C#

Understanding .NET Framework:

What is .NET?

- Application Development Platform, to build Desktop, Web and Mobile Apps.
- Developed by Microsoft, in 2002.
- Provides fully managed, secured application execution environment.
- Supports multiple languages such as C#, VB, VC++ etc.

Modules & Apps:

ASP.Net

WebSites

Web Applications

WebServices

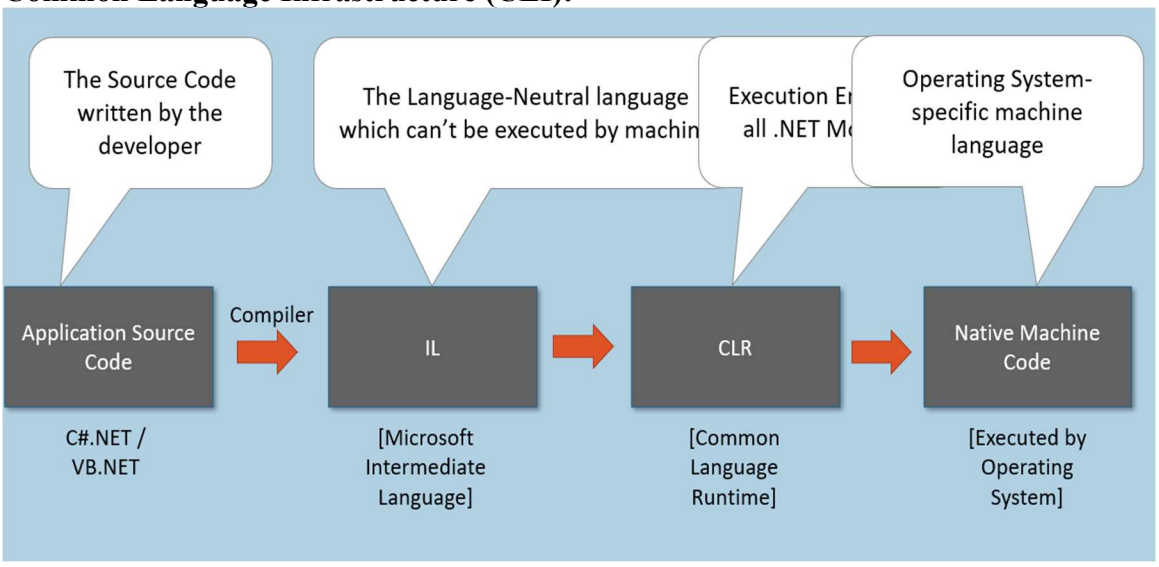
C#.Net

Windows GUI Applications

Windows Services

Console Applications

Common Language Infrastructure (CLI):



Common Language Runtime (CLR)

1. "Execution Engine" for all .net languages.
2. Code-Execution Environment that executes all types of .net applications.
3. Applications developed in any .net language runs based on "CLR" only.
4. CLR is a part of .NET Framework; pre-installed in Windows.

CLR Components:

- Class Loader
- Memory Manager
- Garbage Collector
- JIT Compiler
- Exception Manager
- Thread Manager
- Security Manager

Components of CLR

Class Loader:

Loading classes from compiled source code to memory.

Loads a class, when it is needed (before creating object).

Memory Manager:

Allocating necessary memory for objects.

When an object is created in the code, certain amount of memory will be allocated for the object in application's "heap".

Garbage Collector:

Freeing (deleting) memory of objects.

Identifies all unreferenced objects and delete them in memory (RAM).

JIT (Just-In-Time) Compiler:

Convert the MSIL Code into Native Machine Language.

Compiles the code of a class, when it is needed (before executing that particular class)

Exception Manager:

Raise notifications while run-time errors.

Creates exception logs.

Thread Manager:

Create threads (background process) to execute the code.

The entire program is treated as "Main thread".

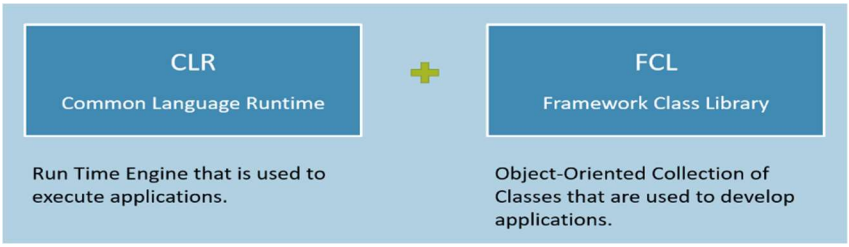
Developer can create sub threads (child threads) to do background processes.

Security Manager:

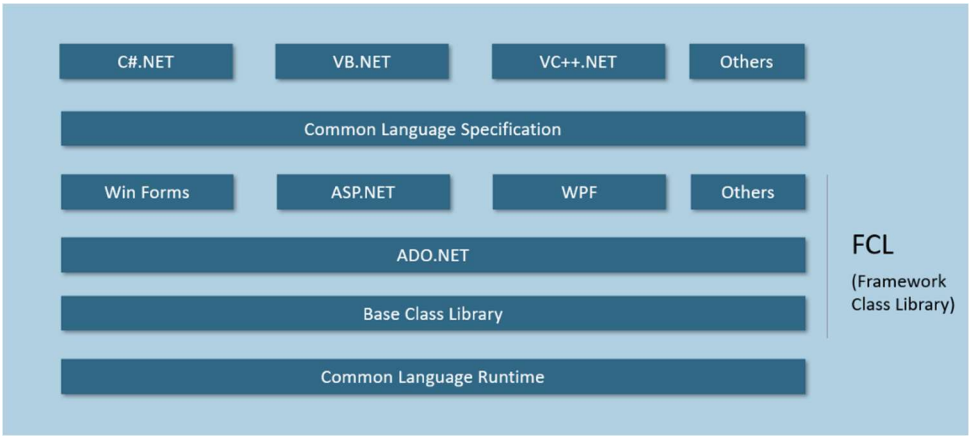
Verifies whether the application has permission to access system resources or not.

Before executing the application, it verifies whether the application has not attacked by malicious programs & has necessary permissions to access files / folders and hardware resources.

.NET Framework



.NET Framework Architecture



Base Class Library (BCL)

Contains a set of pre-defined classes that can be used in all types of .net applications & languages for general I/O operations, type conversion, creation of threads etc.

Eg:

- Console
- String
- StringBuilder
- Convert
- Thread
- Task etc..

ADO.NET

Contains a set of pre-defined classes that can be used in all types of .net applications & languages for connecting to databases, retrieving data from databases, inserting, updating, deleting rows etc.

Eg:

- SqlConnection
- SqlCommand
- SqlDataAdapter
- SqlDataReader etc..

WinForms

Contains a set of pre-defined classes that can be used in Windows GUI applications for development of GUI elements such as form, textbox, button, checkbox, radio button, dropdownlist etc.

Eg:

- Form
- Label
- Button
- Text Box etc.

Windows Presentation Foundation (WPF)

Contains a set of pre-defined classes that can be used in Rich Windows GUI applications for development of GUI elements such as window, textbox, button, checkbox, radio button, dropdownlist etc.

Eg:

- Window
- Label
- Button
- Text Box etc.

Active Server Pages (ASP.NET)

Contains a set of pre-defined classes that can be used in Web Applications for development of GUI elements such as page, textbox, button, checkbox, radio button, dropdownlist etc.

Eg:

- Page
- Label
- Button
- Text Box etc

Common Language Specification (CLS)

Contains a set of rules (concepts) that are common to all .net languages such as C#.NET, VB.NET etc.

Common rules of CLS:

- CTS (Common Type System): Contains data types such as Int32, Int64, Single, Double etc.
- Classes & Objects
- Reference Variables
- Method Parameters
- Generics etc

Versions of .NET Framework

.NET Framework	Year of Release	CLR	C#	VB
1.0	2002	1	1.0	7.0
1.1	2003	1	1.2	7.0
2.0	2005	2	2.0	8.0
3.0	2006	2	2.0	8.0
3.5	2008	2	3.0	9.0
4.0	2010	4	4.0	10.0
4.5	2012	4	5.0	11.0
4.5.1	2013	4	5.0	12.0
4.5.2	2014	4	5.0	13.0
4.6	2015	4	6.0	14.0
4.6.1	2015	4	6.0	14.0
4.6.2	2016	4	6.0	14.0
4.7	2017	4	7.0	15.0
4.7.1	2017	4	7.1 & 7.2	15.0
4.7.2	2018	4	7.3	15.0
4.8	2019	4	8.0	16.0

.NET 1.0 New Features

1. Data Types
2. Classes
3. Objects
4. Arrays
5. Collections
6. All other basic language features.

.NET 1.1 New Features

1. Code Access Security in ASP.NET applications
2. ODBC (Open Database Connection)
3. .NET Compact Framework to run .net apps on small devices

.NET 2.0 New Features

1. 64-bit system execution support
2. Themes, Skins, MasterPages and WebParts, Membership in ASP.NET
3. Partial Classes, Nullable Types, Anonymous Methods, Iterators, Generics in C#
4. CLR 2.0

.NET 3.0 New Features

1. WPF (Windows Presentation Foundation): Framework to develop rich Windows GUI Apps.
2. WCF (Windows Communication Foundation): Framework to develop Service Oriented Applications (SOA).
3. WWF (Windows Workflow Foundation): Framework to develop task automation and transactions using workflows.

.NET 3.5 New Features

1. Data Annotations
2. Entity Framework

.NET 4.0 New Features

1. Task Parallel Library (Tasks)
2. Named Parameters in C#
3. CLR 4.0

.NET 4.5 New Features

1. Windows Store Apps
2. Async and Await in C#

3. New input types, Bundling & Minification, Web Sockets, Anti-XSS in ASP.NET

.NET 4.6 New Features

1. SHA-2, Elliptic Curve Cryptography API
2. 64-bit JIT compiler for Managed Code
3. Introduction of .NET Core (Provides multi-platform support for Windows, LINUX, Mac, Android, iOS).

.NET 4.7 New Features

1. Print-API and Stylus support for WPF
2. Introduction of ASP.NET Core (Provides multi-platform support for Windows, LINUX, Mac as Web Servers).

.NET 4.8 New Features

1. Performance & security updates.
2. Support for high-resolution displays.

.NET Core

- Introduced in 2016.
- Microsoft's Application Development Platform to develop any desktop, web, mobile and embedded (IoT) applications.
- Supports Windows, LINUX, Mac, Android, iOS, Windows Phone devices.
- Versions: 1.0, 1.1, 2.0, 2.1, 2.2, 3.0, 3.1
- Mainly used in Xamarin and ASP.NET Core.
- Open Source (via MIT license)
- Contains the class library, which is a subset of .NET Framework.
- Doesn't contain any Windows-specific classes / run time services.
- C# / VB.NET can used for writing code of .NET Core.

