

C#.net
(c sharp dot net)

Gurchet

What is .Net?

- .Net is a new development platform that creates web applications very fast and support cross language development (sharing programs of one .Net compatible language with program written in other .Net compatible language). The Microsoft .Net Framework is a platform that provides tools and technologies you need to build Networked Applications as well as Distributed Web services and Web applications.

.Net Programming Languages

1. Visual Basic.Net
2. C#
3. Fortran
4. Pascal
5. C++
6. Perl
7. Java Language
8. Python
9. COBOL
10. Microsoft Jscript

What is .Net Frame Work

- The Microsoft .NET Framework is a Computing model that make things easier for application development for the Distributed environment of the Internet.
- The .NET Framework is an environment for Building, deploying, and running Web Services and other applications.

Versions of .Net Framework

<u>Version</u>	<u>Released</u>
• .Net Framework 1.0	Visual Studio 2002
• .Net Framework 1.1	Visual Studio 2003
• .Net Framework 2.0	Visual Studio 2005
• .Net Framework 3.5	Visual Studio 2008
• .Net Framework 4.0	Visual Studio 2010
• .Net Framework 4.0	under construction

CLR (Common Language Runtime)

- The Common Language Runtime (CLR) is the environment where all programs in .NET are run. It provides various services, like memory management and thread management. Programs that run in the CLR need not manage memory, as it is completely taken care of by the CLR.
- For example, when a program needs a block of memory, CLR provides the block and releases the block when program is done with the block. All programs targeted to .NET are converted to MSIL

MSIL or IL

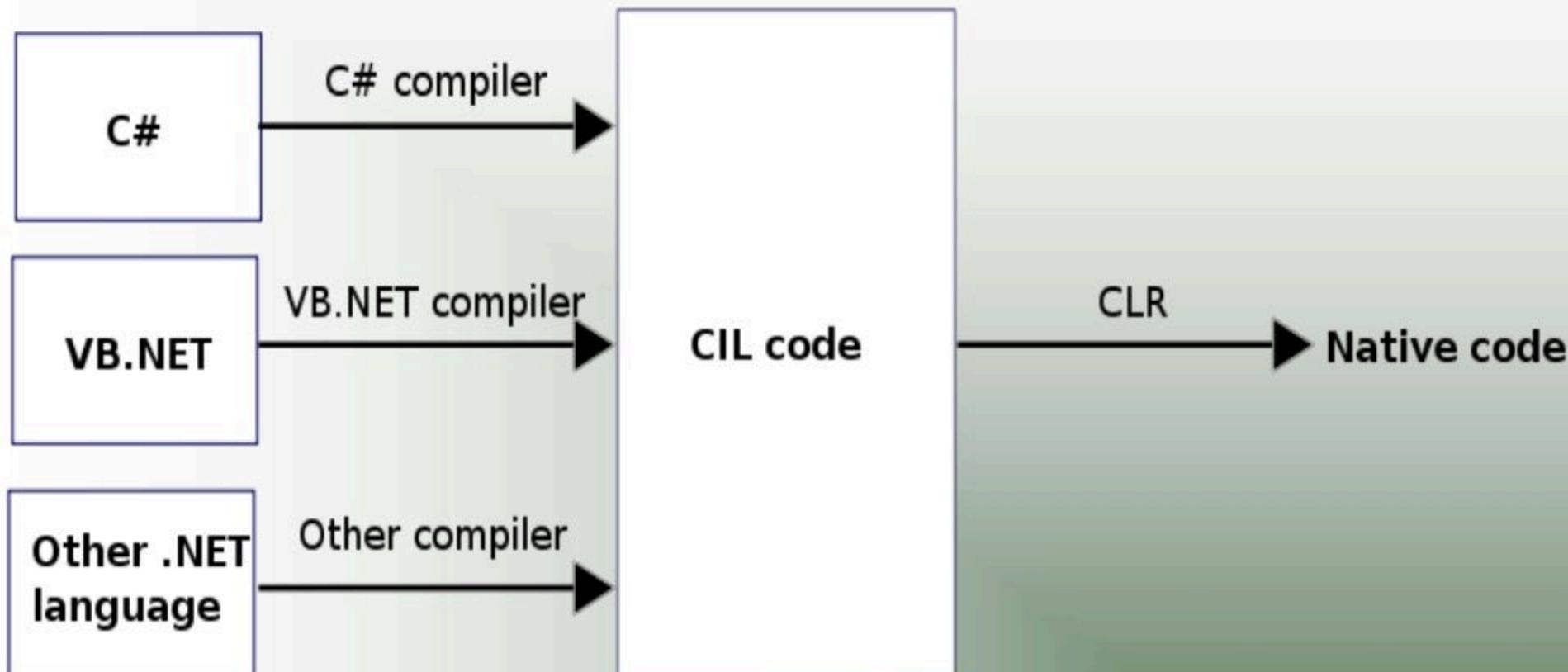
1. .NET languages are not compiled to machine code. They are compiled to an **Intermediate Language (IL)**.
2. CLR accepts the IL code and recompiles it to machine code. The recompilation is just in-time (JIT) meaning it is done as soon as a function or subroutine is called.
3. The JIT code stays in memory for subsequent calls. In cases where there is not enough memory it is discarded thus making JIT process interpretive.

(Microsoft Intermediate Language).MSIL is the output of language compilers in .NET .MSIL is then converted to native code by JIT (Just-in Time Compiler) of the CLR and then native code is run by CLR.

Source code

Bytecode

Native code



Compile time

Runtime

◆ CLR Execution Model



APPLICATION OF .NET

- The .NET Framework can be used to create many types of application.
- Windows applications
- Web application
- Console applications

Console applications

- **Console applications** run from the command prompt. Before the days of windows, computers ran on what we know as DOS. All written output from the application is shown in the command prompt window. The .NET Framework uses C# technology to create console applications.

Console Application



Windows applications

- **Windows applications** are known in the .NET Framework as Windows Forms applications. These look like your normal applications that you run on the windows platform,they include buttons, text boxes, drop down list, graphics . The .NET Framework uses VB(visual basic technology) to create window applications.

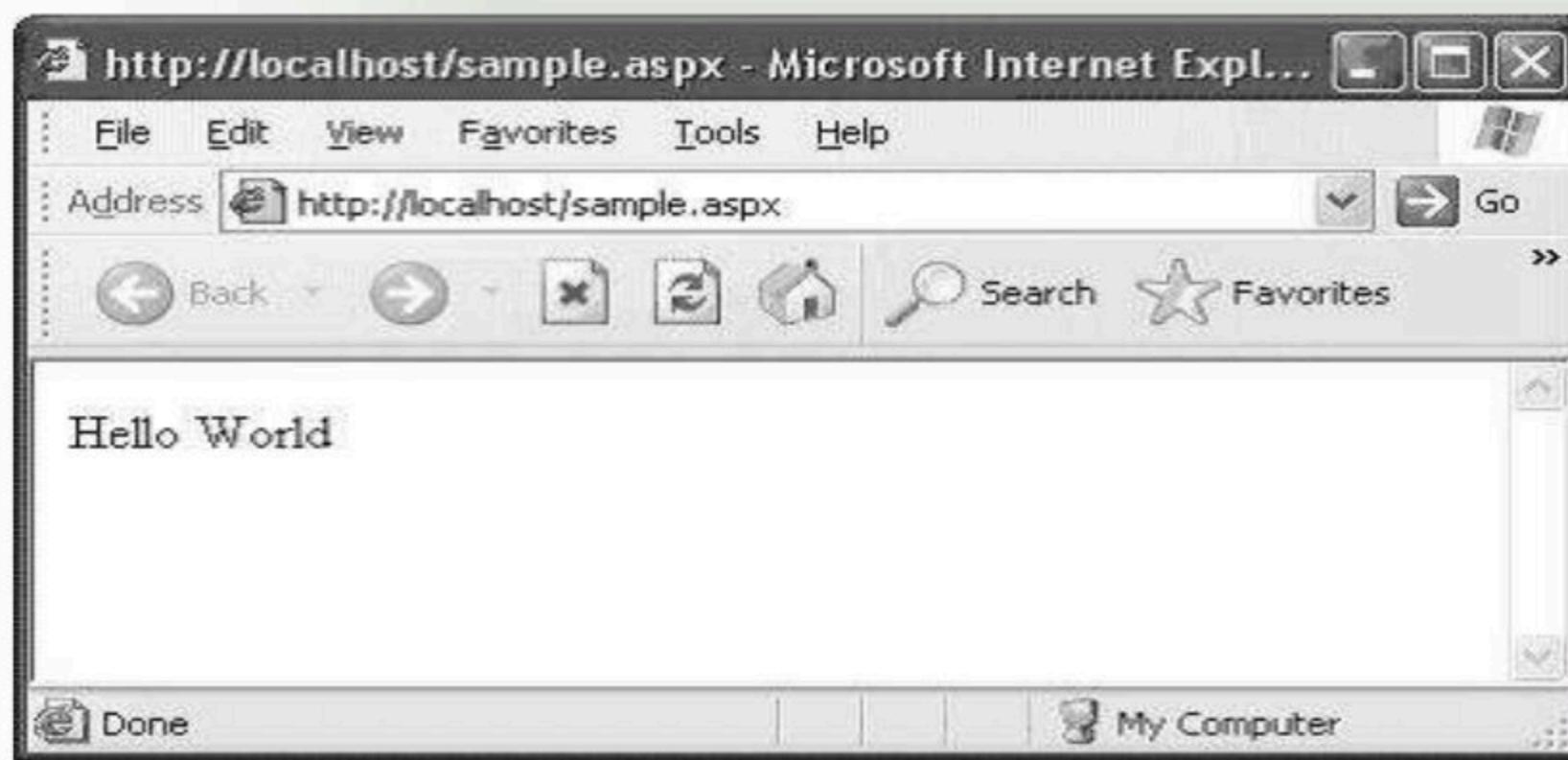
Windows Application



Web applications

- **Web applications** appear on a web page, with the output appearing on the same page.
- The .NET Framework uses ASP technology to create web applications.

Web Application



Introduction to C#

Father of C#

**Anders
Hejlsberg**



What is C#

- C# is an object-oriented language that is based on the .NET framework. C# enables you to develop different types of applications such as Windows Based applications,
Console Based applications,
Web Based applications.

C# contains various features that make it similar to other programming languages such as C,C++, and java .However, there are some additional features in C# that make it different from other programming languages such as c++ and java.

C# Language Features

- C# was developed as a language that would combine the best features of previously existing Web and Windows programming languages.
1. Modern
 2. Object Oriented
 3. Simple and Flexible
 4. Type safety
 5. Scalable
 6. Cross Platform Interoperability

- **Modern**
- C# has been based according to the current trend and is very powerful and simple for building interoperable, scalable, robust applications.
- C# is the best language for developing web application and components for the Microsoft .NET platform.
- **Object Oriented**
- As mentioned, C# is an object-oriented language. It supports all the basic object oriented language features: encapsulation, polymorphism, and inheritance.

- **Simple and Flexible**
- C# is as simple to use as Visual Basic, in that everything in C# represented as an object. All data type and components in C# are objects.
- Pointers are missing in C#.
- In C# there is no usage of "::" or "->" operators
- **Type safety**
- C# is a type safe language. All variables and classes (including primitive type, such as integer, Boolean, and float) in C# are a type, and all type are derived from the object the object type.
- Arrays are zero base indexed and are bound checked.
- Overflow of types can be checked.

- **Scalable**
- .NET has introduced assemblies, which are self-describing by means of their manifest. Manifest establishes the assembly identity, version, culture etc. Assemblies need not to be register anywhere.
- To scale our application we delete the old files and updating them with new ones.

Language and Cross Platform Interoperability

- C#, as with all Microsoft .NET supported language, shares a common .NET run-time library. The language compiler generates intermediate language (IL) code, which a .NET supported compiler can read with the help of the CLR. Therefore, you can use a C# assembly in VB.NET without any problem, and vice versa.

Application of C#

- C# is modern and very power full language. its 13,000 classes. C# is used in developing window .web and device application.
 - Their some applications are
1. **Developing console application** - in console application you make chat servers, user defines classes.
 2. **Developing Windows Application:** in window application use make software for application requirement. Now c# use the flexibility of Visual Basic so window bases software are develop in c# in very easy.

3. **Developing Web Application:** C# is used in ASP.NET so web application is very easy to develop using c# and Asp.net.
4. C# is used to create web services.
5. C# is used to create both web and window Component.
6. Mobile Application is also developed in C#.
7. C# is used for making games.

