architecture

- Interfaces are great for putting together plug-n-play like architectures where **components can be interchanged at will**

- Changing of objects (objects implementing same interface) is possible since both objects offer same functionality.

- It offers great flexibility without extra programming.

- e.g. Music player interface implemented by Sony and Panasonic classes.

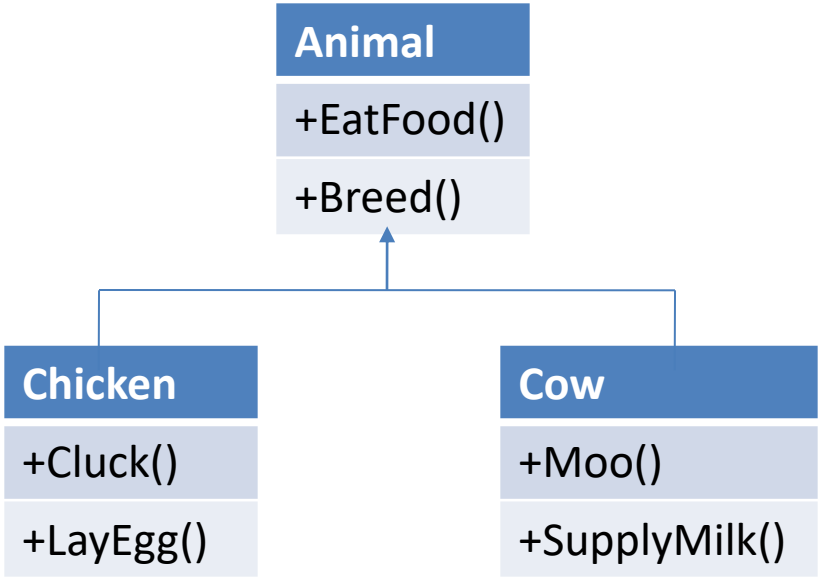
Interfaces

- A class can **support multiple interfaces**
- Multiple classes can support same interface
- Interfaces once published should not be changed
  - Effort on consumer for code changes
  - Recompilation of code is required

Inheritance

Inheritance

- Inheriting an existing class means new class can
  - **reuse**, **extend**, and **modify** the behavior of the existing class
- Inheritance represents a **IS-A relationship.**
- C# follows **single inheritance.**



Inheritance

Base class Member Accessibility

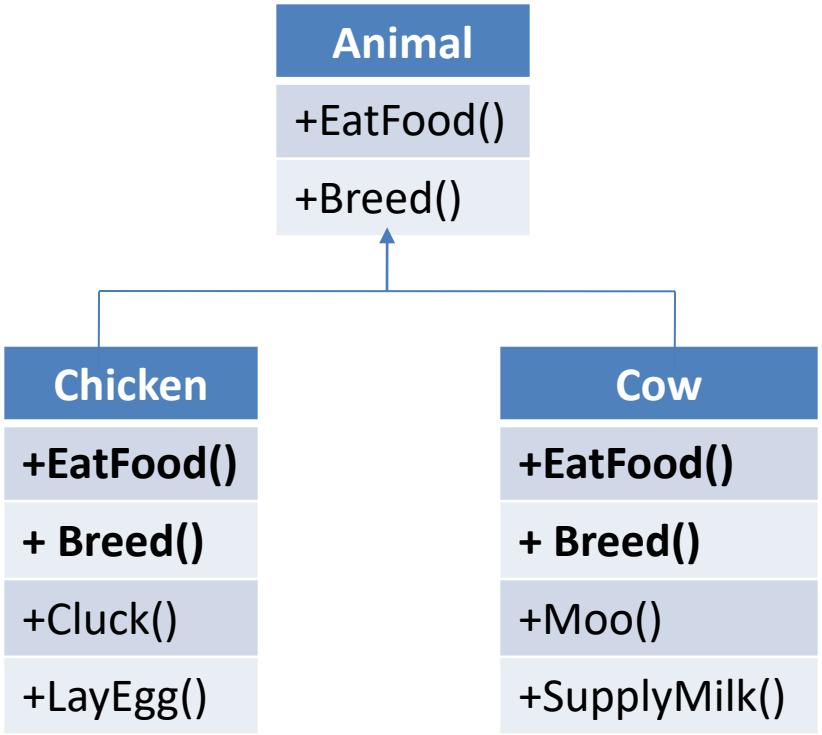
	Derived Class	External Code
Private member	X	X
Public member	√	√
Protected member	√	X

Abstract Class

- Abstract means **incomplete** or **missing implementation.**
- It can **only be declared**. This **forces the derived class to provide the implementation** of it.
- Derived class **MUST** provide implementation for them
- **Can not be instantiated**, one has to **Inherit them**.
- Abstract class may have **abstract members** (i.e. **no implementation**)

- An abstract member is **implicitly virtual**. Abstract can be called as **pure virtual** in some of the languages

Abstract Class



Virtual class member

- Virtual Function **has an implementation**.
- If derived class has to come up with own logic, it has to **override** the virtual function.
  - Note: Original implementation of member in base class remain untouched
- **Virtual** means we **CAN override** it.