Introducing Visual Studio

IDE / Code editor for all types of .net applications & languages.

Should be installed on 'Developer-machine'.

Features:

- Intelli-sense (code completion)
- Syntax highlighting
- Debugger
- GUI Designer

Versions of Visual Studio

Visual Studio	Year of Release	Version Number	Supported .NET Framework versions	Supported O/S
Visual Studio 2002	2002	7.0	.NET Framework 1.0	Windows 2000, XP
Visual Studio 2003	2003	7.1	.NET Framework 1.1	Windows 2000, XP
Visual Studio 2005	2005	8.0	.NET Framework 2.0	Windows 2000, XP
Visual Studio 2008	2007	9.0	.NET Framework 2.0, 3.0, 3.5	Windows 2000, XP, 7
Visual Studio 2010	2010	10.0	.NET Framework 2.0, 3.0, 3.5, 4.0	Windows XP, 7
Visual Studio 2012	2012	11.0	.NET Framework 2.0, 3.0, 3.5, 4.0, 4.5	Windows 7, 8.1, 10
Visual Studio 2013	2013	12.0	.NET Framework 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1	Windows 7, 8.1, 10
Visual Studio 2015	2015	14.0	.NET Framework 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2 .NET 1.0	Windows 7, 8.1, 10
Visual Studio 2017	2017	15.0	.NET Framework 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2 .NET 1.0, 1.1, 2.0, 2.1	Windows 7, 8.1, 10
Visual Studio 2019	2019	16.0	.NET Framework 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8 .NET 2.1, 2.2, 3.0, 3.1, 5.0	Windows 8.1, 10
Visual Studio 2022	2021	17.0	.NET Framework 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1 .NET 2.1, 3.1, 5.0, 6.0, 7.0, 8.0	Windows 8.1, 10

The System.Console class

The "Console" is a class in "System" namespace, which provided a set of properties and methods to perform I/O operations in Console Applications (Command-Prompt Applications).

It is a static class. So all the members of "Console" class are accessible without creating any object for the "Console" class.

The "Console" class is a part of BCL (Base Class Library)

Members of 'Console' class

void Write(value):

It receives a value as parameter and displays the same value in Console (Command-Prompt window).

void WriteLine(value):

It receives a value as parameter and displays the same value in Console and also moves the cursor to the next line, after the value.

void ReadKey( ):

It waits until the user presses any key on the keyboard.

It makes the console window wait for user's input.

void Clear( ):

It clears (make empty) the console window.

After clearing the screen, you can display output again, using Write( ) or WriteLine( ) methods.

string ReadLine( ):

It accepts a string value from keyboard (entered by user) and returns the same

It always returns the value in "string" type only.

Even numbers (digits) are treated as strings.

What is Variable?

Variable is a named memory location in RAM, to store a particular type of value, during the program execution.

All Variables will be stored in Stack.›For every method call, a new "Stack" will be created.

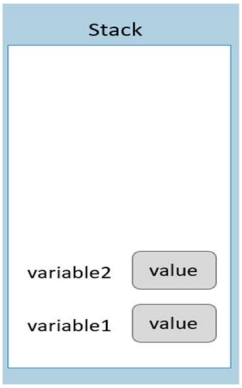
The variable's value can be changed any no. of times.

The variables must be declared before its usage.

The variables must be initialized before reading its value.

Variable's data type should be specified while declaring the variable; it can't be changed later.

The stack (along with its variables) will be deleted automatically, at the end of method execution.



How to create Variables?

Syntax to create a variable:

```
data_type  variable_name;
```

[or]

```
data_type  variable_name = value;
```

Set value into the variable:

```
variable_name = value;
```

Get value from the variable:

```
variable_name
```

Variable / Identifier Naming Rules

1. Variable name should not contain spaces.

Student Name: wrong

StudentName: correct

2. Variable name should not have special characters [except underscore]

Student#Name: wrong

StudentName: correct

3. Duplicate variable names are not allowed.

int x;

double x: wrong (already there was a variable with same name (x)).

4. Variable names can't be same as keywords

int void: wrong

int StudentNo: correct

What is 'type'?

'Type' specifies what type of value to be stored in memory.

"Type" is a.k.a. "data type".

Eg: int, string etc.

Classification of Types

Primitive Types:

(sbyte, byte, short, ushort, int, uint, long, ulong, float, double, decimal, char, bool)

• Strictly stores single value.

• Primitive Types are basic building blocks of non-primitive types

Non-Primitive Types:

(string, Classes, Interfaces, Structures, Enumerations)

• Stores one or more values.

• Usually contains multiple members.

Primitive Types

sbyte:

• 8-bit signed integer

• Size: 1 byte

• Range: -128 to 127

• Default value: 0

• MinValue Command: sbyte.MinValue

• MaxValue Command: sbyte.MaxValue

byte:

• 8-bit un-signed integer

• Size: 1 byte

• Range: 0 to 255

• Default value: 0

• MinValue Command: byte.MinValue

• MaxValue Command: byte.MaxValue

short:

• 16-bit signed integer

• Size: 2 bytes

• Range: -32,768 to 32,767

• Default value: 0

• MinValue Command: short.MinValue

• MaxValue Command: short.MaxValue

ushort:

• 16-bit un-signed integer

• Size: 2 bytes

• Range: 0 to 65,535

• Default value: 0

• MinValue Command: ushort.MinValue

• MaxValue Command: ushort.MaxValue

int:

• 32-bit signed integer

• Size: 4 bytes

• Range: -2,147,483,648 to 2,147,483,647

• Default value: 0

• MinValue Command: int.MinValue

• MaxValue Command: int.MaxValue

• By default, integer literals between -2,147,483,648 to 2,147,483,647 are treated as "int" data type.

uint:

• 32-bit un-signed integer

• Size: 4 bytes

• Range: 0 to 4,294,967,295

• Default value: 0

• MinValue Command: uint.MinValue

• MaxValue Command: uint.MaxValue

• By default, integer literals between 2,147,483,648 to 4,294,967,295 are treated as "uint" data type.

long:

• 64-bit signed integer

• Size: 8 bytes

• Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

• Range: -263 to 263 (power -1)

• Default value: 0

• MinValue Command: long.MinValue

• MaxValue Command: long.MaxValue

• By default, integer literals between 4,294,967,296 and 9,223,372,036,854,775,807 are treated as "long" data type.

ulong:

• 64-bit un-signed integer

• Size: 8 bytes

• Range: 0 to 18,446,744,073,709,551,615

• Default value: 0

• MinValue Command: ulong.MinValue

• MaxValue Command: ulong.MaxValue

• By default, integer literals between 9,223,372,036,854,775,808 and 18,446,744,073,709,551,615 are treated as "ulong" data type.

float:

• 32-bit signed floating-point number

• Size: 4 bytes

• Range: -340282300000000000000000000000000000000 to 340282300000000000000000000000000000000

• Range: -3.402823E+38 to 3.402823E+38

• Range: "MINUS three hundred fourty two hundred eighty-two three hundred nonillion" *to* "three hundred fourty two hundred eighty-two three hundred NONILLION"

• Precision: 7 digits

• Default value: 0F

• MinValue Command: float.MinValue

• MaxValue Command: float.MaxValue

double:

• 64-bit signed floating-point number

• Size: 8 bytes

•
179769313486232000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000

000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000 (294 0's)

to 1797693134862320000000000000000000000000000000000000000000000000000000
000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
(294 0's)

• Range: -1.79769313486232E+308 to 1.79769313486232E+308

• "MINUS one hundred seventy-nine trillion seven hundred sixty-nine billion three hundred thirteen million four hundred eighty-six thousand two hundred thirty-two UNTRIGINTILLION DUOTRIGINTILLION DUOTRIGINTILLION" *to* "one hundred seventy-nine trillion seven hundred sixty-nine billion three hundred thirteen million four hundred eighty-six thousand two hundred thirty-two UNTRIGINTILLION DUOTRIGINTILLION DUOTRIGINTILLION"

• Precision: 15 digits

• Default value: 0D

• Min and Max: double.MinValue, double.MaxValue

• By default, floating-point literals in the specified range are treated as "double" data type.

decimal:

• 128-bit signed floating-point number

• Size: 16 bytes

• Range: -79228162514264337593543950335 to 79228162514264337593543950335

• "MINUS seventy-nine octillion two hundred twenty-eight septillion one hundred sixty-two sextillion five hundred fourteen quintillion two hundred sixty-four quadrillion three hundred thirty-seven trillion five hundred ninety-three billion five hundred fourty-three million nine hundred fifty thousand three hundred thirty-five" *to* "seventy-nine octillion two hundred twenty-eight septillion one hundred sixty-two sextillion five hundred fourteen quintillion two hundred sixty-four quadrillion three hundred thirty-seven

trillion five hundred ninety-three billion five hundred fourty-three million nine hundred fifty thousand three hundred thirty-five"

• Precision: 28 digits

• Default value: 0M

• Min and Max: double.MinValue, double.MaxValue

char:

• 16-bit Single Unicode character

• Character literal should be written in single quotes only. Ex: 'A'

• Size: 2 bytes

• Range: 0 to 137,994 (Unicode codes that represent characters)

• Unicode is superset of ASCII.

• ASCII = 0 to 255 (English language characters only)

• Unicode = ASCII + Other natural language characters

• Default value: \0

Important ASCII / Unicode numbers for characters:

65 to 90 : A-Z

97 to 122: a-z

48 to 57: 0-9

32: Space

8: Backspace

13: Enter

string:

• Collection of Unicode characters

• String literal should be written in double quotes only. Ex: "Abc123"

• Size: Length * 2 bytes

• Range: 0 to 2 billion characters

• Default value: null

bool:

• Stores logical value (true / false)

• Possible values: true, false

• Size: 1 bit

• Default value: false

Default Literals

You can get the default value of respective type using the following syntax.

default(type)

Example: default(int) = 0

Operators

Operator is a symbol to perform operation.

Operator receives one or more operands (values) and returns one value.

Eg: +, -, *, /, == etc.