Difference between c++ and c#

<u>C++</u>	<u>C#</u>
1. In c++ first character of the main method is small.	1. In C# main method is always capital.
2. In c++ program is start with #include statement.	2. C# program is start with using System namespaces.
3. Pointers are used in c++.	3. In C# pointer features are dropped.
4. In c++ web application is not develop.	4. In C# we develop both web and mobile application.

C++

5. c++ support default arguments.
6. C++ does not contain any framework.
7. Error handling is not better in C++.
8. In c++ we use cin and cout for input and output .

C#

5. C# does not support default arguments
6. C# is contained within .Net framework.
7. Error handling is better in C# because in C# intellisence feature makes program error free.
8. C# we use Console.WriteLine and Console.ReadLine () for Input and Output.

C++

9. C++ uses macros.
10. In C++, to create an object of a class cannot use new keyword.
11. C++ supports multiple inheritance.
12. C++ uses semicolon at the end of the class definition.

C#

9. C# does not support.
10. Objects in C# can only be created using new keyword.
11. C# does not support multiple inheritance.
12. C# does not use semicolon at the end of the class definition.

Difference between C# and Java

C#

1. In C# main method is always capital.
2. C# supports namespaces.
3. In C# we use `Console.WriteLine` and `Console.ReadLine()` for Input and Output.

Java

1. In Java main method is always small.
2. Java supports packages.
3. In Java we use `System.out.println()` and `System.in.read()` for input and output.

C#

4. C# supports operator overloading.
5. C# allows variable number of parameters using the param keyword.
6. C# provides for each iteration statement.
7. C# provides static constructors for initialization.
8. C# supports the structure.

Java

4. Java does not support operator overloading.
5. Java does not use param keyword.
6. Java does not provide for each statement.
7. Java does not provide.
8. Java does not support.

C#

9. C# allows switch statements to operate on strings.
10. C# supports events and delegates.
11. In C#, the declaration of an array is:-
`int []var=new int[size];`
12. C# uses ref ,out keywords.
13. To declare a constant value then use 'const' keyword.

Java

9. Java does not allow.
10. Java doesn't support events and delegates.
11. In Java, the declaration of an array is:-
`int var[]={};`
12. Java does not use ref ,out keywords.
13. To declare a constant value then use 'final' keyword.

What is Namespace

- A **Namespace** in Microsoft .Net is like containers of objects. They may contain *unions, classes, structures, interfaces, enumerators and delegates.*

Standard Namespaces in .NET

- In C#, you can use various namespaces provided by .NET framework to access classes contained in them. The various namespaces available in .Net framework that can be used in C# are as follow:-
- **System** :- It contains classes that allow you to perform basic operations such as mathematical operations and data conversion.
- **System.IO** :- It contains classes used for input and output operations for a file in C#.
- **System.Net** :- It contains classes that are used to define network protocols.

- **System.Data** :- It contains classes that are used to make ADO.NET data access architecture.
- **System.Collections**:- It contains classes that implement collection of objects such as list.
- **System.Drawing** :- It contains classes that are used to implement GUI functionalities.
- **System.Web**:- It contains classes that help implement HTTP protocol to access web pages.

- **What is public static void Main ()**
- **public** The keyword public is an access modifier that tells the c# complier that the Main method is accessible by anyone.
- **static** The keyword static declares that the main method is global one and can be called without creating instance of the class.
- **void** The keyword void is a type modifier that states that the main method does not return any value but simply prints some text to the screen.
- **Define method name Main.** Every c# executable program must include The Main() method .C# application can have any number of classes but ‘only one’ class can have Main method to start execution.

Adding Comments in Program

- Comments play very important role in the maintenance of programs. They are used to simplify readability and understanding code. All programs should have information Such as implementation, details of class, and others.
- C# use two type of comments**
 - Single line comment
 - Multiline comments
 - // this is single line comments
 - /* this is

Example Of multi line
Comments

*/

Program Freeform Style

- C# is a freeform language. We need not indent any lines to make the program work properly. C# does not care where on the line we begin coding. Although several alternative styles are possible to write codes of C# program.

```
System.Console.WriteLine ("Hello!");
```

// can be written as:

```
System.Console.WriteLine  
("Hello!"); // or even as:
```

```
System.Console.WriteLine  
("Hello!"  
);
```

Using Aliases for Namespaces

- We can avoid system namespace when console is write because we use Using System;
- So no need to write system namespace when class is used so we write Console.WriteLine instead of System.Console.WriteLine
- In this way we also use names of any class by saying this is aliases of that class

- **using A=System.Console ; // A is alias for System.Console**
- **using B=System.Math ; // B is alias for System.Math**
- **Class abc**
- **{**
- **public static void Main()**
- **{**
- **double x;**
- **X=B.Sqrt(100);**
- **A.WriteLine("square of 100 is: ");**
- **A.ReadLine();**
- **} }**

Command Line Arguments

- Command Line Arguments allows a program to behave in a specific manner depending on the input provided at the time of execution. Command line arguments stored in the string array and specific with the Main() method.

Multiple Main Methods

- C# allows you to define more than one class with Main() method. The Main() method indicates the entry point for execution of a program. As a result, where you define more than one Main() method in a C# program then there are more than one entry points for execution of a program. You need to specify the class name with the Main() method, which you want to execute first.

Structure of C# program

- A C# program contains multiple class definitions, which are the primary elements of the program. A C# program contains following sections:

Documentation Section	optional
Directive Section	optional
Interface Section	optional
Class Section	Essential
Main Method Section	Essential

- **Documentation Section**
This section contains comments such as name of the program and date on which the programmer started creating the program. The documentation section must provide the information about the classes defined in the program.
- **Directive Section**
This section includes all the namespaces that contain classes required for the execution of the program.

- **Interface Section**

The Interface section contains the declarations of interfaces that are used in the program. Interfaces are used to implement the section of multiple inheritance.

- **Main Method Section**

The Main method is an essential section of a C# program. It helps to creates of various classes of the program and establishes communication between them.

Tokens

- A token is defined as the smallest unit of a program. Java tokens are classified into keywords, identifiers, constants, operators and punctuators.

1. Keywords:- Keywords are some special word for a programming language. Keywords has some specific meaning for the compiler. It is also known as Reserved words. Like void, int, char, float, else ,if etc.

2. Identifiers:- Identifiers are the names given to the uniquely identified various programming elements like variables, arrays, methods, classes, objects, packages, interface and so on. rules:-

- An identifier must be unique in a program.
- Alphabets, digits, underscore and dollar sign characters can be used in an identifier.
- An identifier must not start with a digit.

3. Constants:- Constants, also known as literals, are the values that a program cannot alter during its execution.

- Numeric constants
- Character constants
- String constants
- Boolean constants

4. Operators:- Operators are some special symbols which work on operands. Such as addition, subtraction, multiplication etc.

5. Punctuators:- Punctuators, also known as separators, are the symbols that define the structure of a program by dividing and arranging a set of codes. The various punctuators defined are braces '{ }', colon ':', comma ',', period '.', semicolon ';' and parentheses '()'.

Data types

Data type is used to define the type of data

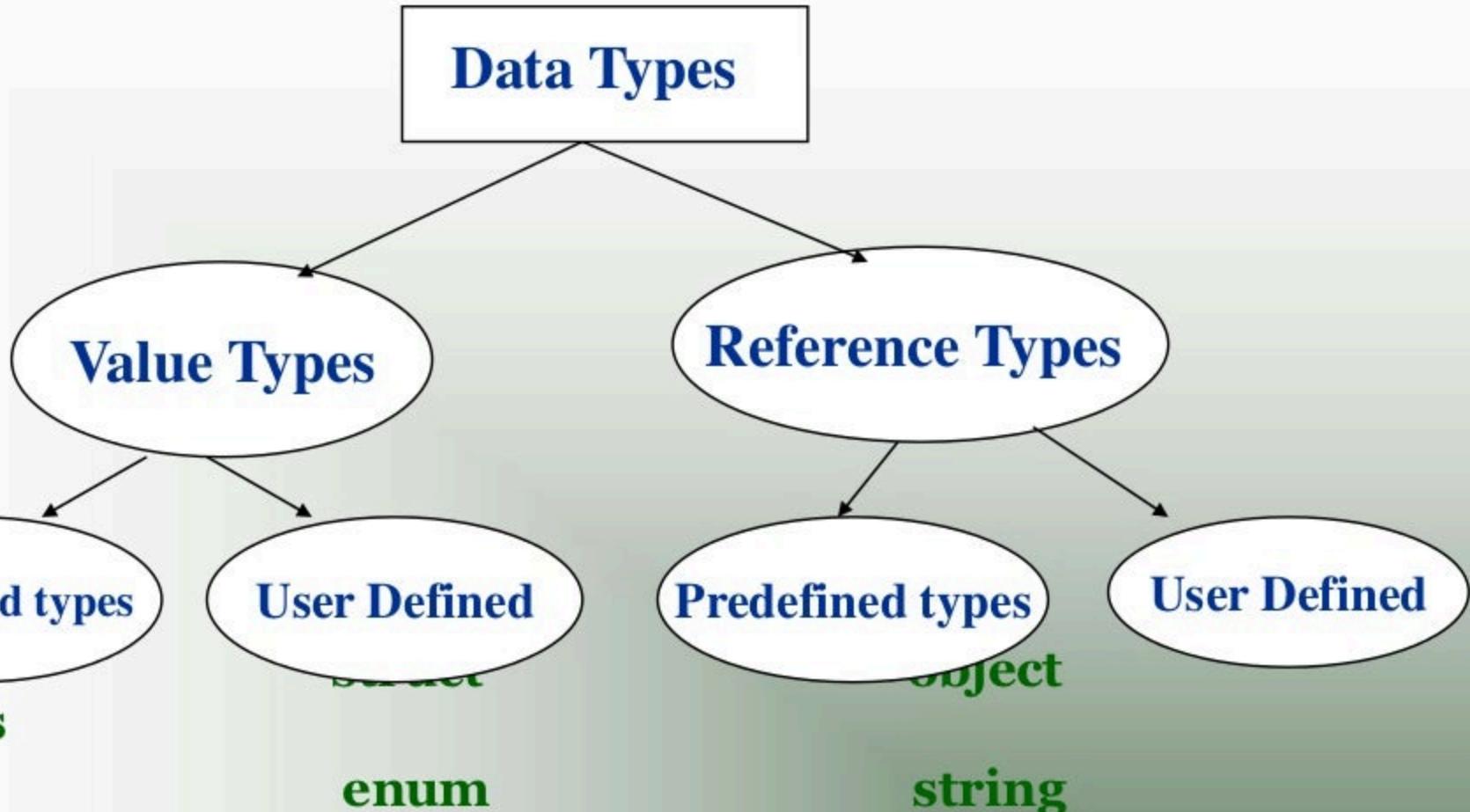
and size of data. It tells the program which

type of data enter into the program.

Data types in C# are classified into two types:-

- 1. Value types**
- 2. Reference types**

Data Types



- **int**

- **classes**

- **float**
- **interfaces**

- **long**
- **delegates**

- **double**

- **char**

- **boolean**

- **enum**

- **string**

- **array**

Value types

- Variable defined as value types are stored in the stack and when they are assigned to other variables their value is copied to that variable. Value types can be further classified into two categories:-
 1. Predefined types
 2. User defined

- Predefined types

1. int :- It is used to accept the numeric values. It can accept whole numbers(0 to 9).

It allocate 4 bytes memory to each variable.

its range -2,147,483,648 to
2,147,483,647.

2. long :- It is also used to accept whole numbers (0 to 9). It allocate 8 bytes memory to each variable. Its range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

3. float :- It is used to accept with or without decimal numbers. It allocate 4 bytes memory to each variable .but when we use float data type then use f letter with float value. Its range 1.5×10^{-45} to 3.4×10^{38} .

4. double:- It is used to used for accepting numeric values.it can accept whole numbers and decimal numbers. It allocate 8 byte memory to each variable.Its range 5.0×10^{-324} to 1.7×10^{308} .

5. char :- It is used to accept a single character in single quotes.It can accept single character only. It takes 2 bytes memory.

Reference types

- When a variable is copied to another variable only the reference of variable is copied and the actual value remains in the same memory location. The reference type create two reference with a single value. Reference type is classified into two categories:
 1. User-defined types
(class,interface,delegates and array)
 2. Predefined type, which include two data types,
 - i. Object type
 - ii. String type

- Predefined C# reference types
- String: Represents a string of Unicode characters. It allows easy manipulation and assignment of strings. Strings are immutable, meaning that once it is created it can't be modified. So when you try to modify a string, such as concatenating it with another string, a new string object is actually created to hold the new resulting string.
- Object: Represents a general purpose type. In C#, all predefined and user-defined types inherit from the object type or System.Object class.