Overriding Members

- By default, a derived class inherits all members from its base class
- To change the behavior of the inherited member, you need to **override**

C# Modifier	Definition
virtual (C# Reference)	Allows a class member to be overridden in a derived class.
override (C# Reference)	Overrides a virtual (overridable) member defined in the base class.
Not supported	Prevents a member from being overridden in an inheriting class.
abstract (C# Reference)	Requires that a class member to be overridden in the derived class.
new Modifier (C# Reference)	Hides a member inherited from a base class

Virtual and Override

```
class Employee
{    public virtual double calculatesalary()
     {       return basic_sal + hra + da;            }
}

class Manager : Employee
{    public override double calculatesalary()
     {       return basic_sal + hra + da + allowances;      }
}

class Program
{    static void Main(string[] args)
     {
          Employee objmgr = new Manager();
          double salary = objmgr.calculatesalary();
          Console.WriteLine(salary);
          Console.ReadLine();
     }
}
```

Virtual class member

```
public class Person
  {
     public virtual void ShowInfo()
     {
        Console.WriteLine("I am Person");
     }
  }

  public class Teacher : Person
  {
     public void ShowInfo()
     {
        Console.WriteLine("I am Teacher");
     }
  }
```

```
static void Main(string[] args)
{
        Person p_obj = new Person();
        p_obj.ShowInfo();

        Teacher t_obj = new Teacher();
        t_obj.ShowInfo();

        Person b_obj = new Teacher();
        b_obj.ShowInfo()
}
```

I am Person
I am Teacher
I am Person

- **Virtual and override concept come into picture only when Inherited functions are called using base class object**
- Base class reference is **unaware of function re defined** in derived class.

Virtual and Override

```
public class Person
  {
     public virtual void ShowInfo()
     {
        Console.WriteLine("I am Person");
     }
  }

  public class Teacher : Person
  {
     public override void ShowInfo()
     {
        Console.WriteLine("I am Teacher");
     }
  }
```

```
static void Main(string[] args)
{
        Person p_obj = new Person();
        p_obj.ShowInfo();

        Teacher t_obj = new Teacher();
        t_obj.ShowInfo();

        Person b_obj = new Teacher();
        b_obj.ShowInfo()
}
```

I am Person
I am Teacher
I am Teacher

- **Override** keyword ensures that **Base class reference will refer overridden function** i.e. re defined in derived class.

Inheritance

- ***Sealed*** class can not be derived
- All classes in C# implicitly inherit from ***Object*** class

- Interfaces may also **inherit from other interfaces**
- Interfaces may have **multiple base interfaces**

Polymorphism in C#.NET

- Through inheritance, **a class can be used as more than one type**;
- it can be used as its **own type**, any **base type**s, or any **interface type** if it implements interfaces. This is called **polymorphism**.
- With polymorphism, the same method or property can **perform different actions depending on the run-time type of the instance** that invokes it
- **Overloading** and **overriding** are used to implement polymorphism

Polymorphism classification

a) Compile Time

- Compiler identifies which **polymorphic form** to execute at **compile time**

- Since early binding, **faster execution**

- Examples of early binding are **overloaded methods**, **overloaded operators** that are called directly by **using derived objects**.

Compile Time Polymorphism

```
class Class1
    {
        public int Sum(int A, int B)
        {
            return A + B;
        }

        public float Sum(int A, float B)
        {
            return A + B;
        }
    }
class Class2 : Class1
    {
        public int Sum(int A, int B, int C)
        {
            return A + B + C;

        }
    }
```

```
class MainClass
  {
     static void Main()
     {
        Class2 obj = new Class2();

        Console.WriteLine(obj.Sum(10, 20));

        Console.WriteLine(obj.Sum(10, 15.70f));

        Console.WriteLine(obj.Sum(10, 20, 30));
     }
  }
}
```

- Compiler identifies which overloaded method to execute based on number of arguments and their data types during compilation itself.
- **Method overloading** is an example for **compile time polymorphism**