1. Arithmetical Operators
2. Assignment Operators
3. Increment and Decrement Operators
4. Comparison Operators

5. Logical Operators
6. Concatenation Operator
7. Ternary Operator

Arithmetical Operators

Used to perform arithmetical operations on the numbers

+ Addition

- Subtraction

* Multiplication

/ Division

% Remainder

Assignment Operators

Used to perform arithmetical operations on the numbers

= Assigns to

+= Add and Assigns to

-= Subtract and Assigns to

*= Multiply and Assigns to

/= Divide and Assigns to

%= Remainder Assigns to

Increment / Decrement Operators

Used to perform arithmetical operations on the numbers

It returns the incremented / decremented value and also overwrites the value of variable.

n++ Post-Incrementation (First it returns value; then increments)

++ n Pre-Incrementation (First it increments value; then returns)

n-- Post-Decrementation (First it returns value; then decrements)

--n Pre-Decrementation (First it decrements value; then returns)

Comparison Operators

Used to compare two values and return true / false, based on the condition.

== equal to

!= not equal to

< less than

> greater than

<= less than or equal to

>= greater than or equal to

Logical Operators

Checks both operands (Boolean) and returns true / false.

& Logical And (Both operands should be true). Evaluates both operands, even if left-hand operand returns false.

&& Conditional And (Both operands should be true). Doesn't evaluate right-hand operand, if left-hand operand returns false.

| Logical Or (At least any one operand should be true). Evaluates both operands, even if left-hand operand returns true.

|| Conditional Or (At least any one operand should be true). Doesn't evaluate right-hand operand, if left-hand operand returns true.

Comparison Operators

^ Logical Exclusive Or - XOR (Any one operand only should be true). Evaluates both operands.

! Negation (true becomes false; False becomes true)

Concatenation Operator

Attaches second operand string at the end of first operand string and returns the combined string.

+ "string1" + "string2" returns "string1string2" (as string)

"string" + number returns "stringnumber" (as string)

number + "string1" returns "numberstring" (as string)

Ternary Conditional Operator

It evaluates the given Boolean value;

Returns first expression (consequent) if true;

Returns second expression (alternative) if false.

```
? :   (condition)? consequent : alternative
```

Operator Precedence

Category	Operator
Postfix	() [] ++ --
Unary	+ - !
Multiplicative	* / %
Additive	+ -
Relational	< <= > >=
Equality	== !=
Logical AND	&&
Logical OR	||
Conditional	?:
Assignment	= += -= *= /= %=

Control Statements

Conditional Control Statements:

- if (simple-if, if-else, else-if, nested-if)
- switch-Case

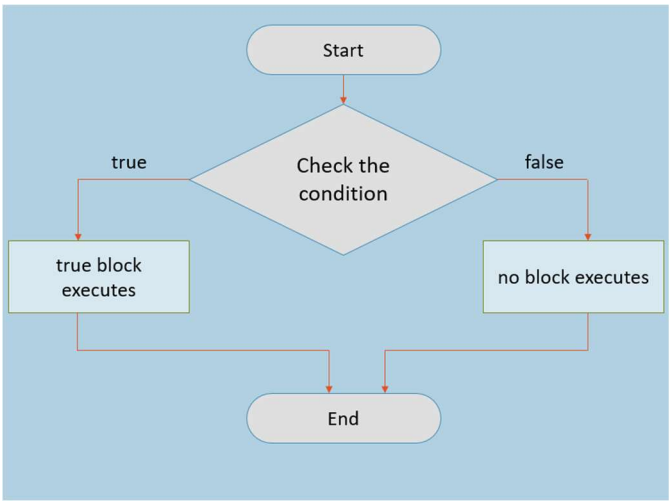
Looping Control Statements:

- while
- do-While
- for

Jumping Control Statements:

- goto
- break
- continue

simple-if



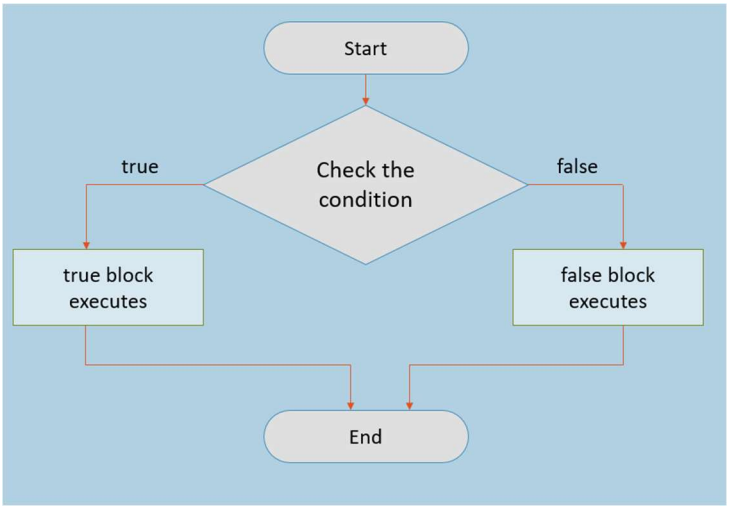
Simple-if - Syntax

```
1. if (condition)
2. {
3.   true block here
4. }
```

Simple-if - Example

```
if (x < 10)
{
  System.Console.WriteLine("x is smaller than 10");
}
```

if-else



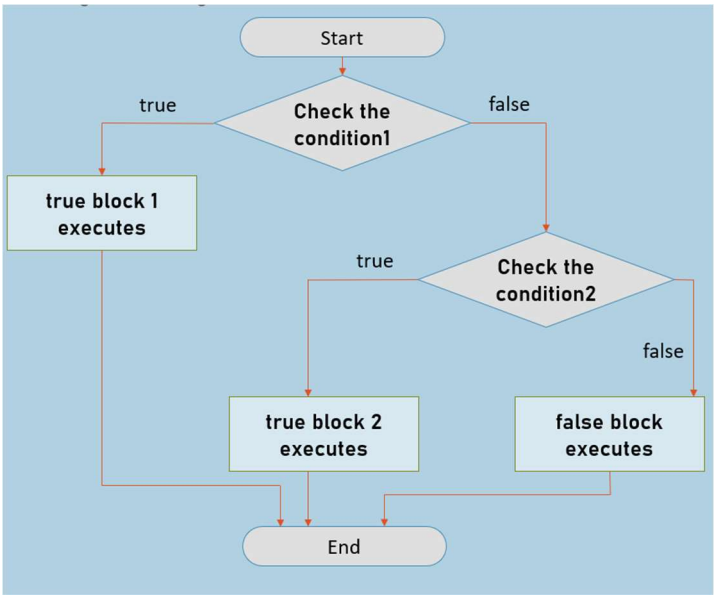
if - else - Syntax

```
if (condition)
{
  true block here
}
else
{
  false block here
}
```

if-else - Example

```
if (x > 10)
{
  System.Console.WriteLine("x is larger than 10");
}
else
{
  System.Console.WriteLine("x is smaller than or equal to 10");
}
```

else-if



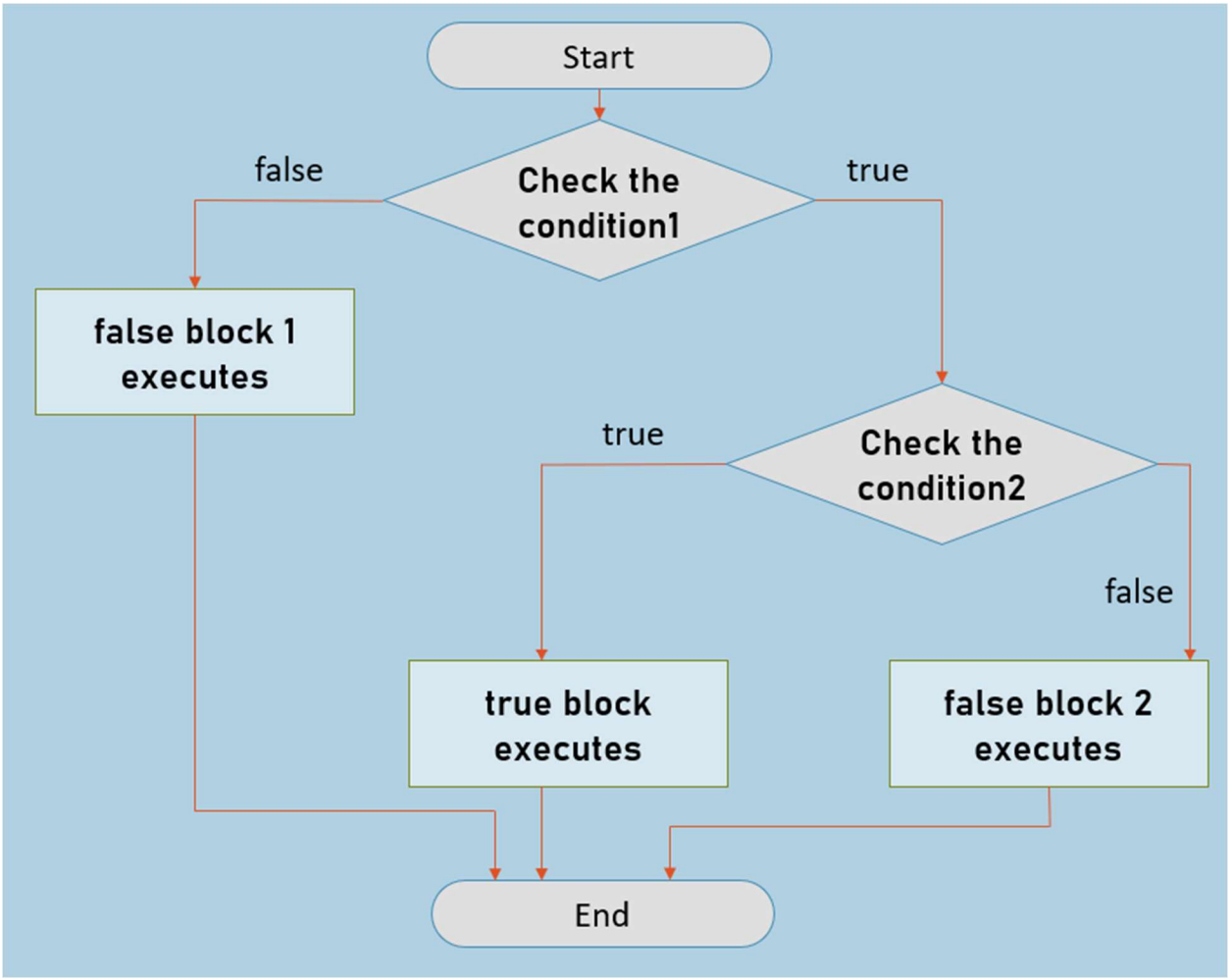
else-if - Syntax

```
if (condition1)
{
  true block 1 here
}

else if (condition2)
{
  true block 2 here
}

else
{
  false block here
}
```