LITERALS in C#

- *Literals*: are the way in which the values that are stored in variables and represented.
- Literals are the value constants assigned to variables in a program in a c# program
- *Examples*
- string name='Summar', float p=1.02
- There are three types of Literals in c#.
 1. Numeric Literals
 2. Boolean Literals
 3. Character Literals
 4. String Literals

- *Numeric Literals* are two type integer Literals and Real Integer.

An Integer Literal refers to a sequence of digits. There are two types of integers, namely decimal integer and hexadecimal integers.

Decimal integer consists of a set of digits, 0 through 9, preceded by an optional minus sign.

- *Valid examples of integer literal are*
- 123 -321 0 654321
- Hex integers Sequence of digits proceeded by ox ox2 ox9f oxbcd ox

- **Real Literals:** Integer literals are inadequate to represent quantities that vary continuously such as distances, height temperatures, process and so on. Numbers containing fractional literals like

17.528 represents these quantities. Such numbers are called real numbers.

Examples are **0.0083, -0.75, 435.36**

- **Boolean Literals:** There are two Boolean literal values:

True

False

They are used as values of relational expressions.

- **Single Character Literals** A single character literal contains a single character enclosed within a pair of single quote marks. Example of character in the above constant are: ‘5’ ‘x’
- **String Literals** A string literal is a sequence of characters enclosed between double quotes. The characters may be alphabets, digits, special character and blank spaces.

Example “Hello c#” “2001” “5+3” etc.

Constant

- Constant variables do not change their value during the execution of a program. The constant variables should be declared or initialized at the declaration time.

```
const int a=10;
```

- The constant variables cannot be assigned a variable expression such as int a=n*10 because that generates error in the program.

Variables

- Variable is a name of memory location where we can store the particular data. The value of the variable varies during the execution of the program.

Naming conventions:

- A variable name always start with alphabet.
- A variable can't contain a blank space.
- We can not create more than one variable with same name in same scope. We can't used keyword as a variable name.
- A variable name should be meaningful and short.

- “underline” character can be used in variable names like **first_name**, **roll_no** etc.

The syntax for declaring variable is:-

data_type variable_name;

The variables having Default values

Type	Default value
integer type	0
char type	'\x000'
float type	0.0f
double type	0.0d
decimal type	0.0m
bool type	False
Enum	0
Reference type	Null

Scope of Variables

- In a program, a variable can be accessed within a particular block or can be accessed in the whole program. This is known as scope of variable. various types of variables in C#.
- **Static variable :-** The static variables are those variables which retain their value through the execution of the program. These variables are declared as local variables.

- **Instance variables:-**In C#, one of the variables of a class that may have a different value for each object of that class. The instance variables hold the state of an object.
- **Local variables :-** These variables are visible only within function in which it appears.

Boxing and Unboxing

- **Boxing**
- The process of converting from a value type to a reference type is called boxing. Boxing is an implicit conversion.
- **int x = 66;**
- **Object z = x;**

- **Unboxing**
- The process of converting from a reference type to a value type is called unboxing. unboxing is an explicit conversion.
- **Object obj = 123;**
- **int num1 = (int)obj;**

Expressions and Operators

- *An expression is a sequence of operators and operands that specify some sort of computation. The operators indicate an operation to be applied to one or two operands.*
- *For example, the operators + and - indicate adding and subtracting operands.*

Type of Operators in C#

7 type of operators

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment Operator
5. Conditional operator
6. Bitwise operator
7. Increment and decrement operator

Arithmetic operators

- * Multiplication
- / Division
- + Addition
- - Subtraction

Relational operators OR comparison operators

- $x == y$ Returns true if x is equal to y
- $x > y$ Returns true if x is greater than y
- $x >= y$ Returns true if x is greater than or equal to y
- $x < y$ Returns true if x is less than y
- $x <= y$ Returns true if x is less than or equal to y
- $x != y$ Returns true if x is not equal to y

Relational operators OR comparison operators

Operator	English	Example	Result
=	Equal To	$4 = 3$	False
<	Less Than	$4 < 3$	False
>	Greater Than	$4 > 3$	True
<=	Less Than Or Equal To	$4 <= 3$	False
>=	Greater Than Or Equal To	$4 >= 3$	True
!=	Not Equal To	$4 != 3$	True

Logical operators

- NOT (!)
- AND (&&)
- OR (||)

Bitwise Operator

- AND (&)
- OR (|)
- XOR (^).

Assignment Operator

- The assignment operator assigns the value of an expression to a variable.
- $- =$
- $+ =$
- $* =$
- $/ =$

Increment and decrement operator

- The **increment and decrement operator increase or decrease the value of an operands by 1.** C# provide unary operator
- **++**
- **-**

Conditional operator OR

Ternary operator

- <expression1> ? <expression2> :
<expression3>

The checked and unchecked operators

- The checked and unchecked operators are two new features in C# for C++ developers.
- These two operators force the CLR to handle stack overflow situations.
- checked and unchecked operators are provided by c#. which can be used for checking or unchecking stack overflows during program execution.
- If an operation is checked , then an exception will be thrown if overflow occurs.
- If it is not checked, no exception will be raised but we will lose data.

```
• using System;  
• class xyz  
• {  
•     public static void Main()  
•     {  
•         int a=200000;  
•         int b=300000;  
•         try  
•         {  
•             int m=checked(a*b);  
•             Console.WriteLine("value of m is: "+m);  
•         }  
•     }  
• }
```

- catch(OverflowException e)
- {
- Console.WriteLine(e);
- }
- }
- }
- }
- In this program $a*b$ produces a value that will easily exceed the maximum value for an “int”, and overflow occurs. As the operation is checked with operator **checked**, and overflow exception will be thrown

There are 3 categorizes the operators

1. The unary operators take **one operand** and use either a prefix notation (Such as $-x$) or postfix notation (such as $x++$).
2. The binary operators take **two operands** and all use what is called infix notation, where the operator appears between two objects
(such as $x + y$).
3. The ternary operator takes three operands and uses infix notation (such as $c? x: y$). Only one ternary operator, `?:`, exists.

Mathematics in C#

- *To perform the basic algebraic and geometric operations in C#, you can use*
- *methods of the **Math** class of the .NET Framework.*

- **How to Find Minimum Value between two numbers using Math Class**
 - using System;
 - class abc
 - {
 - public static void Main()
 - {
 - int x = 80;
 - int y = 73;
 - Console.WriteLine("The minimum number between x and y are "+ Math.Min(x,y));
 - Console.Read();
 - }
 - }

- **How to Find Maximum Value between two numbers using Math Class**

- using System;
- class Program
- {
- public static void Main()
- {
- int n1 = 80;
- int n2 = 73;
- Console.WriteLine("The maximum number ",
Math.Max(n1, n2));
- Console.Read();
- }
- }

- **How to find Absolute Value**
- using System;
- class Program
- {
- public static void Main()
- {
- int num = -786;
- Console.WriteLine("Original Value = ", num);
- Console.WriteLine("Absolute Value =",
Math.Abs(num));
- Console.Read();
- }
- }

Ceiling of a Number

- In arithmetic, the ceiling of a number is the closest integer that is greater or higher than the number considered. In the first case, the ceiling of 12.155 is 13 because 13 is the closest integer greater than or equal to 12.155. The ceiling of –24.06 is –24.

- using System;
- class Program
- {
- static int Main()
- {
- double value1 = 155.55; double value2 = -24.06;
- Console.WriteLine("The ceiling of {0} is {1}",
value1, Math.Ceiling(value1));
- Console.WriteLine("The ceiling of {0} is {1}\n",
value2, Math.Ceiling(value2));
- return 0;
- }
- }

The Floor of a Number

- The lowest but closest integer value of a number is referred to as its floor.
- Example:
- double value1 = 1540.25;
- double value2 = -360.04;
- Console.WriteLine
- ("The floor of value1,Math.Floor(value1));
- Console.WriteLine
- ("The floor of value2, Math.Floor(value2));

Conditional Statements in C#

- The actions that a program takes are expressed in statements. Common actions include
 - declaring variables, assigning values, calling methods, looping through collections, and
 - branching to one or another block of code, depending on a given condition. The order in which
- statements are executed in a program is called the *flow of control or flow of execution*. The flow
- of control may vary every time that a program is run, depending on how the program reacts to
- input that it receives at run time.

Types of Statements

- 1. Block Statement
- 2. Declaration statements
- 3. Expression statements
- 4. Selection statements
- 5. Jump statements
- 6. Iteration statements

There are three categories of program control statements:

- 1. Selection statements: which are the if and the switch
- 2. iteration statements: which consist of the for, while, do-while, and for each loops
- 3. Jump statements: which include break, continue, goto, return, and throw.

```
• Console.WriteLine("What is your name? ");
• string name = Console.ReadLine();
• switch (name)
• {
•     case "HHH":
•         Console.WriteLine("My name is HHH.");
•         goto case "Hunter"; // jump to Hunter so no break is used
•     case "Hunter":
•         Console.WriteLine(" HHH nick name is Hunter");
•         break;
•     case "HBK":
•         Console.WriteLine("My name is HBK.");
•         break;
•     default:
•         Console.WriteLine("your name is not store in our database");
•         break;
• }
```

Jump statement: Break Statement

- for (int i=0; i<10; i++)
- {
- int j = i*i;
- Console.WriteLine(i);
- if (j == 9)
- break;
- Console.WriteLine(j);
- }

Jump statement: Continue Statement

- for (int i=0; i<10; i++)
- {
- int j = i*i;
- Console.WriteLine("i is = " + i);
- if (j == 9)
- Continue;
- Console.WriteLine("j is =" + j);
- }

Array in C#

- An array is a data structure that contains a number of variables of the same type. Arrays
- are declared with a type:
- **type[] arrayName;**
- Arrays can be divided into four categories. These categories are
 - 1. Single-dimensional Arrays
 - 2. Multidimensional Arrays Or Rectangular Arrays
 - 3. Jagged Arrays

- In C#, the lower index of an array starts with 0, and the upper index is number of item minus 1.
- **Single-dimensional Array**
- Single-dimensional arrays are the simplest form of arrays. These types of arrays are
 - used to store number of items of a predefined type. All items in a single dimension array are
 - stored in a row starting from 0 to the size of array -1. You can initialize array item either during
 - the creation of an array or later by referencing array item

- Print 5 Number Using Single Dimension Array
- int[] x = new int[5];
- x[0] = 11;
- x[1] = 21;
- x[2] = 31;
- x[3] = 41;
- x[4] = 51;
- for (int y = 0; y < 5; y++)
- {
- Console.WriteLine(x[y]);
- }

- Print sum of 5 numbers.
- int[] x;
- x = new int[5] {1, 2, 3, 4, 5};
- int sum = 0;
- for (int y = 0; y < 5; y++)
- {
- sum = sum + x[y];
- Console.WriteLine(x[y]);
- }
- Console.WriteLine("sum of all these numbers are:" + sum);

- Print some name using single dimension Array
- string[] x = { "ram", "sofia", "karn", "king" };
- int l = x.Length;
- for (int t = 0; t < l; t++)
- {
- Console.WriteLine("name is " + x[t] + "\n");
- }

- **Multi Dimension Arrays**
- A multidimensional array is an array with more than one dimension. A multi dimension array is declared as following:
- **string[,] name;**
- After declaring an array, you can specify the size of array dimensions if you want a fixed size array or dynamic arrays. For example, the following code two examples create two multi dimension arrays with a matrix of 3x2 and 2x2.
- The first array can store 6 items and second array can store 4 items respectively.
- `int[,] n = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };`
- `string[,] n = new string[2, 2] { {"Summar", "Gagan"}, {"Sony", "Sharan"} };`

- **Example of two dimession matrix**
- int r, c;
 - Console.WriteLine("Enter rows ");
 - r = int.Parse(Console.ReadLine());
 - Console.WriteLine("Enter cols ");
 - c = int.Parse(Console.ReadLine());
 - Console.WriteLine("\n ===== \n");
 - int[,] x = new int[r, c];
 - int lr = 0;
 - int lc = 0;
 - //this loop is for intput row and column from user
 - for (lr = 0; lr < r; lr++)
 - {

```
• for (lc = 0; lc < c; lc++)
• {
•     Console.WriteLine("row " + lr + " col " + lc + ">>
");
•         x[lr, lc] = int.Parse(Console.ReadLine());
•     }
• }
• //this loop is for print row and column from
user
• for (lr = 0; lr < r; lr++)
• {
•     Console.WriteLine();
•     for (lc = 0; lc < c; lc++)
•     {
•         Console.Write(x[lr, lc] + "\t");
•     }
• }
```