Polymorphism classification

b) **Run Time**

- Compiler identifies which **polymorphic form** to execute **at runtime**

- Late binding **gives flexibility** but **slow in execution**

- Example of late binding is **overridden methods** that are called **using base class reference**

Run Time Polymorphism

```
Class Base
    Public  virtual Property X() As Integer
        Get
        Set
    End Property
End Class

Class Derived
    Public override Property X() As
      Integer
        Get
        Set
End Property
End Class
```

```
Module Test
    Sub F()
        Dim Z As Base

        Z = New Base()
        Z.X = 10     ' Calls Base.X

        Z = New Derived()
        Z.X = 10       ' Calls Derived.X
    End Sub
End Module
```

Interface Polymorphism

- Polymorphism is **all about taking any form in the hierarchy**
- Though interface can not be instantiated, variable **(Interface) can be used to invoke methods on object** (which implements that interface)

```
interface IConsume
{
    void EatFood();
}

class Cow: IConsume, Animal
{
 public void EatFood()
 { … }
}

class Chicken: IConsume, Animal
{
 public void EatFood()
 { … }
}
```

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
IConsume ConsumeInterface;

ConsumeInterface = myCow;
ConsumeInterface.EatFood();

ConsumeInterface = myChicken;
ConsumeInterface.EatFood();
```

Simplest way of calling same method on multiple objects

Overriding

- In polymorphism, When a virtual method is called on a reference, **the actual type of the object that the reference refers to** is used to decide which method implementation to use.

- ***override modifier*** may be used on virtual methods and must be used on abstract methods.

- This indicates for the compiler to use the **last defined implementation** of a method.

Override Example

```
public class MyBaseClass
   {
      virtual public  void DoSomething()
      {
         Console.WriteLine("Base Imp");
      }
   }
   public class MyDerivedClass : MyBaseClass
   {
       public override void DoSomething()
      {
         Console.WriteLine("Derived Imp");
      }
}
```

Hiding base class methods

- ***New modifier*** is used to hide an inherited member **from a base class member**

- **New indicates** an **altogether new implementation**

- Base class member getting hidden can be any of the following
  - A constant, field, property, or type introduced
  - A Method
  - An indexer

New Example

```
public class MyBaseClass
   {
        public void DoSomething()
       {
           Console.WriteLine("Base Imp");
       }
   }
   public class MyDerivedClass : MyBaseClass
   {
        new public void DoSomething()
       {
           Console.WriteLine("Derived Imp");
       }
}
```

New Vs Override

```
public class A
    {
            public virtual void One();
            public void Two();
    }

public class B : A
    {
            public override void One();
            public new void Two();
    }