else If - Example

```
if (a > 10)
{
  System.Console.WriteLine("a is greater than 10");
}
else if (a < 10)
{
  System.Console.WriteLine("a is less than 10");
}
else
{
  System.Console.WriteLine("a is equal to 10");
}
```

nested-if



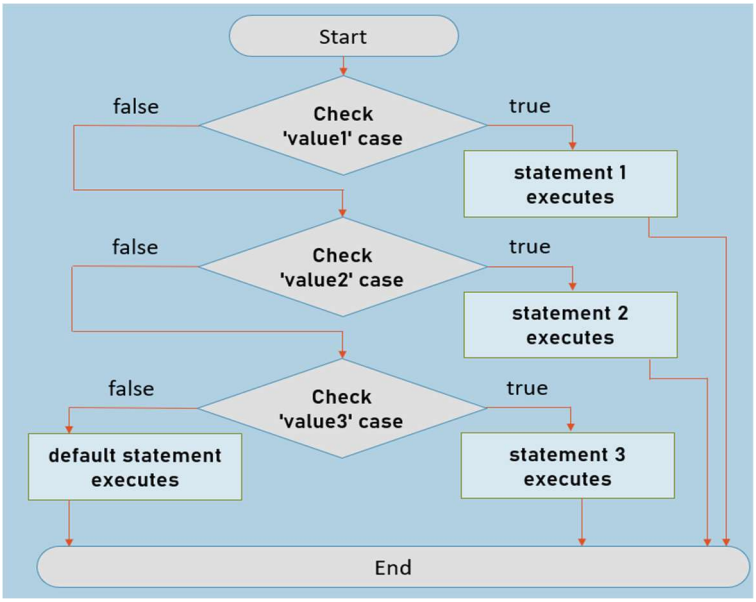
nested-if - Syntax

```
1. if (condition1)
2. {
3.   if (condition2)
4.   {
5.     true block here
6.   }
7.   else
8.   {
9.     false block 2 here
10.  }
11.}
12.else
13.{
14.  false block 1 here
15.}
```

nested If - Example

```
if (a >= 10)
{
  if (a > 10)
  {
    System.Console.WriteLine("a is greater than 10");
  }
  else
  {
    System.Console.WriteLine("a is equal to 10");
  }
}
else
{
  System.Console.WriteLine("a is less than 10");
}
```

switch-case



switch-case - Syntax

```
switch (variable)
{
  case value1: statement1; break;
  case value2: statement2; break;
  case value3: statement3; break;
  …
  default: statement; break;
}
```

switch-case - Example

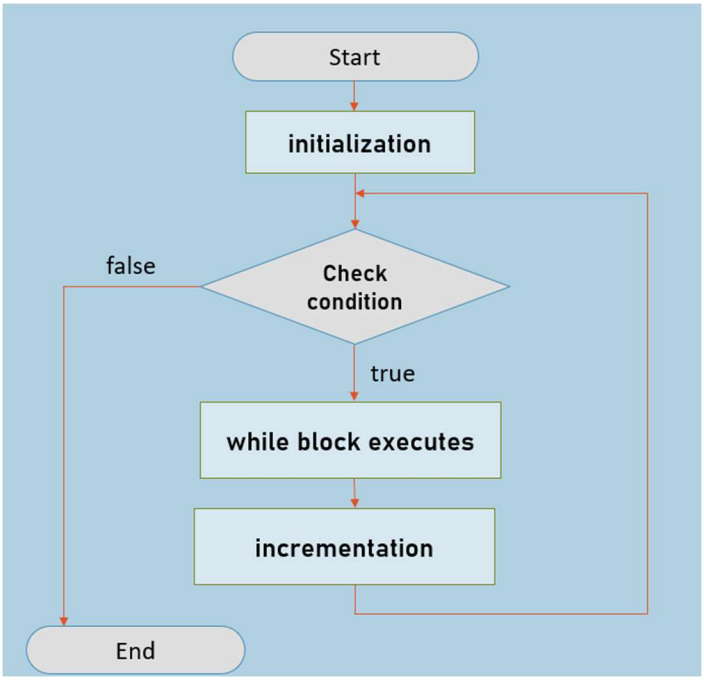
```
switch ( x )
{
  case 1: System.Console.WriteLine("one"); break;
  case 2: System.Console.WriteLine("two"); break;
  case 3: System.Console.WriteLine("three"); break;
  default: System.Console.WriteLine("none"); break;
}
```

Used to check a variable value, many times, whether it matches with any one of the list of values.

Among all cases, only one will execute.

If all cases are not matched, it executes the "default case".

while



while - Syntax

```
initialization;
while (condition)
{
  while block
  incr / decr here
}
```

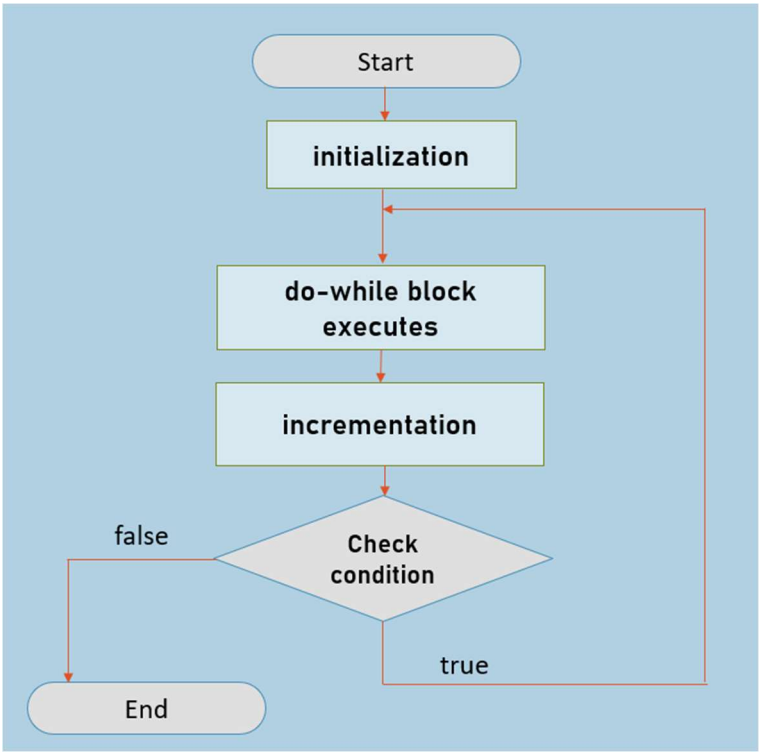
while - Example

```
1. int i = 1;
2. while ( i <= 10 )
3. {
4.   System.Console.WriteLine( i );
5.   i++;
6. }
```

Used to execute a set of statements, as long as the condition is TRUE.

Once the condition is false, it will exit from the while loop.

do-while



do-while - Syntax

```
1. initialization;
2. do
3. {
4.   do-while block
5.   incr / decr here
6. } while (condition);
```

do-while - Example

```
1. int i = 1;
2. do
3. {
4. System.Console.WriteLine( i ); i++;
5. } while ( i <= 10 );
```

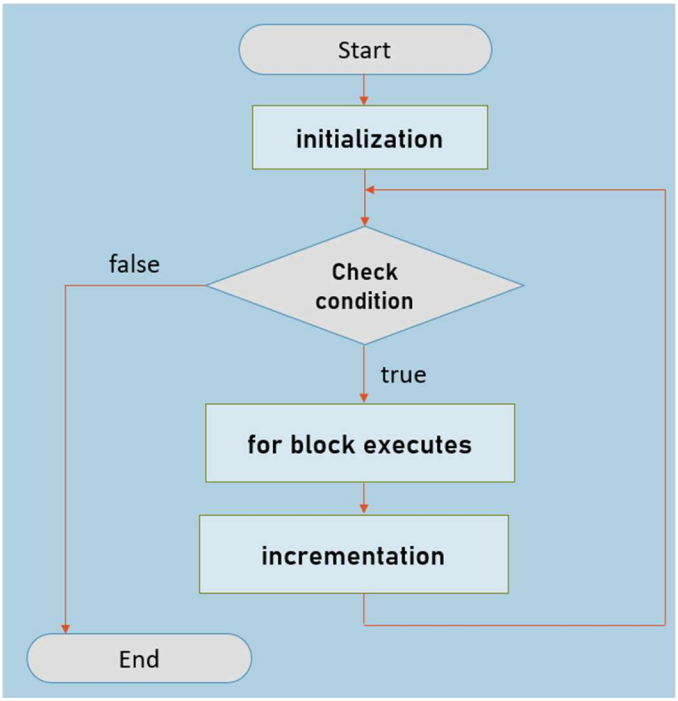
Used to execute a set of statements, as long as the condition is TRUE.›Once the condition is false, it will exit from the while loop.

It is same as "While loop"; but the difference is:

It executes at least one time even though the condition is false, because it doesn't check the condition for the first time.

Second time onwards, it is same as "while" loop.

for



for - Syntax

```
1. for (initialization; condition; incrementation)
2. {
3.   for block
4.
```

for - Example

```
1. for (int i = 1; i <= 10; i++)
2. {
3.   System.Console.WriteLine( i );
4. }
```

Used to execute a set of statements, as long as the condition is TRUE.

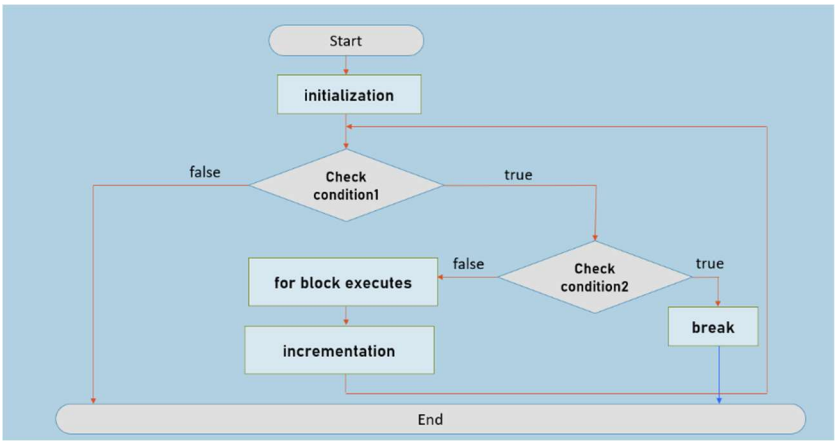
Once the condition is false, it will exit from the while loop.