Variable Size Array

- Variable size array are the array whose dimensions are unknown to the user. The number of dimensions determines the rank of the array.
- Example:
- Single Dimension Array
- `int []i=new int[]{1,2,4,5,7,8};`
- Multi-dimension Array
- `int[,] aa = new int[,] { {1,2},{3,4},{5,6},{7,8},{9,1}};`
- `Console.WriteLine("Count Rows =" +aa.GetLength(0));`
- `Console.WriteLine("Count cols= " +aa.Rank);`
- `int r = aa.GetLength(0);`
- `int c = aa.Rank;`

- **Jagged Arrays**
- Jagged arrays are often called array of arrays. An element of a jagged array itself is an array. For example, you can define an array of names of students of a class where a name itself can be an array of three strings - first name, middle name and last name. Another example of jagged arrays is an array of integers containing another array of integers.
- *For example*
- `int[][] x = new int[][] { new int[] {1,3,5}, new int[] {2,4,6,8,10} };`

```
• int[][] x = new int[2][];
• 
• 
• 
• 
• x[0] = new int[2];
• x[1] = new int[2];
• for (int i = 0; i < 2; i++)
• {
•     for (int j = 0; j < 2; j++)
•     {
•         Console.WriteLine("enter array value =
");
```

• x[i][j] = int.Parse(Console.ReadLine());

```
•     }
• }
Console.WriteLine("\n");
```

```
• for (int i = 0; i < 2; i++)  
• {  
•     for (int j = 0; j < 2; j++)  
•     {  
•         Console.Write(x[i][j]);  
•     }  
•     Console.WriteLine();  
• }
```

System.Array class

- In c# every array we create is automatically derived from the System.Array class. This class defines a number of methods and properties that can be used to manipulate array more easily.

Length Number of items in an array

Rank Number of dimensions in an array

IsFixedLength Indicates if an array is of fixed length

IsReadOnly Indicates if an array is read-only

The various methods provided by the System.Array class are:-

- **BinarySearch :-** it search an element in one dimensional array using the binary search algorithm.
- **SetValue:-** It sets the specified value of the items in one dimensional array.
- **Reverse:-** it reverses the order or items in one dimensional array.
- **Copy:-** it copies all the element of an array to another array and performs type casting on that array.
- **Sort:-** it sorts the elements of one dimensional array.

BinarySearch()

```
using System;
```

```
class abc
```

```
{
```

```
    public static void Main()
```

```
{
```

```
    int[] a = { 1, 2, 3, 4, 5, 6 };
```

```
    int n = 5;
```

```
    int val= Array.BinarySearch(a, n);
```

```
    Console.WriteLine("position of a n variable =" +  
        val);
```

```
    Console.Read();
```

```
}
```

```
}
```

Copy()

```
using System;
class abc {
    public static void Main() {
        int[] a = { 1, 2, 3, 4, 5, 6 };
        int []b=new int [6];
        Array .Copy (a,b,6);
        for(int i=0;i<6;i++) {
            Console .WriteLine (b[i]);
        }
        Console.Read();
    }
}
```

Reverse()

```
using System;
class abc{
    public static void Main(){
        int[] a = { 1, 2, 3, 4, 5, 6 };
        Array.Reverse(a);
        for (int i = 0; i < 6; i++)
        {
            Console.WriteLine(a[i]);
        }
        Console.Read();
    }
}
```

SetValue()

```
using System;  
class abc{  
    public static void Main()  
{  
    int[] a = { 1, 2, 3, 4, 5, 6 };  
    a.SetValue(89, 2);  
    for (int i = 0; i < 6; i++)  
    {  
        Console.WriteLine(a[i]);  
    }  
    Console.Read();  
}}
```

Sort()

```
using System;
class abc{
    public static void Main()
    {
        int[] a = {23,45,34,21,45,77 };
        Array.Sort(a);
        for (int i = 0; i < 6; i++)
        {
            Console.WriteLine(a[i]);
        }
        Console.Read();
    }
}
```

ArrayList class

- System.collections namespace defines a class known as arraylist that can store a dynamically sized array of objects. The arraylist class include a number of methods to support operations such as sorting, removing .it also support a property count that gives the number of object in an array list and property capacity to modify or read the capacity.

- using System;
- using System.Collections;
- class abc
- {
- public static void Main()
- {
- ArrayList arr = new ArrayList();
- arr.Add("hello");
- arr.Add("C#");
- arr.Add("World");
- arr.Add(123);
- }

```
• int alen = arr.Count;  
•     for (int x = 0; x < alen; x++)  
•     {  
•         Console.WriteLine( arr[x]);  
•     }  
•     Console.Read();  
• }
```

Special Characters in

Strings

C# provides syntax to embed special characters in your string. Following are the various special characters in the C#:-

- \t :- it helps embed a tab into the string.
- \n :- it helps insert a new line into the string.
- \v :- it helps insert a vertical tab into the string.
- \f :- it helps insert a form-feed character into the string.
- \\ :- It helps specify a backslash character at the current position.

String handling functions

The .NET Framework provides various methods in the System.String class which helps to create new string objects by combining multiple strings, arrays of strings or objects. The commonly used methods for creating a string:-

1. Concat():- is used to concat two strings.

using System;

class abc

{

public static void Main()

{string fn = "Bhagat";

string ln = " Singh";

string name;

name = String.Concat(fn, ln);

Console.WriteLine(name);

Console.Read();

}

2. Join():-It helps to build a new string by combining an array of string.

e.g.

using System;

class abc{

public static void Main() {

**string[] s = { "Hello", "and", "welcome", "to",
"my", "world!" };**

Console.WriteLine (string.Join (" ",s));

Console.ReadLine(); } }

3. Insert():- It helps to build a new string by inserting a string into the specified index of an existing string.

```
using System;
```

```
class abc
```

```
{
```

```
    public static void Main()
```

```
{
```

```
        string s = "object language";
```

```
        Console.WriteLine(s.Insert(6," oriented"));
```

```
        Console .Read();
```

```
}
```

```
}
```

- 4.CopyTo(): It helps copy specified characters in a string into a specified position in an array of character.

```
• using System;  
• class abc  
• {  
•     public static void Main()  
•     {  
•         string a = "MORNING";  
•         char[] b = { 'E', 'V', 'E','N','I','N','G' };  
•         Console.WriteLine("Before copy");
```

```
• foreach (char x in b)
• {
•     Console.WriteLine("After copy");
•     foreach (char x in b)
•     {
•         Console.WriteLine(x);
•     }
•     Console.ReadLine();
• }
```

Trimming and Removing

- We need to remove spaces or new lines from the beginning or ending of a C# string. We use the .NET Framework's Trim method to do this efficiently.
- Methods
- Trim()
- TrimEnd()
- TrimStart()
- Remove()

- Trim(): This method helps to remove the white spaces from both the beginning and ending of the string.
- Example
 - `string st = "This is an example string. ";`
 - `Console.WriteLine("Before Trim = "+st);`
 - `st = st.Trim();`
 - `Console.WriteLine("After Trim = "+st);`

- TrimEnd() :**TrimEnd** removes ending characters from a string.
- Example:
- string st = " This is an example string.!";
- Console.WriteLine("Before TrimEnd = "+st);
- st = st.TrimEnd ('.', '!');
- Console.WriteLine("After TrimEnd = "+st);

- TrimStart(): It helps remove characters specified in an array of characters from the beginning of a string.

- **Example**

- `string st = "\t, Hello Everyone.;"`

`Console.WriteLine("Before TrimStart method= "+st);`

- `st = st.TrimStart('\t','');`

- `Console.WriteLine("After TrimStart method= "+st);`

- Remove(): Remove a specified number of characters from a specified index position in a string.
- Example:
- string test1 = "Hello"; // Five character string.
- string result1 = test1.Remove(3); // Start removing at index 3.
- Console.WriteLine(result1);
- OutPut: Hel

String Padding

- Padding a string adds whitespace or other characters to the beginning or end of the string.
- Method for Padding
 - PadLeft()
 - PadRight()

- PadRight():PadRight adds spaces to the right of strings.
- PadLeft():PadLeft adds spaces to the left of strings.
- Example:

```
•     string s = "Tom".PadRight(4);  
•     string s2 = "Cat".PadLeft(4);  
•     Console.WriteLine("jerry");  
•     Console.WriteLine("Kitten");  
•     Console.WriteLine(s2);
```

String Compare

- **Compare** determines the sort order of strings.
- Methods of string comparison are:-
 1. Compare()
 2. StartWith()
 3. EndWith()
 4. Equals()
 5. IndexOf()
 6. LastIndexOf()

- Compare(): It helps compare the value of two string and returns an integer value.

- Example:

- string a = "hello!";
- string b = "hello?";
- int c = string.Compare(a, b);
- Console.WriteLine("Compare "+c);

- Equals():It helps determine whether two strings are the same and return Boolean value.
- Example:
- `string a = "hello" + 1;`
- `string b = "HELLO" + 1;`
- `Console.WriteLine(string.Equals(a, b));`
- `Console.Read();`

Changing Character Case

- **ToUpperCase()** : It Converts all characters in a string to upper case.
- **ToLowerCase()** : It Converts all characters in a string to lower case.
- Example:
 - `string a = "hello";`
 - `string b = "HELLO";`
 - `Console.WriteLine(a.ToUpper());`
 - `Console.WriteLine(b.ToLower());`

String Parsing

- You can parse strings using the “**split**” method.
- **Split** separates strings.
- The C# language introduces the Split method.
- This method handles splitting upon string and character delimiters.
- Example:

```
string s = "there is a cat";
string[] words = s.Split();
foreach (string word in words)
{
    Console.WriteLine(word); }
```
- **OutPut:** there
- Is
- A
- cat

String Search

- You can perform search operation on a string, which allow you to search a string for a specified number of character.
- Using String Method

Search String Using String Method

- StartWith(): It helps determine whether a string begins with a string passed and return a Boolean value.
- Example:
- ```
string input = "Hello World";
```
- ```
if (input.StartsWith("hello") ||
```



```
input.StartsWith("Hello"))
```
- ```
{
```
- ```
    Console.WriteLine(true);
```
- ```
}
```
- ```
else
```
- ```
{
```
- ```
    Console.WriteLine(false );
```
- ```
}
```
- OutPut:- true

- EndWith(): It helps determine whether a string ends with a string passed and return a Boolean value.
- Example:
  - string input = "Hello World";
  - if (input.EndsWith("World") ||  
input.StartsWith("WORLD"))
    - {
    - Console.WriteLine(true);
    - }
    - else
      - {
      - Console.WriteLine(false );
      - }

- **IndexOf()**: It helps return the index position of a character or string, starting from the beginning of the string for which you are looking and returns an integer value.
- This method finds the first index of the char argument. It returns -1 if the char was not found.
- Example:
  - `string s = " Every day is beautiful";`
  - `int c = s.IndexOf("is");`
  - `Console.WriteLine(c);`
- **OutPut:** 11

- **LastIndexOf()**: It helps return the index position of a character or string, starting from the end of the string for which you are looking and returns an integer value.
- **LastIndexOf** searches strings in reverse.
- This finds the last index of the char argument. It returns -1 if the char was not found.
- Example:
  - `string s = " Every day is beautiful";`
  - `int c = s.LastIndexOf ("is");`
  - `Console.WriteLine(c);`
- OutPut: 11

- SubString(): **Substring** extracts strings. It requires that you indicate a start index and a length. It then returns a completely new string with the characters in that range.
- Substring is ideal for getting parts of strings
- Example:
  - `string input = "OneTwoThree";`
  - `string sub = input.Substring(6, 5);`
  - `Console.WriteLine( sub);`
  - OutPut:-Three

# String Builder Class

- Once created a string cannot be changed. A **StringBuilder** can be changed as many times as necessary.
- The StringBuilder type is used to build up a larger buffer of characters.
- You can call the Append method on the StringBuilder instance to add more and more data.

```
• using System;
• using System.Text;
• class Program
• {
• public static void Main()
• {
• StringBuilder st = new StringBuilder();
• st.Append("hello");
• st.Append("bye");
• st.Append(123);
• Console.WriteLine(st);
• Console.Read();
• }
• }
```

- Remove Method
- You can call the Remove method to remove data from StringBuilder instance.
  - `st.Remove(4, 3);`
- Replace Method
- You can call the Replace method to replace data from StringBuilder instance.
  - `st[5] = 'j';`
  - `Console.WriteLine(st);`
  - `st.Replace("123", "456");`
  - `Console.WriteLine(st);`

# Methods in C#

- Method is a set of statements which perform a specific task. The statements defined in it are executed in the sequence they have been defined in the method. A method takes parameters as input and returns a value when you execute the method. A method is declared in a class or a structure. An example of a method is the main method, which is the entry point of a program.