B b = new B();
A a = b as A;
```

b.One(); Calls implementation in **B**

b.Two(); Calls implementation in **B**

a.One(); Calls implementation in **B**

Since Virtual n Overide is used**, latest implementation will be used**

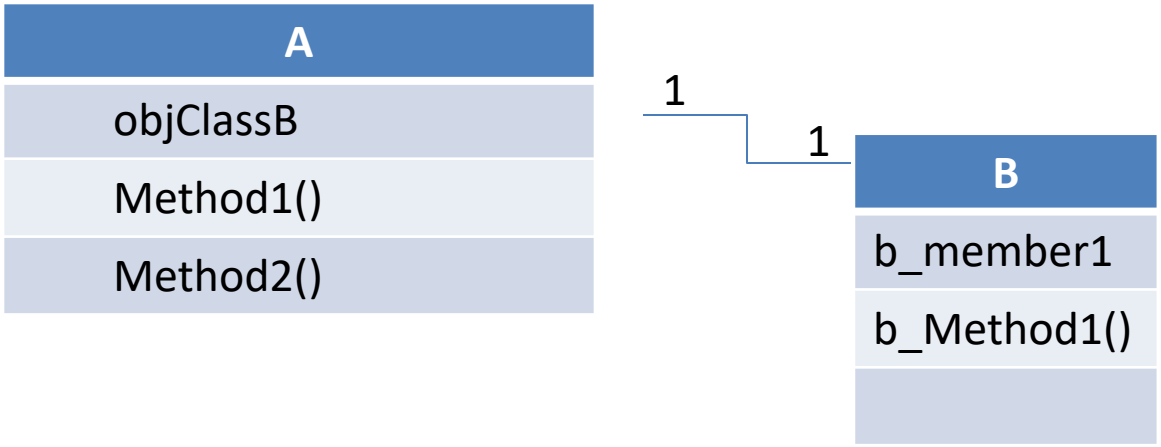
a.Two(); Calls implementation in **A**

Relationship between Objects

- Containment
  - One class contains Other class
  - Containing class controls access to members of contained class

- Collections
  - In this, one class contains **multiple instances of other class**

Containment



Access to Member object can be

- **Public**:

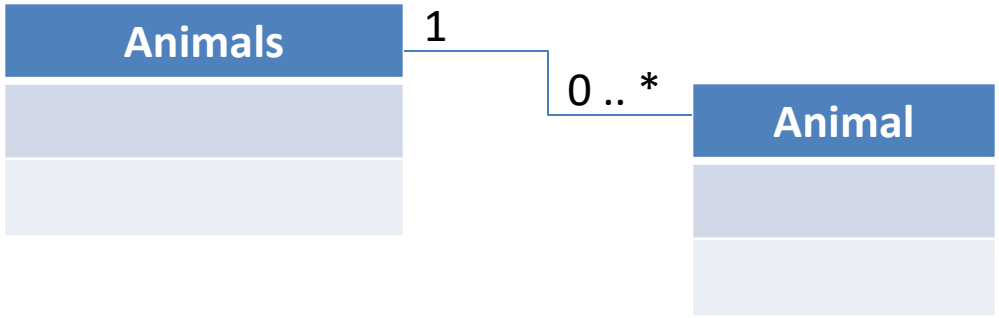
  Every one can access it

- **Private:**

  Access to contained objects can be restricted and can be made available in controlled manner

Note: Irrespective of how the member of Class B are defined i.e. public or private, Containing object can control the access by defining access specifier

Collections



- Collection is object, containing multiple instance of the another object (same type)

- Supports **Add**() and **Remove**() operation

- **Item** property returns object **based on Index** specified.

Collections, Comparisons and Conversions

Collections, Comparisons and Conversions

Advanced concepts in C#

- Collections
  - Collection enables to maintain group of objects
  - Collection can perform
    - Controlling access to items
    - Searching
    - Sorting
- Comparison
  - For comparing objects (e.g. collection **sorting**)
  - Comparing objects using **operator overloading**
  - Using IComparable and IComparer interface
- Conversion
  - Customizing Type conversions

Array

- **Arrays** are used for storing **objects of similar data types.**
- Can access Array elements by its **numeric index.**
- Indexes start at zero.

Using Array

Animal (Abstract)

::Feed()

- Cow
- Chicken

:: LayEgg()

```
Animal []  animalsArray = new Animal[2] ;
Cow myCow =  new Cow();

// Adding contents
animalsArray[0] = myCow;
animalsArray[1] = new Chicken();