It is same as "While loop"; but the difference is:

We can write all loop details (initialization, condition, incrementation), in-one-line.

break

```
1. break - Syntax
2. for (initialization; condition1; incrementation)
3. {
4.   if (condition2)
5.   {
6.     break;
7.   }
8.   for block code here
9. }
```



break - Syntax

```
1. for (initialization; condition1; incrementation)
2. {
3.   if (condition2)
4.   {
```

```
    break;
  }
  for block code here
}
```

break - Example

```
for (int i = 0; i <= 10; i++)
{
  if (i == 6)
  {
    break;
  }
  System.Console.WriteLine(i);
}
//Output: 0, 1, 2, 3, 4, 5
```

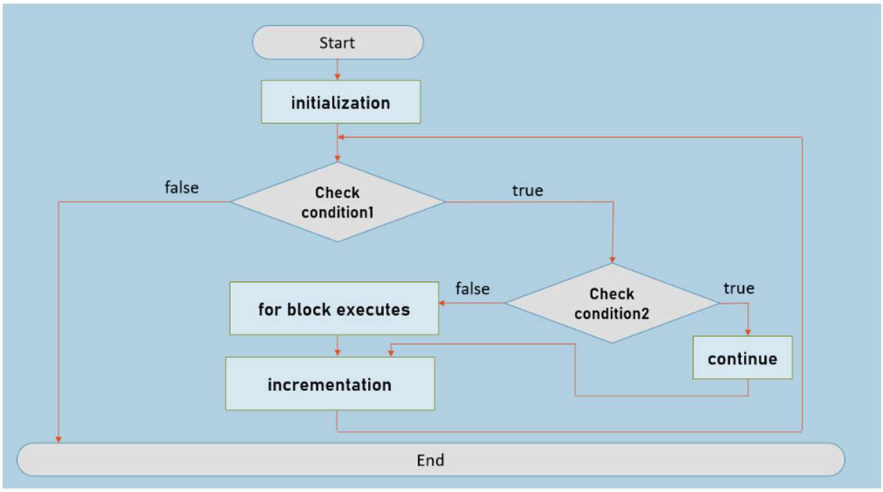
Used to stop the execution of current loop.

It is recommended to keep the "break" statement, inside "if" statement.

It can be used in any type of loop (while, do-while, for).

continue

```
for (initialization; condition1; incrementation)
{
  if (condition2)
  {
    continue;
  }
  for block code here
}
```



continue - Syntax

```
for (initialization; condition1; incrementation)
{
  if (condition2)
  {
    continue;
  }
  for block code here
}
```

continue - Example

```
for (int i = 0; i <= 10; i++)
{
  if (i == 6)
  {
    continue;
  }
  System.Console.WriteLine(i);
}
//Output: 0, 1, 2, 3, 4, 5, 7, 8, 9, 10
```

Used to skip the execution of current iteration; and jump to the next iteration.

It is recommended to keep the "continue" statement, inside "if" statement.

It can be used in any type of loop (while, do-while, for).

nested-for

nested for - Syntax

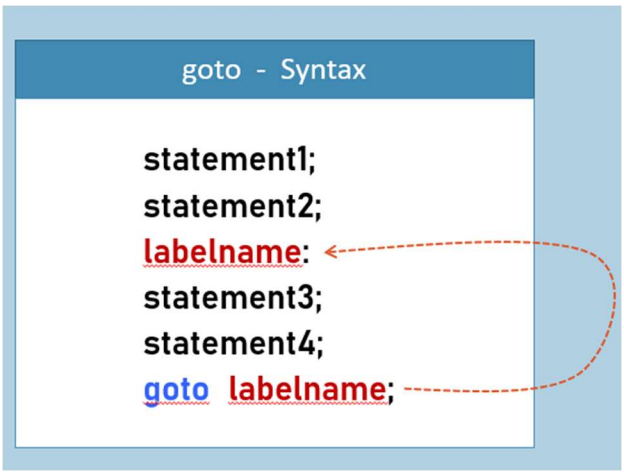
```
for (initialization; condition1; incrementation)
{
  for (initialization; condition2; incrementation)
  {
    inner-loop code here
  }
  outer-loop code here
}
```

nested for - Example

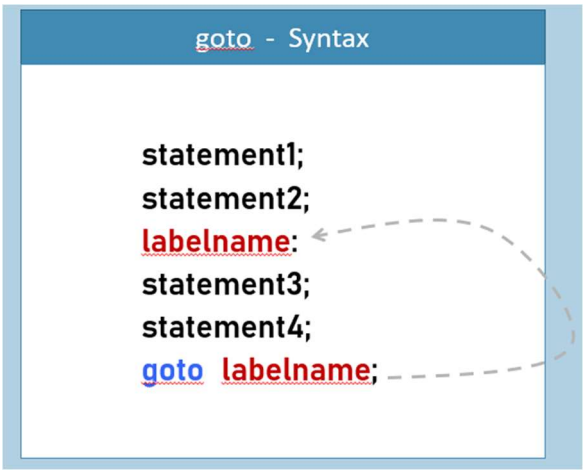
```
for (int i = 0; i < 5; i++)
{
  for (int j = 0; j < 5; j++)
  {
    System.Console.WriteLine( j );
  }
```

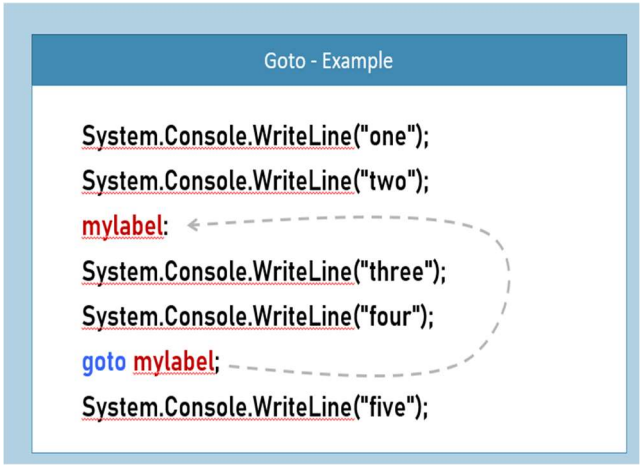
```
7. }
8.
9. //Output: 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2,
   3, 4,
```

goto



goto - Syntax





Used to jump to the specific label.

You must create a label with some specific name.

The label can be present at the top of "goto statement"; or at the bottom; but it should be in the same method.

Introduction to OOP

Programming Model for Scalable Applications.

Used in most popular languages such as Java, Python, JavaScript, C++ etc.

Goal of OOP is to group-up some data and its operations as a single unit called "Object".

Objects

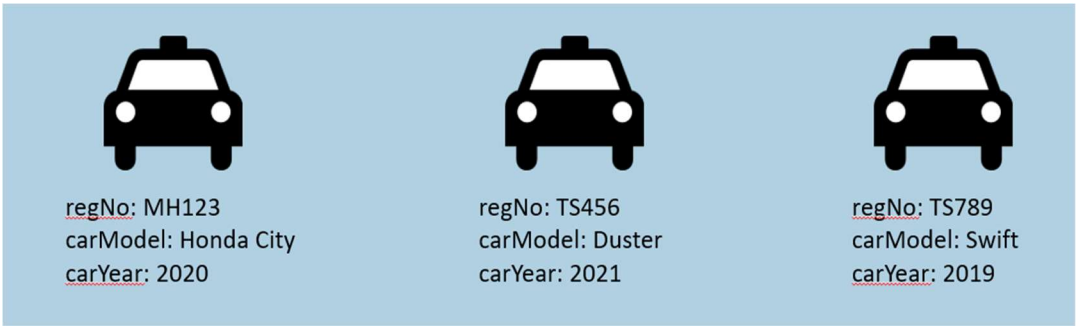
Object is a small unit (entity) in the program that represents a real-world person or thing.

Eg: You, Your laptop

Any physical thing can be considered as object.

Object is instance (example) of "class".

Object stores a set of fields (details about object).



Classes

Class is a model of objects.

Class (a.k.a "type") represents structure (list of fields and methods) of data that you want to store in similar objects.

Class isn't collection of objects.

Objects are created based on "Class".

Eg:

```
1. class Car
2. {
3.   string regNo;
4.   string carModel;
5.   int carYear;
6. }
```

Methods

Method is a collection of statements to perform certain operation (process or work), such as performing some calculation, displaying some output, checking some conditions etc.

Method should be a member (part) of class.

The code statements are not allowed outside the class; they are allowed inside the method only.

Eg:

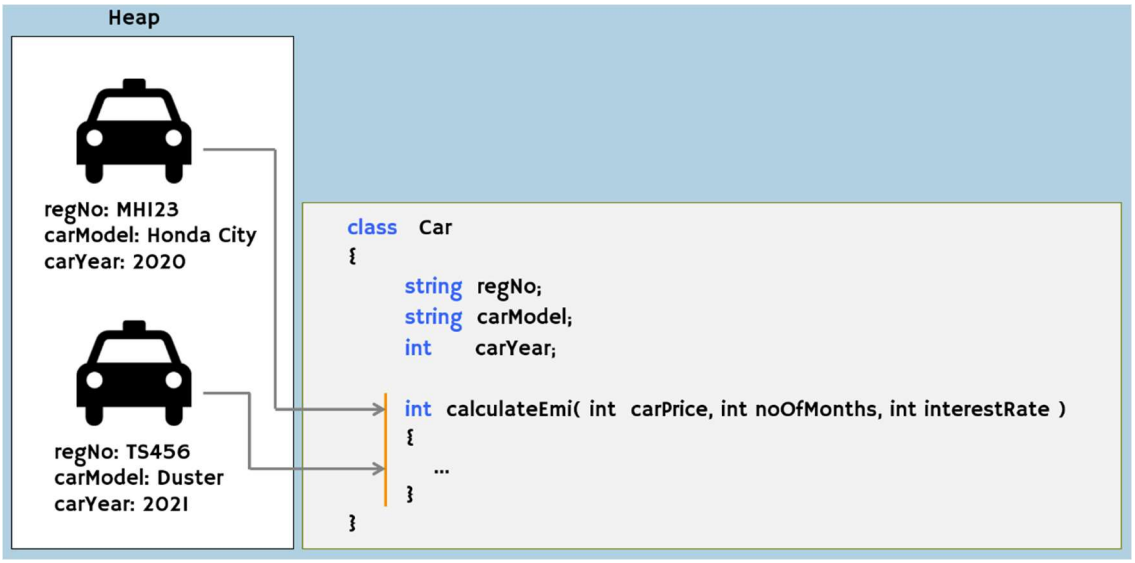
```
1. class Car
2. {
3.   int calculateEmi( int carPrice, int noOfMonths, int interestRate )
4.   {
5.     //do calculation here
6.     return (emi);
7.   }
8. }
```