# Declaring a Method

- A method declaration contains:-
- Method modifier
- Method name
- Return type
- Method body
- Parameter list (optional)

- **Method Modifiers**
- Public
- protected
- private
- internal
- New
- Static

- **Public:-** Signifies that the member is accessible from outside the class's definition and hierarchy of derived classes.
- **Protected :-** The member is not visible outside the class and can be accessed by derived classes only.
- **Private:-** The member cannot be accessed outside the scope of the defining class. Therefore, not even derived classes have access to these members.
- **Internal:-** The member is visible only within the current compilation unit. The *internal* access modifier creates a hybrid of *public* and *protected* accessibility depending on where the code resides
- **New:-** This method hides an inherited method with same signature
- **Static:-** This method does not operate on a specific instance of the class

- Method name:- A method name is declared with a name, which is an identifier for invoking the method.
- Naming rules of method:-
  1. Method name can start with lowercase or uppercase letter.
  2. Underscore character can be used to start method name.

- **Return type :-**The return type of a method defines the type of data that will be returned by the method after execution.
- To return a value from the method, you write the return keyword in the method body to specify the value that should be returned.
- The return keyword is followed by an expression.
- This expression can be a literal value or a variable.

- Parameter List :- The parameter list contains parameters that are passed to a method.
- The use of parameter list in the method is optional.
- To write a method that doesn't take any parameters, write the empty parentheses () .
- For methods with parameters, the parameter list is enclosed in parentheses, written after the method name.
- Each parameter must contain its name and data type.
- Multiple parameters defined in a method are separated by comma.

- **Method Body :-** The method body is a set of statement that define the function performed by the method.
- The method body is written inside the curly braces { }.
- The opening curly brace is written after the signature of the method and closing curly brace is written after the last statement in the method body.

# Types of Method

1. Without return without argument(or parameter)

Public void abc()

2. With return without argument (or parameter)

Public int abc()

3. Without return with argument (or parameter)

Public void abc(int a)

4. With return with argument (or parameter)

Public int abc(int a)

# Types of Parameters

- Value parameter
- Out put parameter
- Reference parameter
- Parameter array

# Value parameters:

- Value parameter is also called In parameter.
- A parameter declared with no modifiers is a value parameter.
- A value parameter corresponds to a local variable that gets its original value from the corresponding argument supplied in the method invocation.
- A method is permitted to assign new values to a value parameter.
- Such assignments only affect the local storage location represented by the value parameter-they have no effect on the actual argument given in the method.

```
• using System;
• class program
• {
• public static void abc(int x)
• {
• x = x + 5;
• Console.WriteLine("value of X is " + x);
• }
• public static void Main()
• {
• int a = 10;
• abc(a);
• Console.WriteLine("value of y is " + a);
• Console.Read(); } }
```

# Reference parameters:

- A parameter declared with a **ref modifier** is a reference parameter.
- Like value parameter, a reference parameter does not make a new storage location.
- Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

```
• using System;
• class program
• {
• public static void abc(ref int x)
• {
• x = x + 5;
• }
• public static void Main()
• {
• int a = 90;
• Console.WriteLine("value of y without reference " +
• a);
• abc(ref a);
• Console.WriteLine("value of y with reference " + a);
• Console.Read(); } }
```

# Output parameters:

- A parameter declared with an **out modifier** is an output parameter.
- Similar to a reference parameter, an output parameter does not create a new storage location.
- Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.
- Every output parameter of a method must be definitely assigned before the method returns.

```
• using System;
• class program
• {
• public static void abc(out int y)
• {
• y = 100;
• Console.WriteLine("y is " + y);
• }
• public static void Main()
• {
• int a = 5;
• abc(out a);
• Console.WriteLine("value of y is " + a);
• Console.Read(); } }
```

# Parameter arrays:

- A parameter declared with a **params modifier** is a parameter array.
- If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type.
- For example, the types `string[]` and `string[][]` can be used as the type of a parameter array, but the type `string[,]` can not. It is not possible to combine the **params** modifier with the **ref** and **out** modifiers.

```
• using System;
• class pro
• {
• public static void abc(params int [] Y)
• {
• Console.WriteLine("result is ");
• foreach (int x in Y)
• Console.Write(""+x);
• Console.ReadLine();
• }
• public static void Main()
• {
• int[] tt ={ 88, 99, 77 };
• abc(tt); } }
```

# What is the difference between out and ref in C#?

- 1) out parameters return compiler error if they are not assigned a value in the method. Not such with ref parameters.
- 2) out parameters need not be initialized before passing to the method, whereas ref parameters need to have an initial value before they are passed to a method.

# Method overloading

- Method overloading means having two or more methods with the same name but different signatures in the same scope. These two methods may exist in the same class or another one in base class and another in derived class.

- using System;
- class fan
- {
- public int fancode(int fc)
- {
- Console.WriteLine("Fan name is Khatan");
- return fc;
- }
- public double fancode(double fc)
- {
- Console.WriteLine("Fan name is USHA");
- return fc;
- } }

- class fanname
- {
- public static void Main()
- {
- fan f = new fan(); // create object of class fan
- int x =1;
- double y=1.5;
- //method name fancode is same but result differs if signature changes
- f.fancode(x);
- f.fancode(y);
- Console.Read();
- }
- }

# Structure and Enumeration

- C# supports two kind of value type namely
  1. Predefined types
  2. User defined types
    - We have defined and use predefined data type such as int and double .C# allows us to define our own complex value types (user-defined data types) these data types are
      1. Structures
      2. Enumerations

# Structure

- A structure in C# is simply a composite data type consisting of number elements of other types.
- The structure in C# can contain fields, methods, constants, constructors, properties, indexers, operators and even other structure types.
- Structures are similar to classes in C#.

# Structure Declaration & Object Creation

- The keyword struct can be used to declare a structure.

Syntax:-

```
<modifiers> struct <struct_name>
```

```
{
```

```
//Structure members
```

```
}
```

- Where the modifier can be private, public, internal or public. The struct is the required keyword.

- Example:

```
• struct book
 • {
 • public int bookcost;
 • public string book_title;;
 • }
```

```
• using System;
• {
• class Program
• {
• struct comdata //struct declare
• {
• public string name;
• public string location;
• }
• class display
• {
• public static void Main()
• {
```

- comdata cdetails; // use struct
- cdetails.name = "HCL";
- cdetails.location = "NOIDA";
- //Display the result
- Console.WriteLine("company name is " +  
cdetails.name);
- Console.WriteLine("company location is " +  
cdetails.location);
- Console.Read();
- } } } }

# Struct & Methods

- A C# struct can also contain methods. The methods can be either static or non-static. But static methods can access only other static members and they can't invoke by using an object of the structure. They can invoke only by using the struct name.

- **Example of Structure using Method**

- using System;
- class Program
- {
- struct area
- {
- public double areaofcircle(int x)

```
• {
• double a;
• a = 3.14 * x * x;
• return a;
• }
• public double areasquare(int side)
• {
• double aa;
• aa = side * side;
• return aa;
• } }
```

```
• class display
• {
• public static void Main()
• {
• area kite;
• double ans1= kite.areaofcircle(10);
• double ans2= kite.areasquare(25);
• Console.WriteLine("Area of Circle:"+ans1);
• Console.WriteLine("Area of Square:" + ans2);
• Console.Read();
• }
• }
```

- **Nested structs (struct in struct)**
- using System;
- struct emp
- {
- public string name;
- public string des;
- public struct manager
- {
- public double sal;
- public double tada;
- }

- public struct clerk
- {
- public double sal;
- public double tada;
- }
- }
- class display
- {
- public static void Main()
- {
- emp customer;
- customer.name="Harman";
- customer.des ="manager";

- emp.manager man;
- emp.clerk cle;
- man.sal=9000;
- man.tada=2500;
- cle.sal = 6000;
- cle.tada = 1500;
- Console.WriteLine (" name is "+customer.name +  
" "+ "designation " +customer.des + " "+  
"salary "+ (man.sal+man.tada) );
- Console.Read () ;
- }
- }

# Enumeration

- An enumerated type has values which are different from each other, and which can be compared and assigned, but which do not have any particular concrete(existing) representation in the computer's memory; compilers and interpreters can represent them arbitrarily(randomly).
- The **enum** keyword is used to declare an enumeration.

- **Example of Enumeration**
- using System;
- class Program
- {
- enum days
- {
- sunday, //Position 0
- monday, //Position 1
- tuesday, //Position 2
- wednesday, //Position 3
- thursday, //Position 4
- friday, //Position 5
- saturday, //Position 6
- }

- public static void Main()
- {
- days x; //Declare type days
- x = days.sunday;
- Console.WriteLine(x);
- Console.Read();
- }
- }

# Constructors

- *One of the biggest advantages of an OOP language such as C# is that you can define special methods that are always called whenever an instance of the class is created. These methods are called constructors. A constructor in c# is a special method that shares the same name of its class.*
- *Constructor is used to initialize an object (instance) of a class.*
- *Constructor is like a method without any return type.*
- *Constructor has same name as class name.*
- *Constructor can be overloaded.*

# **Constructors generally following types**

- Default Constructor
- Parameterized constructor
- Private Constructor
- Static Constructor
- Copy Constructor
- Constructor overloading

# Default Constructor

- A constructor that takes no parameters is called a default constructor.
- When a class is initiated default constructor is called which provides default values to different data members of the class.

- using System;
- class Program
- {
- class c1
- {
- int a, b;
- public c1()
- {
- this.a = 10;
- this.b = 20;
- }
- public void display()

```
• { Console.WriteLine("Value of a", a);
• Console.WriteLine("Value of b ", b);
• }
• }
public static void Main()
{
 // Here when you create instance of the class
 //default constructor will be called.
 c1 ob1 = new c1();
 ob1.display();
 Console.ReadLine();
}
}
```

# Parameterized constructor

- Constructor that accepts arguments is known as parameterized constructor.
- There may be situations, where it is necessary to initialize various data members of different objects with different values when they are created.
- Parameterized constructors help in doing that task.

- using System;
- class Program
- {
- class c1
- {
- int a, b;
- public c1(int x, int y)
- {
- this.a = x;
- this.b = y;
- }
- public void display()
- {

- Console.WriteLine("Value of a", a);
  - Console.WriteLine("Value of b", b);
  - }
  - }
- class main
  - {
  - public static void Main()
  - {
- // Here when you create instance of the class
  - //parameterized constructor will be called.
  - c1 ob1 = new c1(10, 20);
  - ob1.display();
  - Console.ReadLine();
  - } }

# Private Constructor

- Private constructors are used to restrict the instantiation of object using 'new' operator.
- A private constructor is a special instance constructor.
- It is commonly used in classes that contain static members only.
- If you don't want the class to be inherited we declare its constructor private.
- We can't initialize the class outside the class or the instance of class can't be created outside if its constructor is declared private.

```
• using System;
• class Program
• {
• class c1
• {
• int a, b;
• // Private constructor declared here
• private c1(int x, int y)
• {
• this.a = x;
• this.b = y;
• }
• public static c1 create_instance()
• {
```

```
• return new c1(12, 20);
• }
• public void display()
• {
• int z = a + b;
• Console.WriteLine(z);
• } }
• public static void Main()
• {
• c1 ob1 = c1.create_instance();
• ob1.display();
• Console.ReadLine();
• } }
```

# Static constructors

- Static constructors are executed only once.
  - *Remember: a static constructor is invoke automatically when a class is loaded into memory.*
1. There can be only one static constructor in the class.
  2. The static constructor should be without parameters.
  3. It can only access the static members of the class.
  4. There should be no access modifier in static constructor definition

- using System;
- class Program
- {
- public class test
- {
- static string name;
- static int age;
- static test()
- {

Console.WriteLine("Using static constructor to initialize static data  
members");

- name = "John cena";
- age = 23;
- }

```
• public static void display()
• {
• • Console.WriteLine("Using static function");
• • Console.WriteLine(name);
• • Console.WriteLine(age);
• }
• }
• public static void Main()
• {
• test.display();
• Console.ReadLine();
• }
• }
```

# Copy Constructor

- If you create a new object and want to copy the values from an existing object, you use copy constructor.
- This constructor takes a single argument: a reference to the object to be copied.