ForEach (Animal myAnimal in animalsArray)
….
//Invoking Methods
animalsArry[0].Feed();
(Chicken)animalsArray[1].LayEgg();
```

Limitations of Array

- Array size is fixed. Once the array is created we **can never increase the size of an array**.
- We can **never insert an element** into the **middle** of an array.
- **Deleting or removing elements from the middle** of the array.

Collections

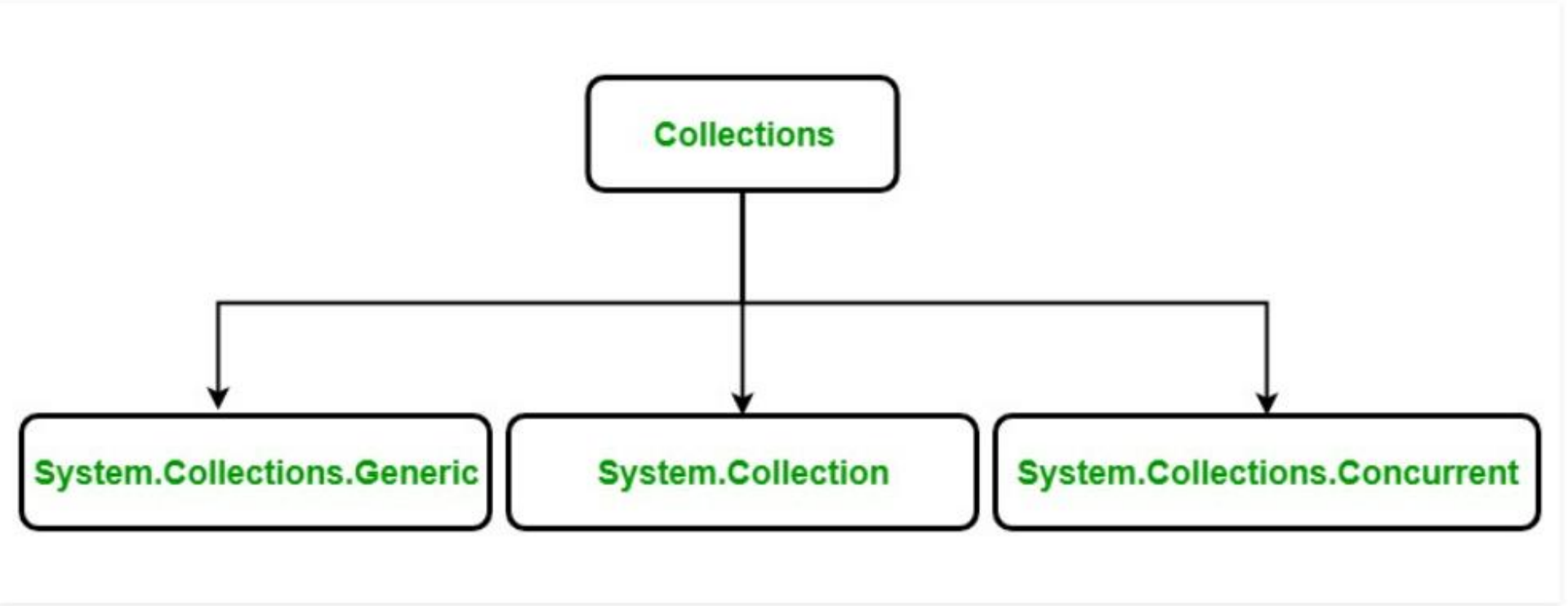
- Collections provide a more flexible way to work with groups of objects

- The group of objects can **grow and shrink dynamically**
- **One can insert, delete element from the middle of collection.**

- The .NET framework provides **specialized classes** for data storage and retrieval. They are efficient and performant

- There are two distinct collection types in C#.
  - System.Collections
  - System.Collections.Generic

Collections



Collections

- Collection classes offer more functionality by implementing interfaces from System.Collections
- Implementing customized collection classes is encouraged as
  - These classes **can be more specific to the objects**, which should be enumerated
  - Custom collection classes are **strongly types**.
  - **Direct member access or method invocation possible**
  - One can **expose specialized methods** (as per requirement)

    e.g. subset from Cards collection

Interfaces in System.Collections

- IEnumerable
  - Provides the ability to **loop through items** in collection
- ICollection
  - Provides the ability to **obtain Item count**, also **copying items in Array**
  - Inherits from IEnumerable
- IList
  - Provides **list of items** for collection
  - Ability to **access the item**
  - Inherits from IEnumerable and ICollection

- IDictionary
  - Similar to IList
  - Items are **accessed through Key** instead of Index
  - Inherits from IEnumerable and ICollection

System.Collections Classes

Class name	Description
ArrayList	It is a dynamic array means the size of the array is not fixed, it can increase and decrease at runtime.
Hashtable	It represents a collection of key-and-value pairs that are organized based on the hash code of the key.
Queue	It represents a first-in, first out collection of objects. It is used when you need a first-in, first-out access of items.
Stack	It is a linear data structure. It follows LIFO(Last In, First Out) pattern for Input/output.

ArrayLists

- ArrayList is a collection from a standard **System.Collections** namespace
- It is a **dynamic array**.
- It **provides random access** to its elements.
- An ArrayList **automatically expands** as data is added.
- Unlike arrays, an ArrayList **can hold data of multiple data types.**
- Array elements can be accessed via **integer Index**
- **Indexing** of elements and **insertion** and **deletion at the end** of the ArrayList **takes constant time**.
- **Inserting** or **deleting** an element **in the middle** of the dynamic array is more costly.

ArrayList

- **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

ArrayLists Example

```
static void Main()
   {
      ArrayList da = new ArrayList();

      da.Add("Visual Basic");
      da.Add(344);
      da.Add(55);
      da.Add(new Empty());
      da.Remove(55);

      foreach(object el in da)
      {
         Console.WriteLine(el);
      }
   }
```

Using ArrayList

```
ArrayList animalsArrayList = new ArrayList();

Cow myCow = new Cow();
animalsArrayList.Add(MyCow);
animalsArrayList.Add(new Chicken());

ForEach (Animal myAnimal in animalsArrayList)
….
animalsArrayList.RemoveAt(0);
animalsArrayList.AddRange(animalsArray);
 int index = animalsArrayList.IndexOf(myCow)