Object & Class Association

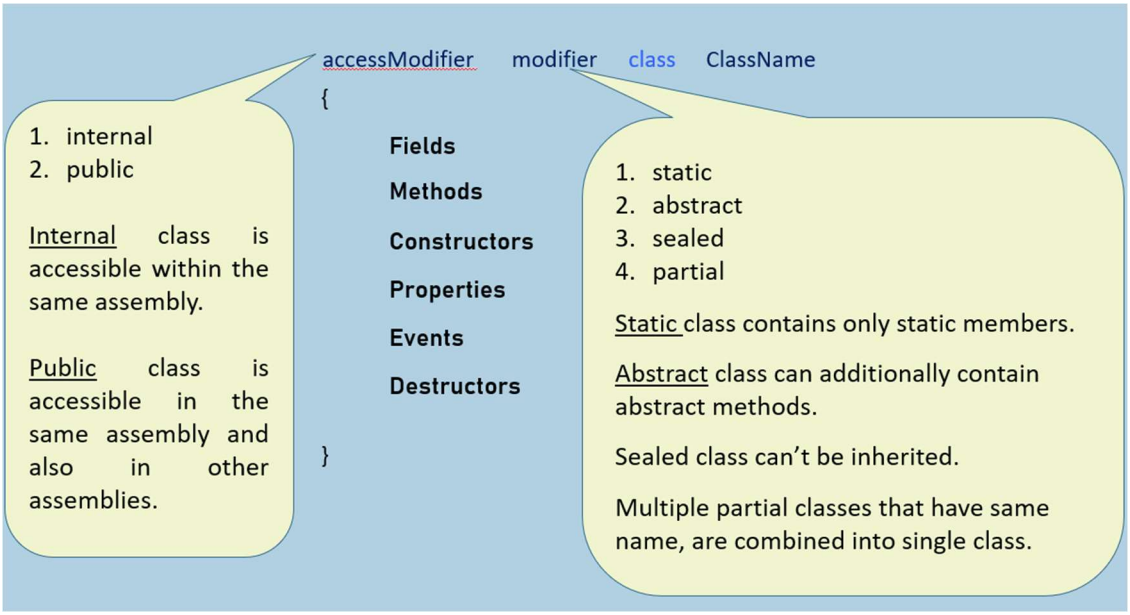
Object stores fields.

Object associates with all methods of its class. Means, object can call methods of its class.

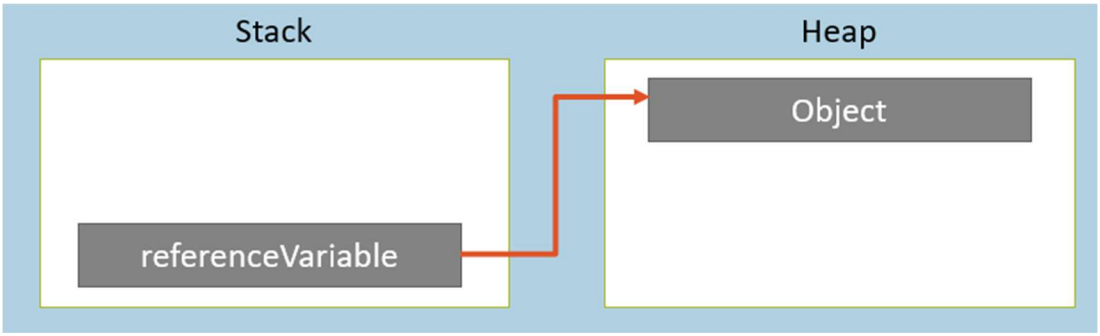
Class declares list of fields; defines list of methods.



Creating Class



Creating Object



1. Creating Reference Variable

```
ClassName referenceVariable;
```

2. Create Object and Store its reference into the Reference Variable

```
referenceVariableName = new ClassName( );
```

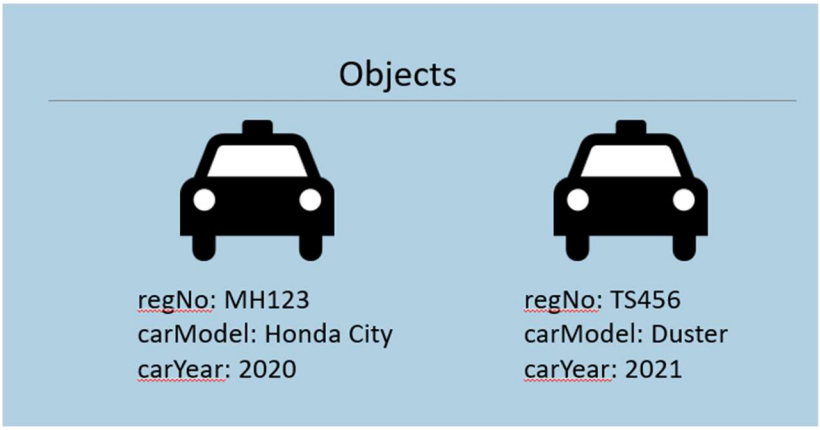
Key Points to Remember

- Object is a programmatic representation of a person or thing.
- All objects are created based on classes; stored in 'heap'.
- For each application execution, a new heap will be created (and only one).
- All reference variables (local variables of methods) are stored in stack. For each method call, a new stack will be created.
- Method is a collection of statements to perform some operation / calculation.
- Class supports two access modifiers: 'internal' and 'public'.
- Class supports four modifiers: 'static', 'abstract', 'sealed' and 'partial'.
- Objects stores actual data (group of fields) & can access methods of class.
- A reference variable stores address of an (only one) object.

Fields

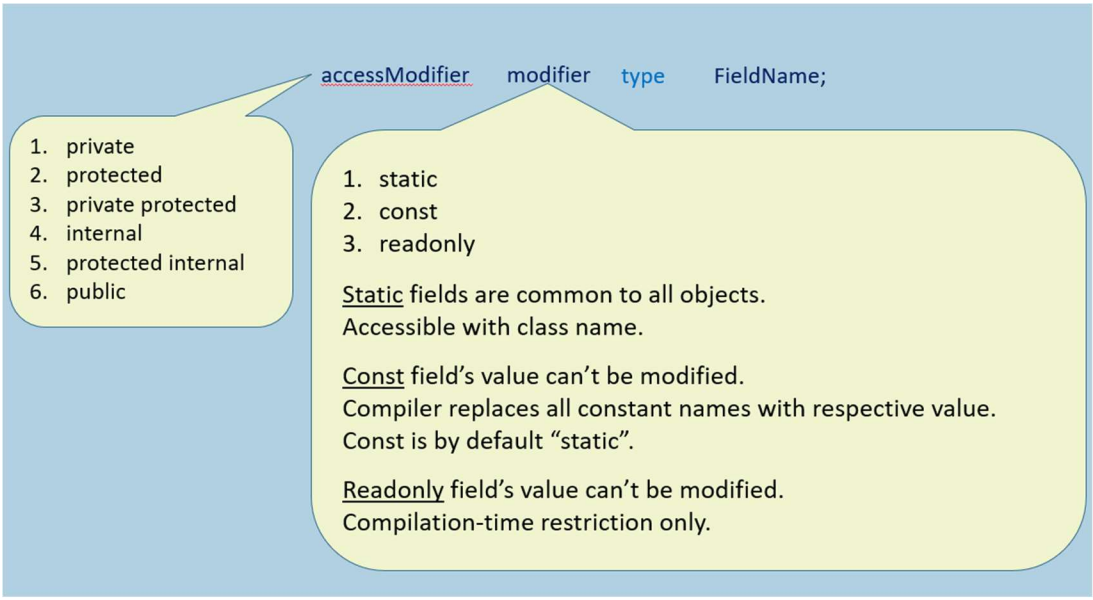
Variables that are declared in the class; stored in the objects.

Isolated for each object.




```
1. class Car
2. {
3.   string regNo;
4.   string carModel;
5.   int carYear;
6. }
```

Syntax of Field



Access Modifiers of Fields

Access Modifier (a.k.a. "Access Specifier" or "Visibility Modifier) specifies the accessibility of fields, where the fields can be accessible; they provide security for the fields.

Access Modifier	In the same class	In the child classes at the same assembly	In the other classes at the same assembly	Child classes at other assembly	Other classes at other assembly
private	Yes	No	No	No	No
protected	Yes	Yes	No	Yes	No
private protected	Yes	Yes	No	No	No
internal	Yes	Yes	Yes	No	No
protected internal	Yes	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes	Yes

Static Fields

Static fields are store outside the object.

Static fields are common to all objects of a class.

Class Memory in Heap

bankName: Bank of Dummyland

Objects in Heap

object 1:

accountNumber: 1001

accountHolderName: Scott

currentBalance: 5000

object 2:

accountNumber: 1002

accountHolderName: Bob

currentBalance: 6000

```
1. class BankAccount
2. {
3.   long accountNumber;
4.   string accountHolderName;
5.   double currentBalance;
6.   static string bankName;
7. }
```

Instance Fields (vs) Static Fields

Storage:

Instance Fields: Stored in Objects

Static Fields: Stored in Class's memory.

Related to:

Instance Fields: Represents data related to objects.

Static Fields: Represents common data that belongs to all objects

Declaration:

Instance Fields: Declared without "static" keyword. Syntax: type fieldName;

Static Fields: Declared with "static" keyword. Syntax: static type fieldName;

Accessible with:

Instance Fields: Accessible with object (through reference variable).

Static Fields: Accessible with class name only (not with object).

When memory gets allocated:

Instance Fields: Allocated separately for each object, because instance fields are stored "inside" the objects.

Static Fields: Allocated only once for the entire program; i.e. when the class is used for the first time while executing the program

Constant Fields

Constant Fields are like static fields, that are common to all objects of the class.

We can't change the value of constant field.

Constant Fields are accessible with class name [not with object].

Constant Fields are not stored in the object; will not be stored anywhere.

Constant Fields will be replaced with its value, while compilation; so it will not be stored anywhere in memory.

Constant Fields must be initialized, in line with declaration (with a literal value only).

Constants can also be declared as 'local constants' (in a method).