- using System;
- class Program
- {
- class c1
- {
- int a, b;
- public c1(int x, int y)
- {
- this.a = x;
- this.b = y;
- }
- // Copy constructor
- }

```
• public c1(c1 a)
• {
• this.a = a.a;
• this.b = a.b;
• }
• public void display()
• {
• int z = a + b;
• Console.WriteLine(z);
• }
• }
```

- public static void Main()
- {
- c1 ob1 = new c1(10, 20);
- ob1.display();
- // Here we are using copy constructor. Copy  
constructor //is using the values already defined  
with ob1
- c1 ob2 = new c1(ob1);
- ob2.display();
- Console.ReadLine();
- }
- }

# Constructors overloading

- Constructor is overloaded when two methods have same name and both name is same as class name

```
• using System;
• class myClass
• {
• public int iCounter, iTotal;
• public myClass()
• {
• iCounter = 0;
• iTotal = 0;
• }
• public myClass(int iCount, int iTot)
• {
• iCounter = iCount;
• iTotal = iTot;
• } }
```

```
• class TestmyClass
• {
• public static void Main()
• {
• myClass cls = new myClass();
• myClass cls1 = new myClass(3, 4);
• Console.WriteLine(cls1.iCounter);
• Console.WriteLine(cls1.iTotal.);
• }
• }
```

# Destructors

- A destructor is called when it's time to destroy the object. Destructors can't take parameters
- **class myClass**
- {
- **~myClass()**
- {
- **// free resources**
- }
- }

# Inheritance

- The mechanism (method) of designing or constructing one class from another is called Inheritance.
- The concept behind inheritance, take the base class that contains methods and properties and inherits it into the derived class, that now contains all the members and properties of the base class as well as its own newly added members or properties.
- Implementing inheritance in C# is similar to implementing it in C++. You use a colon (:) in the definition of a class to derive it from another class.



Class A – the initial class that is used as the Base for the derived class is referred to as *Base Class , Parent Class , Super Class*

Class B – the derived class that is used Functionality of Base Class it is also called Child class or sub class

# **Advantages of Inheritance**

1. Reuse the existence code
2. Save time, we need not do the code verification and testing.
3. Here, we can add and modify the methods of the existing class.
4. Help in modularization of code.

# Types of Inheritance

1. Single inheritance
  2. Multilevel inheritance
  3. Hierarchical Inheritance
- C# doesn't support multiple Inheritances.

# Single Inheritance



Single Inheritance ( Only one base Class)

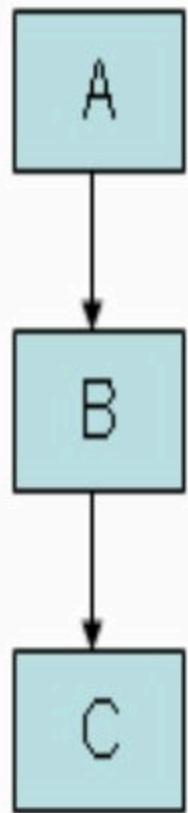
# Single Inheritance

- It has only one base class that is inherited by the derived class. Here the derived class has all the features of the base class and can add new features or modify existing features.
- The inheritance also depends on the access specifier that is used at the time of base class inheritance.

- using System;
- class c //base class
- {
- public string a = "printf";
- public string b = "scanf";
- }
- class c\_plus : c //drived class
- {
- public string c = "cout";
- public string d = "cin";
- }

```
• class main
• {
• public static void Main()
• {
• c_plus obj = new c_plus();
• Console.WriteLine (obj.a);
• Console.WriteLine (obj.b);
• Console.WriteLine (obj.c);
• Console.WriteLine (obj.d);
• Console.Read();
• }
• }
```

# Multi-level Inheritance



Multilevel Inheritance ( several base classes )

# Multi-level Inheritance

- In the multi-level inheritance, here we having a chain of inheritance i.e. the base class A is inherited by derived class B, and it is further inherited by another derived class C. So, the feature we have in base class A, is also there in Derived class B, and all the combination features of class A and class B are there in derived class C.

- using System;
- class c //base class
- {
- public string a = "printf";
- public string b = "scanf";
- }
- class c\_plus : c //drived class
- {
- public string c = "cout";
- public string d = "cin";
- }

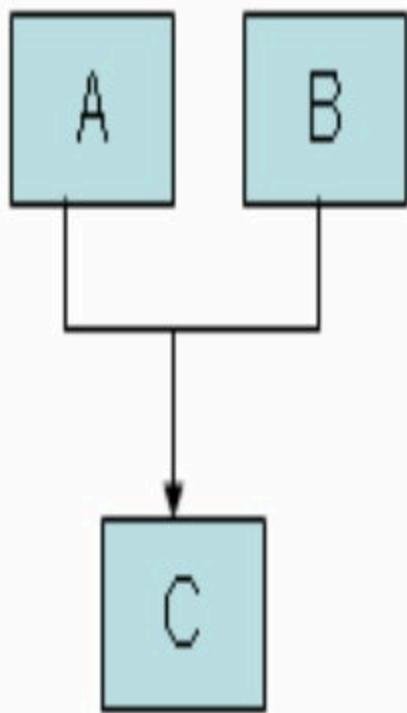
```
• class c_sharp : c_plus
• {
• public string e = "Console.WriteLine";
• public string f = "Console.ReadLine";
• }

• class main
• {
• public static void Main()
• {
• c_sharp obj = new c_sharp();
• }
• }
```

```
• Console.WriteLine (obj.a);
 • Console.WriteLine (obj.b);
 • Console.WriteLine (obj.c);
 • Console.WriteLine (obj.d);
 • Console.WriteLine(obj.e);
 • Console.WriteLine(obj.f);

 • Console.Read();
 • }
 • }
```

# Multiple Inheritance

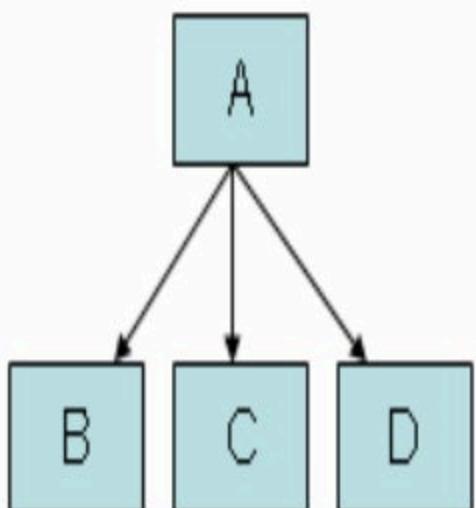


Multiple Inheritance (several base Class)

# Multiple Inheritance

- In C#, we **don't** have multiple inheritance where one class can inherit two base classes.
- C# doesn't support multiple Inheritances.
- So, the multiple inheritance can be done by using the concept of **interfaces**. Here one class can inherit one class and can implements more than one interface.

# Hierarchical Inheritance



Hierarchical Inheritance  
( one base class many derived classes )

- using System;
- class c //base class
- {
- public string a = "printf";
- public string b = "scanf";
- }
- class c\_plus : c //drived class
- {
- public string c = "cout";
- public string d = "cin";
- }
- class c\_sharp : c
- {

```
• public string e = "Console.WriteLine";
• public string f = "Console.ReadLine";
• }
• class main
• {
• public static void Main()
• {
• c_sharp obj = new c_sharp0 ;
• Console.WriteLine(obj.a);
• Console.WriteLine(obj.b);
• Console.WriteLine(obj.e);
• Console.WriteLine(obj.f);
• Console.Read();
• }
• }
```

# Method overloading using inheritance

- using System;
- class a //base class
- {
- public int sum(int n1, int n2)
- {
- int s;
- s = n1 + n2;
- return s;
- } }

- class b : a //drived class
- {
- public int sum(int n1, int n2, int n3)
- {
- int s;
- s = n1 + n2 + n3;
- return s;
- }
- }

- public static void Main()
- {
- b obj = new b();
- int x;
- x = obj.sum(10, 10);
- int y;
- y = obj.sum(10, 10, 10);
- //here we see method name sum is same but give different
  - // result when method parameter changes
- Console.WriteLine("sum =" + x);
- Console.WriteLine("\nsum =" + y);
- Console.Read();
- }
- }

# Method overriding using inheritance

- using System;
- class a //base class
- {
- public void cname()
- {
- Console.WriteLine("This is first  
method");
- }
- }

```
• class b : a //drived class
• {
• new public void cname()
• // new keyword is used to hiding method cname of
class a
• {
• Console.WriteLine("This is second method");
• }
• public static void Main()
• {
• b obj = new b();
• b.cname();
• Console.Read();
• }
```

# **Sealed Class**

- In order to prevent a **class** in C# from being inherited, the **sealed** keyword is used.
- Thus a sealed class may not serve as a base class of any other class.
- It is also obvious that a sealed class cannot be an abstract class. Code below...
- sealed class ClassA
  - {
  - public int x;
  - public int y;
  - }

# Sealed Method

- If a method is not to be inherited, but the class is, then the method is sealed.
- It becomes a **sealed method** of a class.
- It is important to note here that in C#, a method may not be implicitly declared as sealed.
- This means that a method cannot be sealed directly.
- A method in C# can be sealed only when the method is an overridden method.
- Once the overridden method is declared as sealed, it will not be further overriding of this method.

- See code sample below, where an overridden method is sealed.
- using System;
- Class first
- {
- public int a;
- public int b;
- public virtual void show()
- {
- Console.WriteLine("This is a virtual method");
- }
- }

```
• class second: first
• {
• public override sealed void show()
• {
• Console.WriteLine("This is a sealed method");
• }
• }
```

- class program
- {
- public static void Main()
- {
- second obj = new second();
- obj.show();
- Console.Read();
- }
- }

# **Virtual Method**

- If a function or a property in the base class is declared as virtual it can be overridden in any derived classes
- This is useful because the compiler verifies that the ‘override’ function has the same signature as the virtual function
- **Hiding Methods**
- Similar to the above scenario if the methods are declared in a child and base class with the same signature but without the key words virtual and override, the child class function is said to hide the base class function

- In the example above the function fnAge in grandChildClass hides the function fnAge in its parent class i.e. abcClass
- The C# compiler will generate a warning in this case. The **new** keyword should be used when we intend to hide a method.

# What is Polymorphism

- *Polymorphism is the core concept of OOP's. Polymorphism means one name many forms.*
- Polymorphism in OOP means '*same name different functionality*'.
- What this means in action is that we have a same name for different functionalities.
- For example, we have the same name – ‘Sleep’ which we can use for making each animal sleep, be it tiger or lion or cat or anything.

- What I mean is that the method for sleeping is different in each of these cases/animals... Cat sleeps differently than a tiger and so on, but we still use 'Sleep' to make all animals sleep.
- This means that we have different functionalities but having the same name that is Sleep.
- This is Polymorphism (many + forms).
- **Types of polymorphism**
- 1. Compile Time Polymorphism
- 2. Run Time Polymorphism

- **Compile Time Polymorphism**
- The compile time polymorphism, here the polymorphism is implemented during compile time, that means at the time of compilation the compiler knows where to bind the method.
- The compile time polymorphism can be implemented through:
  - Method Overloading
  - Operator Overloading

# Run Time Polymorphism

- The run time polymorphism, here the compiler doesn't know which method to call at runtime.
- Here we can call the derived method through base class pointer at runtime.
- The run time polymorphism can be implemented through: Virtual – Override Keyword.
- **Define Overloading and Overriding**

# **Overloading**

- Whenever we introduce/write a new functionality with the same name as an existing functionality but changing the signature, we are implementing Overloading.
- In overloading we can differentiate between the two functionalities by their signatures.
- And the call made to that functionality depends on the context.
- This way both the functionalities can co-exist.

# ***Overriding***

- Whenever we write a new functionality with the name and signature both same, the existing functionality gets *overridden* by the new functionality hence can't be used.
- When a call to the functionality is made, only the new functionality is called.
- This is called *Overriding*.

# Interface

- Interfaces basically define a blueprint for a class. The programmed definition of an interface looks very similar to a class, but nothing is implemented. Interfaces define the properties, methods, events, and indexers, but the interface does not define the implementation of any of these. It just declares their existence. Interfaces will not actually define any functionality. They just define ways in which interactions with a class takes place.