(Cow)animalsArrayList[1].Feed();
(Chicken)animalsArrayList[1].LayEgg();
Int ArrayLength = animalsArrayList.Count;
```

- ArrayList supports **Variable length data**
- ArrayList implements **IList**, **IEnumerable** and **ICollection**

How to get Subset of Cows or Chicken only???

Generic Collections

- The generic collections are more flexible and are the preferred way to work with data.
- A generic collection is useful when **every item** in the collection has the **same data type.**
- The generic collections or generics were introduced in .NET framework 2.0.
- Generics enhance code reuse, type safety, and performance.

Generic Collections

Class name	Description
Dictionary<TKey,TValue>	It stores key/value pairs and provides functionality similar to that found in the non-generic Hashtable class.
List<T>	It is a dynamic array that provides functionality similar to that found in the non-generic ArrayList class.
Queue<T>	A first-in, first-out list and provides functionality similar to that found in the non-generic Queue class.
SortedList<TKey,TValue>	It is a sorted list of key/value pairs and provides functionality similar to that found in the non-generic SortedList class.
Stack<T>	It is a first-in, last-out list and provides functionality similar to that found in the non-generic Stack class.
HashSet<T>	It is an unordered collection of the unique elements. It prevent duplicates from being inserted in the collection.
LinkedList<T>	It allows fast inserting and removing of elements. It implements a classic linked list.

List

- List is a **strongly typed list** of objects that can be accessed by index

- Found under **System.Collections.Generic** namespace.

```
static void Main()
{
    List<string> langs = new List<string>();

    langs.Add("Java");
    langs.Add("C#");
    langs.Add("C");
    langs.Add("C++");
    langs.Add("Ruby");
    langs.Add("Javascript");

    Console.WriteLine(langs.Contains("C#"));

    Console.WriteLine(langs[1]);
    Console.WriteLine(langs[2]);

    langs.Remove("C#");
    langs.Remove("C");

    Console.WriteLine(langs.Contains("C#"));

    langs.Insert(4, "Haskell");

    langs.Sort();

    foreach(string lang in langs)
    {
        Console.WriteLine(lang);
    }
}
```

Dictionary

- A **dictionary**, also called an associative array.
- Found under **System.Collections.Generic** namespace.

- It is a collection of unique keys and a collection of values,
  - where each key is associated with one value.

- **Retrieving and adding values is very fast**.

- Dictionaries **take more memory**, because for each value there is also a key.

Dictionary Example

```
Dictionary <string, string> domains = new Dictionary<string, string>();
 domains.Add("de", "Germany");
domains.Add("sk", "Slovakia");
domains.Add("us", "United States");
domains.Add("ru", "Russia");
domains.Add("hu", "Hungary");
domains.Add("pl", "Poland");

Console.WriteLine(domains["sk"]);
Console.WriteLine(domains["de"]);

Console.WriteLine("Dictionary has {0} items", domains.Count);
```

Collections Demo

- D:\Work\DOT NET2\Project\CollectionsDemo

Implementing Custom Collection

- **CollectionBase** Class exposes interface IEnumerable, IList and ICollection

- CollectionBase class provides implementation **only** for
  - IList ::Clear()
  - IList ::RemoveAt()
  - ICollection ::Count()

- CollectionBase provide **access to the stored object** through protected properties
  - **List** : which **gives access to items** through **IList** interface
  - **InnerList**: an **ArrayList** object used for **storing items**

- Inherit custom class from **CollectionBase** class

Custom Collection Example

```
Public class Animals : CollectionBase
{
    public void Add(Animal newAnimal)
    {
       List.Add(newAnimal)
    }
     public void Remove(Animal oldAnimal)
    {
      List.Remove(oldAnimal)
    }
}
Animals animalsCollection = new Animals();
animalsCollection.Add(new Cow());

Foreach (Animal myAnimal in animalsCollection)
     myAnimal.Feed();
```

Indexer

- Indexer is a **special kind of property** which provides **array like access**
- Custom Collection class **requires Indexer implementation**

```
public class Animals : CollectionBase
{
    public Animal this [int IndexVal]
    {
    get{
            return (Animal)List[IndexVal];
        }
    set{
            List[IndexVal] = value
       }
    }
}
```

```
animalsCollection[0].Feed();
```

- **this** keyword along with **[]** indicate **indexers signature**
- indexer code internally calls an indexer on the List property
- Explicit casting is necessary as the return is **system.object**

Keyed Collections

- Indexing of items is Key based.. User friendly
- Custom Class should implement IDictionary interface
- **DictionaryBase** class, implements **IEnumerable**, **ICollection** and **IDictionary** interfaces

Keyed Collection - Example