AccessModifier const type FieldName = value;

Readonly Fields

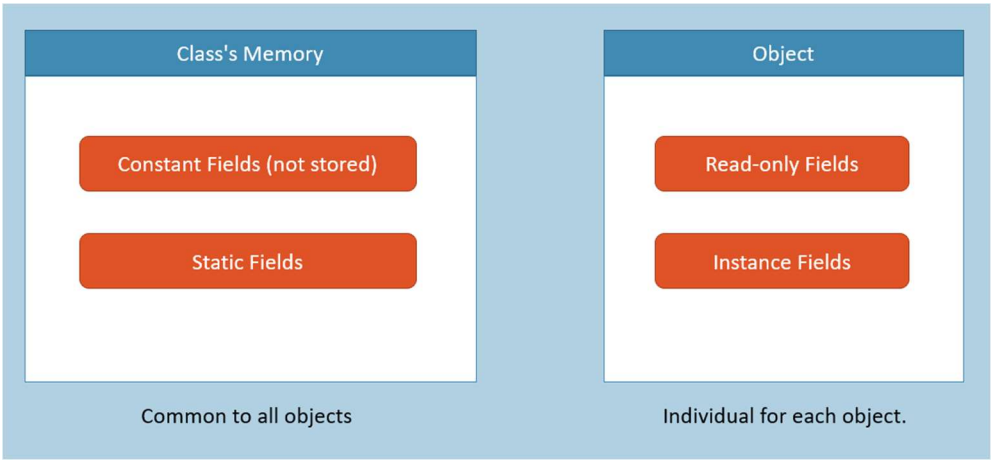
Readonly Fields are like instance fields, that is stored in every object, individually.

We can't change the value of readonly field.

Readonly Fields are accessible with reference variable [with object].

Readonly Fields must be initialized, either "in-line with declaration" [or] "in the constructor".

AccessModifier readonly DataType FieldName = value;



Key Points to Remember

- Fields are variables that are declared in the class; but stored in objects.
- Access modifiers of fields: private, protected, private protected, internal, protected internal, public
- Modifiers of fields: static, const, readonly
- Instance fields are individual for each object; Static fields are common (one-time) for all objects.
- Constants must be initialized along with declaration; Readonly fields must be initialized either 'along with declaration' or in 'instance constructor'.

Methods

Method is a function (group of statements), to do some process based on fields.

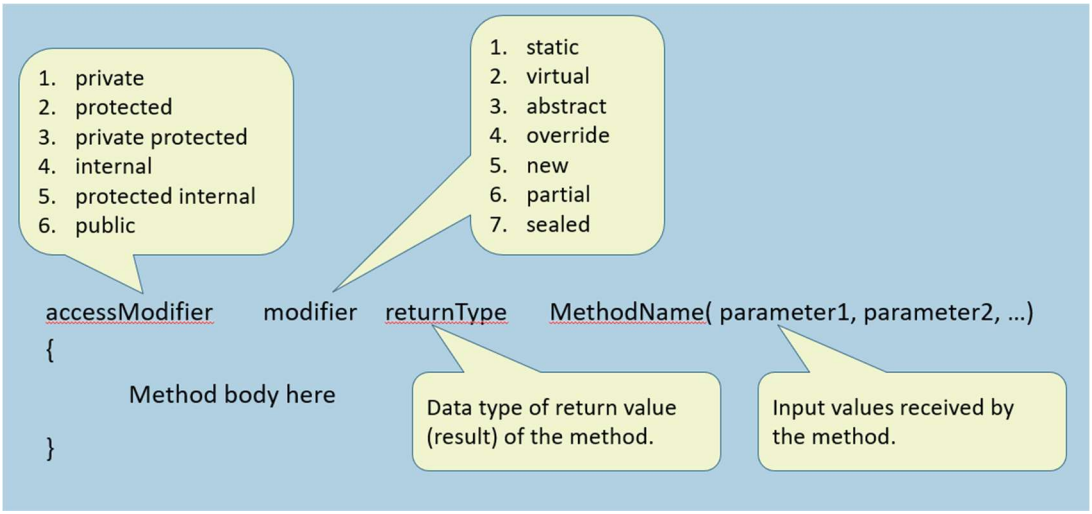
Methods are parts of the class.

Methods can receive one or more input values as "parameters" and return a value as "return".

Eg:

```
1. class Car
2. {
3.   int calculateEmi( int carPrice, int noOfMonths, int interestRate )
4.   {
5.     //do calculation here
6.     return (emi);
7.   }
8. }
```

Syntax of Method



Access Modifiers of Methods

Access Modifiers (a.k.a. "Access Specifier" or "Visibility Modifier) of methods, are same as access modifiers of fields.

Access Modifier	In the same class	In the child classes at the same assembly	In the other classes at the same assembly	Child classes at other assembly	Other classes at other assembly
private	Yes	No	No	No	No
protected	Yes	Yes	No	Yes	No
private protected	Yes	Yes	No	No	No
internal	Yes	Yes	Yes	No	No
protected internal	Yes	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes	Yes

Encapsulation

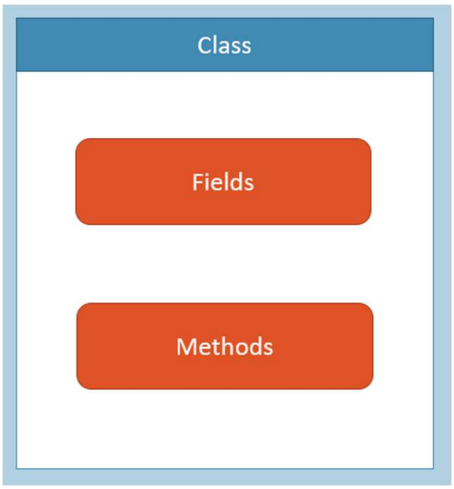
• Encapsulation is a concept that binds together the data and operations that manipulate the data, and that keeps both safe from outside interference and misuse.

• Concept of grouping-up the data and manipulations.

Fields: Store data

Methods: Manipulate fields (data).

Class is used to group-up the "fields" and "methods".

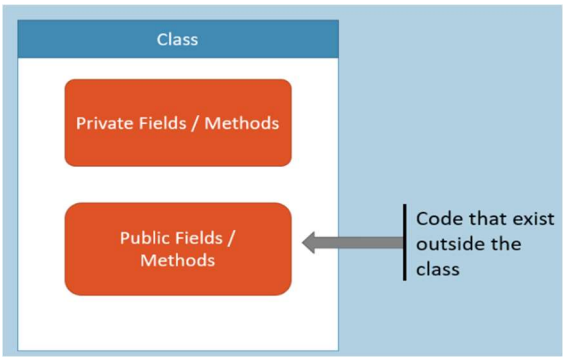


Abstraction

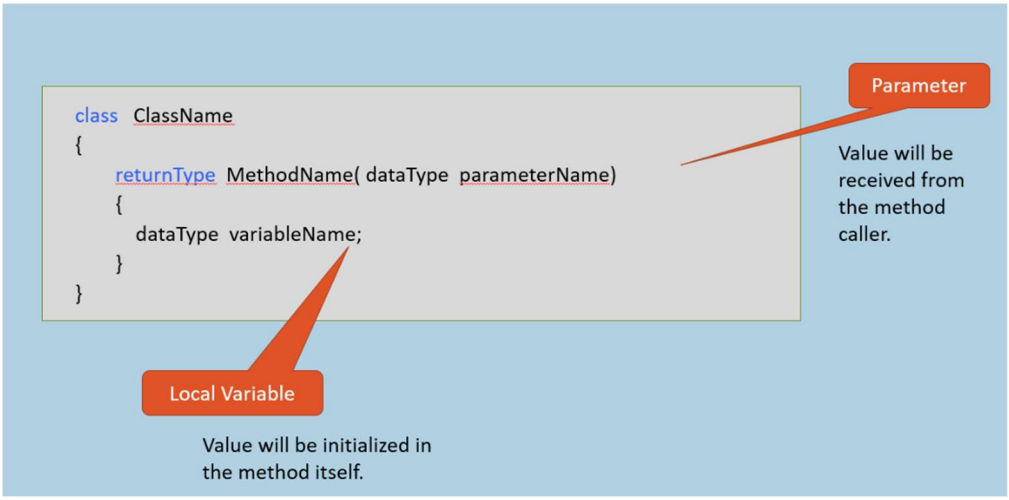
• Abstraction is the concept of providing only limited data or operations to the code exist outside of class.

• Concept of hiding some "private data / operations" and providing some "public data / operations" to the code outside of class.

• Implemented using "private fields / methods" and "public fields / methods".



Local Variables and Parameters



Parameters:

The variables that are being received from the "method caller" are called as "parameters".

The parameters are stored in the Stack of the method.

For every method call, a new stack will be created.

Local Variables:

The variables that are declared inside the method are called as "Local variables". Local variables can be used only within the same method.

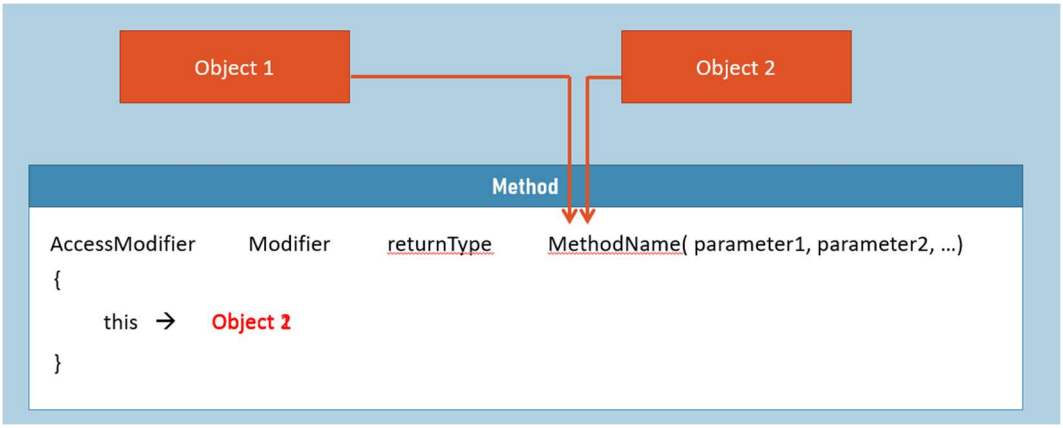
Local variables are stored in the same stack, just like parameters.

The stack will be deleted at the end of method execution. So all local variables and parameters will be deleted.

"this" keyword

The "this" keyword refers to "current object", which method has invoked the method.

The "this" keyword is available only within the "instance methods".



Instance Methods (vs) Static Methods

Association:

Instance Methods: Associated with Objects

Static Methods: Associated with class.

Manipulates:

Instance Methods: Manipulates instance fields.

Static Methods: Manipulates static fields.

Declaration:

Instance Methods: Declared without "static" keyword. Syntax: `returnType methodName( ) { }`

Static Methods: Declared with "static" keyword. Syntax: `static returnType methodName( ) { }`

Accessible with:

Instance Methods: Accessible with object (through reference variable).

Static Methods: Accessible with class name only (not with object).

Can access (fields)

Instance Methods: Can access both instance fields and static fields

Static Methods: Can't access instance fields; but can access static fields only.