- using System;
- interface addition
- {
- int add(int x, int y);
- }
- interface multiplication
- {
- int mul(int x, int y);
- }
- class xyz : addition, multiplication
- {
- }

```
• public int sub(int x, int y)
• {
• int s;
• s = x - y;
• return s;
• }
• public int mul(int x, int y)
• {
• int s;
• s = x * y;
• return s;
• }
```

```
• public int add(int x, int y)
• {
• int s;
• s = x + y;
• return s;
• }
• public static void Main()
• {
• int x, y;
• x = 20;
• y = 10;
• xyx obj = new xyx();
```

- Console.WriteLine("sum " + obj.add(x, y));
- Console.WriteLine("mul " + obj.mul(x, y));
- Console.WriteLine("sub " + obj.sub(x, y));
- Console.Read();
- }
- }

- using System;
- interface inter
- {
- void add();
- }
- interface inter1
- {
- void add();
- }
- class abc:inter,inter1
- {
- int a,b,c;
- }

```
• void inter.add()
• {
• a=10;
• b=20;
• c=a+b;
• Console.WriteLine(c);
• }
• void inter1.add()
• {
• a=20;
• b=30;
• c=a+b;
• Console.WriteLine(c);
• }
```

```
• public static void Main()
• {
• abc obj=new abc();
• inter c=(inter)obj;
• c.add();
• inter1 d=(inter1)obj;
• d.add();
•
• Console.Read();
• }
• }
```

- **Abstract Classes and Interfaces**
- Some classes are not designed to have direct instances. Rather, they are designed simply to be inherited from, by relatives which may themselves have direct instances (or not). A class is 'abstract' just in case it cannot itself have direct instances.
- Classes can be abstract because in a class it is possible to specify a class method without specifying its body. Such methods are themselves termed 'abstract'. Where a class contains an abstract method it cannot be instantiated, since it is not specified what should happen were the method to be called.

- An 'interface' is a class which has only abstract methods. However, such a class is declared not with the 'class' keyword but the 'interface' keyword.

# Difference between abstract classes and interfaces

- 1. Abstract classes can have concrete methods while interfaces have no methods implemented.
- 2. Interfaces do not come in inheriting chain, while abstract classes come in inheriting chain.
- 3. Interfaces have only pure virtual abstract method, while abstract classes may have non abstract methods.
- 4. All members are public [by default] in interfaces but you have to declare public members in abstract classes

- using System;
- abstract class one
- {
- abstract public void add(int a,int b);
- }
- class two : one
- {
- public override void add(int a,int b)
- {
- int c;
- c=a+b;
- Console.WriteLine (c);
- }
- }

```
• class main
• {
• public static void Main()
• {
• two obj = new two();
• obj.add(10,40);
• Console.Read();
• }
• }
```

# Operators Overloading

- *Operator overloading is a concept that enables us to redefine the existing operators so that they can be used on user defined types like classes.*
- *Remember that it is not possible to overload all operators in C#. The following table shows the operators and their overload ability in C#.*

# Why we use operator overloading?

- If you have a type that will be used in thousands of places then overloading it could be good idea if those overloads move intuitive sense.

**Operator can be  
overloaded**

- `+, -, !, or ~` must take a parameter of the defining type and can return any type
- `++` or `-` must take and return the defining type
- `true` or `false` must take a parameter of the defining type and can return a `bool`

# **Operator can't be overloaded**

- &&
- () (Conversion operator)
- = -> new , is , as , sizeof

# **Important point to remember about operator overloading**

1. Operator function is used for overloading purpose.
2. These method/function must be public and static.
3. They can take only value arguments.
4. ref and out parameters are not allowed.
5. Unary operators take **only one** argument
6. Binary operator takes **only two** arguments
7. Remember –one of the arguments must be a user – defined type (such as class or struct type)
8. Operator overload methods can't return void.

## **9. public static return\_type operator op(argument list)**

- {
- //statements
- }
- Op              Operator to be overloaded.(like  
                  +,-,++,--,== etc)
- return\_type    Must be class or struct(can't  
                  be void).

# Binary operator overloading

```
• using System;
• class opload
• {
• int a,b;
• public void setdata(int x, int y)
• {
• a = x;
• b = y;
• }
• public void show()
• {
• Console.WriteLine("a = "+a);
• Console.WriteLine("b = "+b);
• }
• }
```

- public static opload operator +(opload x,  
opload y)
- {
- opload obj = new opload();
- obj.a = x.a + y.a;
- obj.b = x.b + y.b;
- return obj;
- }
- }

```
• class main
• {
• public static void Main()
• {
• opload x = new opload();
• opload y = new opload();
• opload obj = new opload();
• x.setdata(2, 3);
• y.setdata(4,5);
• obj = x + y;
• obj.show();
• Console.Read();
• }
• }
```

# Unary operator overloading

- using System;
- class test
- {
- int a;
- public void setdata()
- {
- a = 10;
- }
- public void output()
- {
- Console.WriteLine(a);
- }

```
• public static test operator ++(test x)
• {
• x.a = x.a + 10;
• return (x);
• }
• class Program
• {
• public static void Main()
• {
• test x = new test();
• x.setdata();
• x++;
• x.output();
• Console.Read(); } }
```

# **Overloading True and False operator**

- The syntax for overloading the true and false operators is similar to that of other unary operators.
- Two limitations exist.
  1. Firstly, the return value must be a Boolean.
  2. Secondly, it is invalid to overload only one of the two operators; if the true operator is overloaded than so must be false and vice versa.

# Error Handling and Exceptions

# Exception handling

Exception handling is an in built mechanism in .NET framework to detect and handle run time errors. The .NET framework contains lots of standard exceptions. The exceptions are anomalies that occur during the execution of a program. They can be because of user, logic or system errors. If a user (programmer) do not provide a mechanism to handle these anomalies, the .NET run time environment provide a default mechanism, which terminates the program execution.

- C# provides three keywords **try, catch and finally** to do exception handling. The try encloses the statements that might throw an exception whereas catch handles an exception if one exists. The finally can be used for doing any clean up process.

# Syntax

- **try**

```
{
// Statement which can cause an exception.
}
```

- **catch(Type x)**

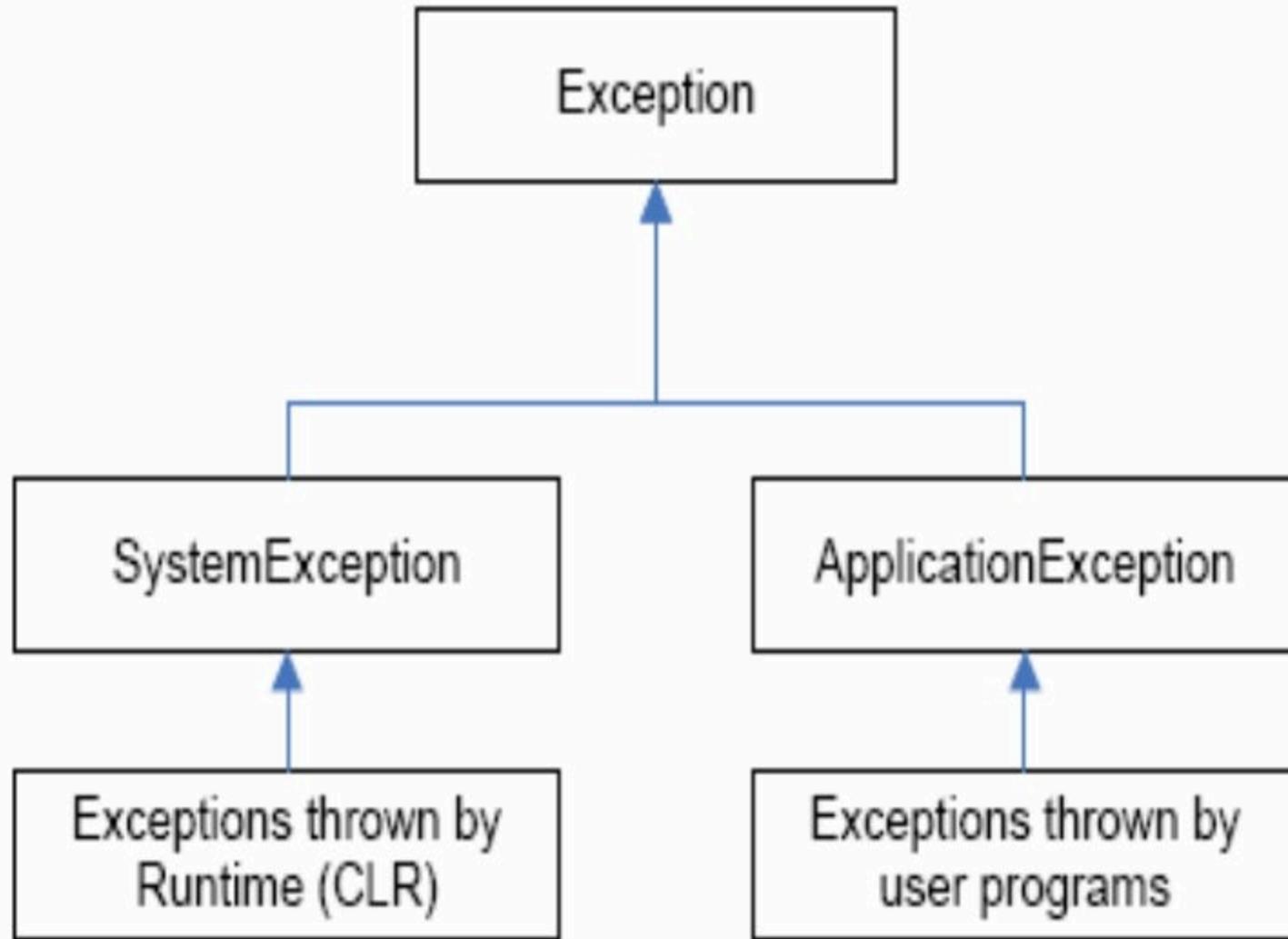
```
{
// Statements for handling the exception
}
```

- **finally**

```
{
//Any cleanup code
}
```

```
• using System;
• class abc
• {
• public static void Main()
• {
• try
• {
• int x, y, z;
• x = 25;
• y = 0;
• z = x / y;
• Console.WriteLine("ans is " + z);
• Console.Read();
• }
• }
• }
```

- catch (Exception ex)
- {
- Console.WriteLine(ex.Message);
- Console.Read();
- }}}



*Exceptions Hierarchy in the .NET framework*

# **Standard Exceptions**

- **There are two types of exceptions:**
  1. System.Exception
  2. The ApplicationException

## **System.Exception**

System.Exception is the base class for all exceptions

- In C#. Several exception classes inherit from this class including ApplicationException and SystemException.
- These two classes form the basis for most other runtime exceptions.
- Other exceptions that derive directly from System.Exception include IOException, WebException etc.

- The common language runtime throws SystemException.
- The ApplicationException is thrown by a user program rather than the runtime.
- The SystemException includes the ExecutionEngineException, StackOverflowException etc.
- It is not recommended that we catch SystemExceptions nor is it good programming practice to throw

# User-defined Exceptions

- In C#, it is possible to create our own exception class. But Exception must be the ultimate base class for all exceptions in C#.
- So the user-defined exception classes must inherit from either Exception class or one of its standard derived classes.

- using System;
- class myexce : Exception
- {
- public myexce(String n)
- {
- Console.WriteLine(n + " Your age is less than 10");
- }
- }
- class Program
- {
- public static void Main(string[] args)
- {

```
• String name;
• int age;
• try
• {
• Console.WriteLine("Enter name ");
• name = Console.ReadLine();
• Console.WriteLine("Enter age ");
• age = int.Parse(Console.ReadLine());
• if (age < 10)
• throw new myexce(name);
• }
```

```
• catch (Exception e)
 {
 • Console.WriteLine("Error occur");
 • }
 • Console.Read();
 }
}
```

# Throwing an Exception

- In C#, it is possible to throw an exception programmatically. The '**throw**' keyword is
- used for this purpose. The general form of throwing an exception is as follows.
- **throw exception\_obj;**

```
• using System;
• class ABC
• {
• public static void Main()
• {
• try
• {
• throw new DivideByZeroException("Invalid
• Division");
• }
• catch (DivideByZeroException e)
• {
• Console.WriteLine("Exception");
• }
• }
• }
```

- Console.WriteLine("LAST STATEMENT");
- Console.Read();
- }
- }

# Multiple Catch Blocks

- A try block can throw multiple exceptions, which can handle by using multiple catch blocks.
- Remember that more specialized catch block should come before a generalized one.
- Otherwise the compiler will show a compilation error.