```
Public class Animals : DictionaryBase
{
    public void Add(string  Animal_KeyID, Animal newAnimal)
    {       Dictionary.Add(Animal_KeyID, newAnimal)
    }
     public void Remove(string Animal_KeyID)
    {       Dictionary.Remove(Animal_KeyID)
    }

    public Animal this [string Animal_KeyID]
    {
    get{
            return (Animal)Dictionary[Animal_KeyID];
        }
    set{
            Dictionary[Animal_KeyID] = value
       }
    }
}
```

Iterators

- **ForEach** relies on Enumerator pattern
  - GetEnumerator() method

- When ForEach gets executed on a collection, internally
  - CollectionObject.**GetEnumerator**() is called. **It returns IEnumerator reference**
  - IEnumerator ::**MoveNex**t method of the returned IEnumerator interface is called
  - If MoveNext is successful then IEnumerator ::**Current** property gets reference to an object
  - Loop is iterated until MoveNext returns false

IEnumerator

- **Collections are supposed to implement IEnumerable interface**

- Implementing the **IEnumerable** interface provides automatic **support for the foreach loop**

```
interface IEnumerable
{
     IEnumerator GetEnumerator();
    // Returns Enumerator object
}
```

- Enumerator object should implement **IEnumerator** interface

```
interface IEnumerator
{
    Object Current {get;}
    bool MoveNext();
    void Reset()
}
```

Code Sample

```
Foreach (object obj in list)
{
        DoSomething(Obj)
}
```

```
Enumerator e = list.GetEnumerator();
while (e.MoveNext()!= null)
 {
   object obj = e.Current;
   DoSomething(obj);
}
```

foreach makes enumerating easy

But enumerators are hard to write!

IComparable

- IComparable exposes single method **CompareTo()**

- CompareTo()
  - Single input parameter is an object of type **System.Object**
  - **Returns int.**

    Return value can be used to inform **magnitude between comparison**. E.g. Person1 is 5 yrs elder to Person2

```
If (Person1.CompareTo(Person2) == 0)
    Console.WriteLine("Same age")
```

IComparer

- IComparer exposes single method **Compare()**

- Compare()
  - Input parameter are **two objects of type System.Object**
  - returns a **value** indicating whether one is less than, equal to, or greater than the other.

```
If (PersonComparer.Compare(Person1, Person2) == 0)
    Console.WriteLine("Same age")
```

- Since any two objects can be compared using this method, its preferred to have basic type checking before returning comparison result

Comparer class

- **Comparer** Standard .NET class which **implements IComparer** interface
- Performs **culture specific comparison** between simple types
- Can compare types **which implement IComparable interface**

```
String str1 = “First”;
String str2 = “Second”;
Console.WriteLine(“Result1: {1}”, Comparer.Default.Compare(str1,str2));

int firstNumber = 35;
int secondNumber = 23
Console.WriteLine(“Result2: {1}”, Comparer.Default.Compare(firstNumber,
    secondNumber));
```

o/p Result1: -1
Result2: 1

Note: This comparison **doesn’t give** any idea of the **magnitude of the difference**

Comparer class

Points to consider

1. Ensure that objects passed for comparing are of same type, else exception will be thrown.
2. Objects passed for **Comparer.Compare()** are checked to see **if they implement IComparable**. If yes, then that implementation is used.
3. **Null values** are **allowed**, and **interpreted as less than** any object
4. String comparison is done using **current culture**. Its possible to compare string using different culture.
5. String comparison is case sensitive. **CaseInsensitiveCompare** class can be used for insensitive comparison

Sorting Collections using IComparable and IComparer

- Sorting on collections is supported
  - Either by default comparison
  - Custom methods

- Custom class should implement their own implementation for comparison

ArrayList.Sort()

- Method can be used with **no parameters**, then **default comparison** can be used
- Method can be invoked by passing **IComparer interface** for comparing pairs of objects

Note: Sorting is not implemented in many collection classes e.g. CollectionBase.

Collection Sorting Example

```
class person : IComparable
{
    public int CompareTo(object obj)
    {
        if(obj is person)
        {
            person otherPerson = obj as person;
            return this.age – otherPerson.Age;
        }
    }
}
```

Collection Sorting Example

```
public class PersonComparer : IComparer
{
    public static IComparer Default = new PersonComparer ();

    public  int Compare (object x, object y)
    {
    return Comparer.Default.Compare(((Person) x).Name, ((Person)
    y).Name);
    }
}
```

Collection Sorting Example

```
ArrayList list = new ArrayList();
list.Add(new person("Jim", 30))
list.Add(new person("Bob", 25))
list.Add(new person("Bert", 27))
```

```
list.Sort()
Bob 25
Bert 27
Jim 30
```

List.Sort ()checks if the object it is containing if it has implemented **CompareTo()**

If yes, then that's get called else default sorting mechanism will be invoked

```
list.Sort(PersonComparer.Default)
Bert 27
Bob 25
Jim 30
```

Conversions

- For **converting object of one type to another**.
- Conversion of **unrelated objects** require conversion implementation.
  - Implicit Conversion
  - Explicit Conversion

- unrelated objects means they are **not inherited** from same tree **nor** do they **implement same interface**

Internal Conversion

```
Public class ConvClass1
{
    public int val;
    public static implicit operator ConvClass2(ConvClass1 op1)
     {
        ConvClass2 obj_class2= new ConvClass2();
         obj_class2.val = op1.val;
        return obj_class2;
     }
}
```

```
ConvClass1 op1 = new ConvClass1();
op1.val = 3;
ConvClass2 op2 = op1;
```

Can we convert Op2 to Op1 implicitly?

Explicit Conversion

```
Public class ConvClass2
{
    public double val;
    public static explicit operator ConvClass1(ConvClass2 op2)
     {
        ConvClass1 obj_class1= new ConvClass1();
        checked {obj_class1.val = (int) op2.val;};
        return obj_class1;
     }
}

ConvClass2 op2 = new ConvClass2();
op2.val = 3e15;
ConvClass1 op1 = (ConvClass1)op2;
```

as Operator

- **as** operator is used to **perform conversions between compatible types**

  <operand> as <type>

Rules

- If <operand> is of type <type>
- If <operand> can be **implicitly converted** to type <type>
- If <operand> can be **boxed** into <type>

Note:

null will be returned if conversion fails. No exception will be thrown.

as Operator

```
class MyClass1
{….}
public static void Main()
  {
     object [] myObjects = new object[6];
     myObjects[0] = new MyClass1();
    myObjects[1] = "hello";
     myObjects[2] = 123;
     myObjects[3] = 123.4;
     myObjects[4] = null;

     for (int i=0; i<myObjects.Length; ++i)
     {
       string s = myObjects[i] as string;
       Console.Write ("{0}: ", i);
       if (s != null)
         Console.WriteLine ( """ + s + """ );
       else
         Console.WriteLine ( "not a string" );
     }
```

```
0: not a string
1: 'hello'
2: not a string
3: not a string
4: not a string
```

References

- Book referred “Beginning Visual C# 2010” by Wrox publication.

- C# tutorial

  http://cplus.about.com/od/learnc/ss/csharptutorial_5.htm

- **Assignment I**

- Here’s a small application to be created to demonstrate Object Oriented features in .NET using C#.
- Accounts are created & managed by the application for a Bank. There are 2 types of account that can be created
- 1. Savings
- 2. Current
- • The opening balance for Saving account is 1000, & for Current is 5000
- • The minimum balance for Saving Account is 1000 & that for Current should not be less than the ODA.
- • Account ID should be auto generated, Name can be modified & Balance can be updated only through transactions.
- Based on the above requirement, build a sample test application
- Note: This is a simple application to demonstrate OOP features.

Assignment II

1. Write a program that reads an integer number and calculates and prints its square root. If the number is invalid or negative, print "Invalid number". In all cases finally print "Good bye". Use try-catch-finally.
2. Write a method `ReadNumber(int start, int end)` that enters an integer number in given range [start…end]. If an invalid number or non-number text is entered, the method should throw an exception. Using this method write a program that enters `10` numbers:

$$a_1, a_2, \ldots a_{10}, \text{ such that } 1 < a_1 < \ldots < a_{10} < 100$$

Practical Assignments

- Console based
  - WAP to print 100 to 1 on screen in the set of 10
  - WAP for implicit () and explicit conversion
    - float to double
    - float to short
    - boolean to string
    - Checked , unchecked , application level configuration
  - WAP for String based Switch case
    - MH = print Maharashtra
    - DL = Delhi
  - Write a program Swap function using Ref parameter and Out parameter

Summary

Question Bank

- Write a note on Static Constructors and Private Constructors
- What is Extension Methods in C#? Explain with example.
- What is Polymorphism? Describe Interface Polymorphism.
- What is an Interface? Describe Interface Polymorphism.
- Explain the usage of “override” and “new” keyword.
- Write a note on Indexer. Give example.
- Explain IComparable and IComparer interface.

- Write a program to implement Custom Exception. Create InvalidStudentNameException class in a school application, which does not allow any **special character** or **numeric value** in a name of any of the students. Use **Regex("^[a-zA-Z]+$")** to check Student Name