- using System;
- class ABC
- {
- public static void Main()
- {
- int x = 0;
- int div = 0;
- try
- {
- div = 100 / x;
- Console.WriteLine("Not executed line");
- }
- }

```
• catch (DivideByZeroException de)
• {
• Console.WriteLine("DivideByZeroException");
• }
• catch (Exception X)
• {
• Console.WriteLine("Exception");
• }
• finally
• {
• Console.WriteLine("Finally Block");
• }
• Console.WriteLine("Result is {0}", div);
• Console.Read(); } }
```

- using System;
- class abc
- {
- public static void Main()
- {
- int[] a = { 6, 0 };
- try
- {
- int x = a[0] / a[1]; // divide by zero
- //int x = a[0] / a[11]; // index was out of bound
- Console.WriteLine("ans is " + x);
- Console.Read();
- }

```
• catch (ArithmeticException ex)
• {
• Console.WriteLine(ex.Message);
• Console.Read();
• }
• catch (IndexOutOfRangeException ex)
• {
• Console.WriteLine(ex.Message);
//Console.WriteLine("array value is more than
asign");
• Console.Read();
• }
```

```
• catch (ArrayTypeMismatchException ex)
• {
• Console.WriteLine(ex.Message);
• Console.Read();
• }
• catch (Exception ex)
• {
• Console.WriteLine(ex.Message);
• Console.Read();
• } } }
```

# Introduction to Objects and Classes

# Introduction to Objects and Classes

- In our world we have classes and objects for those classes.
- Everything in our world is considered to be an object.
- For example, people are objects, animals are objects too, minerals are objects; everything in the world is an object.

- **But what about classes?**
- In our world we have to differentiate between objects that we are living with.
- So we must understand that there are classifications (this is how they get the name and the concepts of the Class) for all of those objects.
- For example, I'm an object, Ram is object too, Sunita is another object. So we are from a people class (or type).
- I have a dog called Tiger so it's an object. My friend's dog, Niki, is also an object so they are from a Dogs class (or type).
- In our world we have classifications for objects and every object must be from some classification.

- So, a Class is a way for describing some properties and functionalities or behaviors of a group of objects.
- In other words, the class is considered to be a template for some objects.

# What is Class

- A class is basically a blueprint for a custom data type. Once you define a class, you use it by loading it into memory. A class that has been loaded into memory is called an object or an instance. You create an instance of a class by using the C# keyword **new**.

```
• using System;
• class SampleClass
• {
• public void city()
• {
• Console.WriteLine("i live in Patiala City");
• }
• }
• class Program
• {
• public static void Main()
• {
• SampleClass obj = new SampleClass(); obj.city();
• Console.Read();
• }
• }
```

# Class Members

- Methods Similar to C++ functions. Methods implement some action that can be performed by an object.
- Properties Provide access to a class attribute (a field). Useful for exposing fields in components.
- Events Used to provide notification.
- Constants Represents a constant value.
- Fields Represents a variable of the class
- Operators Used to define an expression (+, \*, ->, ++, [], and so on ).
- **Instance, Constructors, Methods** are called during initialization of an object.

- Static Constructors Called automatically.
- Destructors Called when an object is being destroyed.
- Types--All local types used in a class.

# Properties

- Important set of members of a class is variables. A variable is a type that stores some value.
- The property member of a class provides access to variables. Some examples of Properties are font type, color, and visible properties. Basically, Properties are fields.
- A field member can be accessed directly, but a property member is always accessed through access or and modifier methods called get and set, respectively.
- If you have ever created active X controls in C++ or visual basic, or created JavaBeans in java, you understand.

# Why use properties

- if you already have the field available?  
First of all, properties expose
- fields in classes being used in components. They also provide a means for doing necessary

- Class property member example using System;
- using System;
- class myClass
- {
- private int Age;
- // Age property
- public int iAge
- {
- get
- {
- return Age;
- }
- }

```
• set
• {
• Age = value;
• } } }
• class TestmyClass
• {
• static void Main()
• {
• myClass cls = new myClass();
• // set properties values
• cls.iAge = 25;
• Console.WriteLine("Age is " + cls.iAge);
• Console.Read();
• }
• }
```

```
• using System;
• class person
• {
• private string color; // Color property
• public string icolor
• {
• get
• {
• return color;
• }
• set
• {
• color = value;
• } } }
```

```
• class TestmyClass
• {
• static void Main()
• {
• person obj = new person ();
• // set properties values
• obj.icolor = "Black";
• Console.WriteLine("Eye color is"+obj.icolor);
• Console.Read();
• } }
```

# Delegates

*Delegates in C# are objects which points towards a function which matches its signature. Delegates are reference type used to encapsulate a method with a specific signature.*

# What is Delegate

- Meaning of Delegate is “a person acting for another person”
- In c# it really means method acting for another method.
- Delegates are mainly used with the class events.
- A delegate type Encapsulates (use) a method with a certain signature, called a callable entity.
- Delegates are the type safe and secure version of function pointers (callback functionality).
- Delegate instances are not aware of the methods they encapsulate; they’re aware only and return type.

# Where are Delegates used

- The most common example of using delegates is in events.
- You define a method that contains code for performing various tasks when an event (such as a mouse click) takes place.
- This method needs to be invoked by the runtime when the event occurs.
- Hence this method, that you defined, is passed as a parameter to a delegate.

# **Write steps to invoke delegates**

- Creating and using delegates involve four steps.
- 1. Delegate declaration
- 2. Delegate methods definition
- 3. Delegate instantiation
- 4. Delegate invocation

- A **delegate declaration** defines a class using the class System. Delegate as a base class.
- **Delegate method** is any function whose signature matches the delegate signature exactly.
- The **delegate instance** holds the reference to delicate methods.
- The instance is used to **invoke the method** indirectly.

# There are three steps in defining and using a delegate:

- *Declaration syntax.*
- **delegate void MyDelegate();**
- *Declares a delegate named*
- **MyDelegate** that no arguments and returns void.
- The next step is to *create an instance of delegate* and call it:
  - **MyDelegate del =new MyDelegate(TestMethod);**
  - **del();**

```
• using System;
• delegate void MyDelegate();
• class Test
• {
• static void TestMethod()
• {
• Console.WriteLine("Test Method called");
• }
• public static void Main()
• {
• MyDelegate del = new MyDelegate(TestMethod);
• del();
• Console.Read();
• }
• }
```

```
• using System;
• delegate void abc();
• class Test
• {
• static void book()
• {
• System.Console.WriteLine(" this book is very
• easy");
• }
• static void color()
• {
• System.Console.WriteLine("my best color is
• blue");
• }
• }
```

- static void film()
- {
- System.Console.WriteLine("story of some films are same");
- }
- public static void Main()
- {
- abc xyz = new abc(film);
- xyz();
- Console.Read();
- }
- }

# What is Multicast Delegates

- MultiCast Delegates are nothing but a single delegate that can invoke multiple methods of matching signature.
- MultiCast Delegate derives from System.MulticastDelegate class which is a subclass of System.Delegate.

- using System;
- delegate void Delegate\_Multicast();
- class Class2
- {
- static void Method1()
- {
- Console.WriteLine("You r in Method 1");
- }
- static void Method2()
- {
- Console.WriteLine("You r in Method 2");
- }

- public static void Main()
- {
- Delegate\_Multicast func = new  
        Delegate\_Multicast(Method1);
- func += new Delegate\_Multicast(Method2);
- func(); // Method1 and Method2 are called
- func -= new Delegate\_Multicast(Method1);
- func(); // Only Method2 is called
- Console.Read();
- }
- }

```
• using System;
• delegate void MDele();
• class abc
• {
• static void city()
• {
• Console.WriteLine("Patiala");
• }
• static void state()
• {
• Console.WriteLine("Punjab");
• }
• }
```

- public static void Main()
- {
- MDele show = new MDele(city);
- MDele show2 = new MDele(state);
- MDele show3 = show + show2;
- show();
- show2();
- Console.WriteLine("\n\nthis is reference method  
made by joining two delegate\n\n");
- show3();
- Console.Read0();
- }
- }

# Delegates vs. Interfaces

- Delegates and interfaces are similar in that they enable the separation of specification and implementation.
- Multiple independent authors can produce implementations that are compatible with an interface specification.
- Similarly, a delegate specifies the signature of a method, and authors can write methods that are compatible with the delegate specification.
- When should you use interfaces, and when should you use delegates?

# C# Events

- An **event** can have many handlers. With the event handler syntax in the C# language, we create a notification system. In this way we attach additional methods without changing other parts of the code. This makes programs easier to maintain.
- An event is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying event handlers.

- using System;
- public delegate void ab();
- class Program
- {
- public static event abc \_show;
- public static void Main()
- {     // Add event handlers to Show event.
- \_show += new abc(Dog);
- \_show += new abc(Cat);
- \_show += new abc(Mouse);

- // Invoke the event.
- \_show.Invoke();
- }
- static void Cat()
- {
- Console.WriteLine("Cat");
- } static void Dog()
- { Console.WriteLine("Dog");
- } static void Mouse()
- { Console.WriteLine("Mouse"); } }
- **Output** DogCatMouseMouse

# C# Indexer

- An **indexer** provides array-like syntax.
- It allows a type to be accessed the same way as an array.
- Properties such as indexers often access a backing store.
- With indexers you often accept a parameter of int type and access a backing store of array type.
- An indexer is a member that enables an object to be indexed in the same way as an array.

- **Example**
- This program contains a class that has an indexer member, which itself contains a get accessor and a set accessor. These accessors are implicitly used when you assign the class instance elements in the same way as you can assign elements in an array.
- **The indexer** provides a level of indirection where you can insert bounds-checking. In this way you can improve reliability and simplicity with indexers.

# Program that uses indexer with

- using System; **int [C#]**
- class Layout
- {
- string[] \_values = new string[100];
- // Backing store
- **public string this[int number]**
- {
- **get**
- {  
// this is invoked when accessing Layout instances with the [ ].

```
if (number >= 0 && number < _values.Length)
{
 // Bounds were in range, so return the stored value.
 return _values[number];
}

// Return an error string.
return "Error";
}

set
{
 // This is invoked when assigning to Layout instances with the [].
```

```
if (number >= 0 && number < _values.Length)
{
 // Assign to this element slot in the internal array.
 _values[number] = value;
}
}
}}
```

```
class Program
```

```
{
 public static void Main()
 {
```

```
// Create new instance and assign elements
// ... in the array through the indexer.

Layout layout = new Layout();

layout[1] = "Frank Gehry";

layout[3] = "I. M. Pei";

layout[10] = "Frank Lloyd Wright";

layout[11] = "Apollodorus";

layout[-1] = "Error";

layout[1000] = "Error";

// Read elements through the indexer.

string value1 = layout[1];
```

```
string value2 = layout[3];
string value3 = layout[10];
string value4 = layout[11];
string value5 = layout[50];
string value6 = layout[-1];
// Write the results.
Console.WriteLine(value1);
Console.WriteLine(value2);
Console.WriteLine(value3);
Console.WriteLine(value4);
Console.WriteLine(value5); // Is null
Console.WriteLine(value6); } }
```

# Change color of output in c#

- using System;
- class stringdata
- {                    public static void Main()  
    {
- string a="Good",b="Morning", c="everyone";
- **Console.ForegroundColor = ConsoleColor.Red;**
- **Console.BackgroundColor = ConsoleColor.White;**
- Console.WriteLine(a);
- Console.WriteLine(b);
- Console.WriteLine(c);
- Console.Read();
- }
- }

# Console.Clear () and Console.Beep ()

- using System;
- class stringdata
- {
- public static void Main()
- {
- string a="Good",b="Morning", c="everyone";
- Console.Beep();
- Console.Read();
- }
- }

# Checking Caps Lock Status

- using System;
- class pro
- {
- public static void Main()
- {
- if (Console.CapsLock)
  - { Console.WriteLine("Caps Lock is on!");
  - }
- else
  - { Console.WriteLine("Caps Lock is off!");
  - }
- Console.Read();
- }}

# Checking Num Lock Status

- using System;
- class pro
- {
- public static void Main()
- {
- if (Console.NumberLock )
- {     Console.WriteLine("Num Lock is on!");
- }
- else
- {     Console.WriteLine("Num Lock is off!");
- }
- Console.Read();
- } }