Can access (methods)

Instance Methods: Can access both instance methods and static methods.

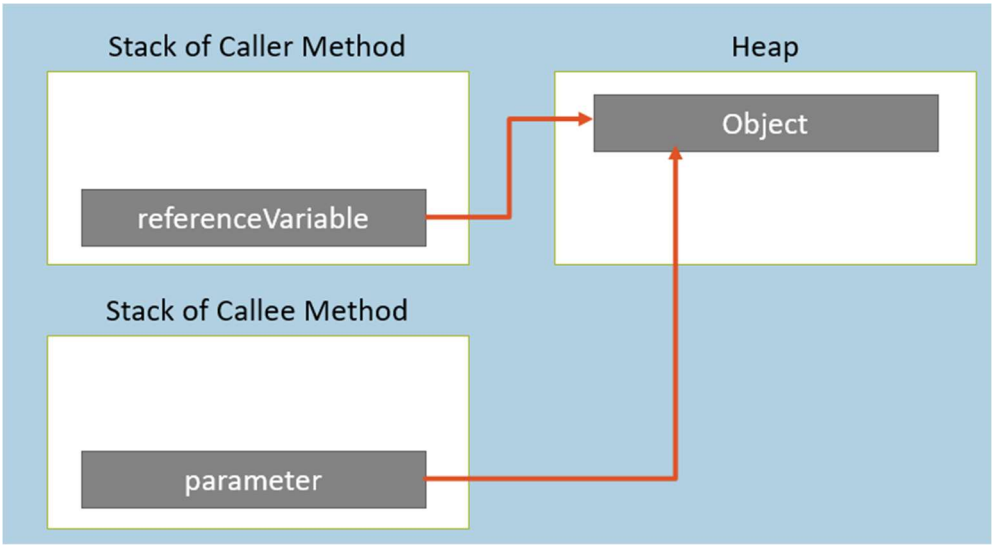
Static Methods: Can't access instance methods; but can access static methods only.

"this" keyword:

Instance Methods: Can use "this" keyword, as there must be "current object" to call instance method.

Static Methods: Can't use "this" keyword, as there is NO "current object" while calling instance methods.

Reference Variables as Arguments



If you pass "reference variable" as argument, the reference (address) of object will be passed to the method.

The parameter's data type will be the class name.

If you make any changes to object in the method, the same will be affected automatically in the caller method, as you are accessing the same object.

Default Arguments

Default value of the parameter of a method.

If you don’t pass value to the parameter, the default value gets assigned to the parameter.

To void bothering to pass value to the parameter; instead, take some default value into the parameter automatically, if the method caller has not supplied value to the parameter.

```
accessModifier   modifier    returnType  MethodName( parameter1 )
{
                 Method body here
}
```

Named Arguments

Supply value to the parameter, based on parameter name.

Syntax: `parametername: value`

You can change order of parameters, while passing arguments.

Parameter names are expressive (understandable) at method-calling time.

Calling a method: `MethodName(ParameterName : value, ParameterName : value);`

Method Overloading

Writing multiple methods with same name in the same class, with different parameters.

Caller would have several options, while calling a method.

Difference between parameters of all the methods that have same name, is MUST.

```
MethodName( )
MethodName( int )
MethodName( string )
MethodName( int, string )
MethodName( string , int)
MethodName( string , string, int)
etc.
```

Parameter Modifiers

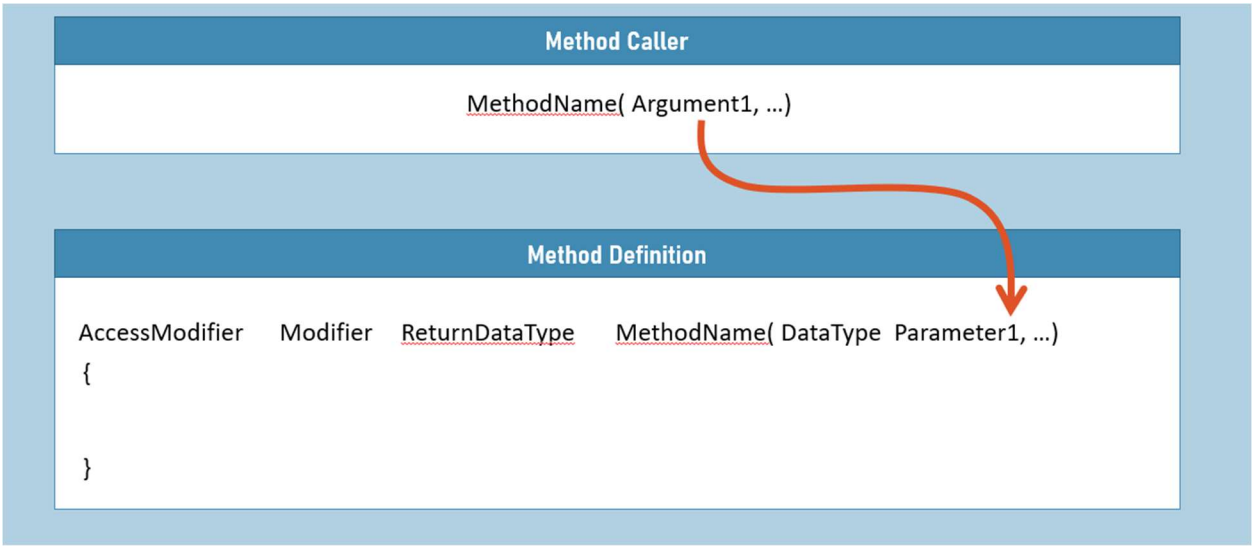
Specifies how the parameter receives a value.

- Default [No keyword]
- ref
- out
- in

- params

Parameter Modifiers (default)

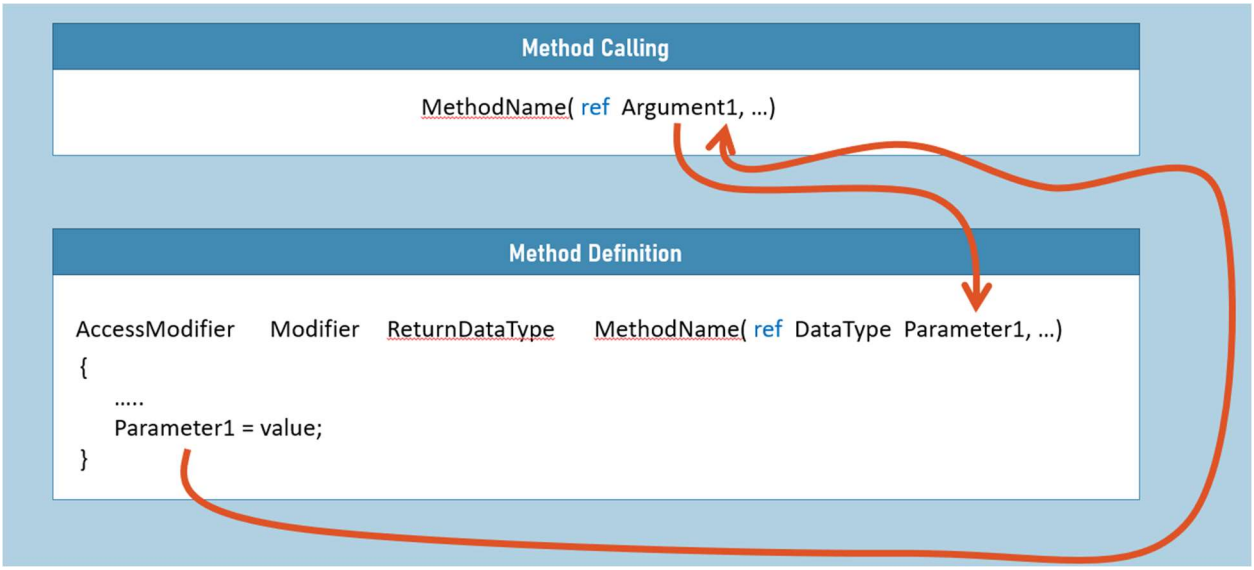
The "Argument" will be assigned into the "Parameter" but not reverse.



Parameter Modifiers (ref)

The "Argument" will be assigned into the "Parameter" and vice versa.

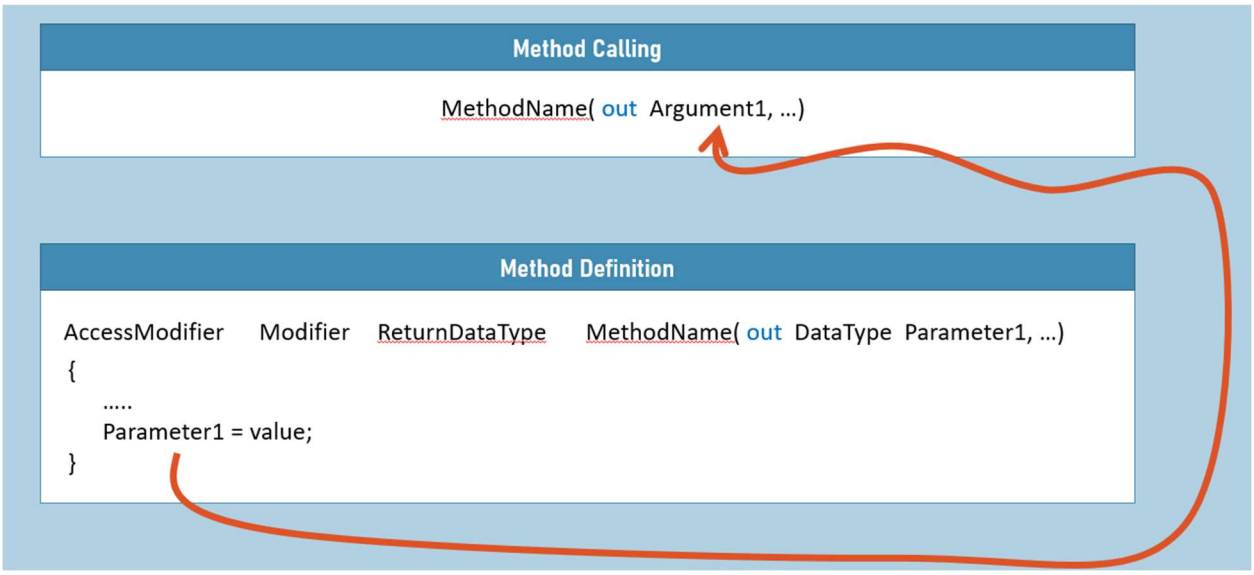
The Argument must be a variable and must be pre-initialized.



Parameter Modifiers (out)

The "Argument" will not be assigned into the "Parameter" but only reverse.

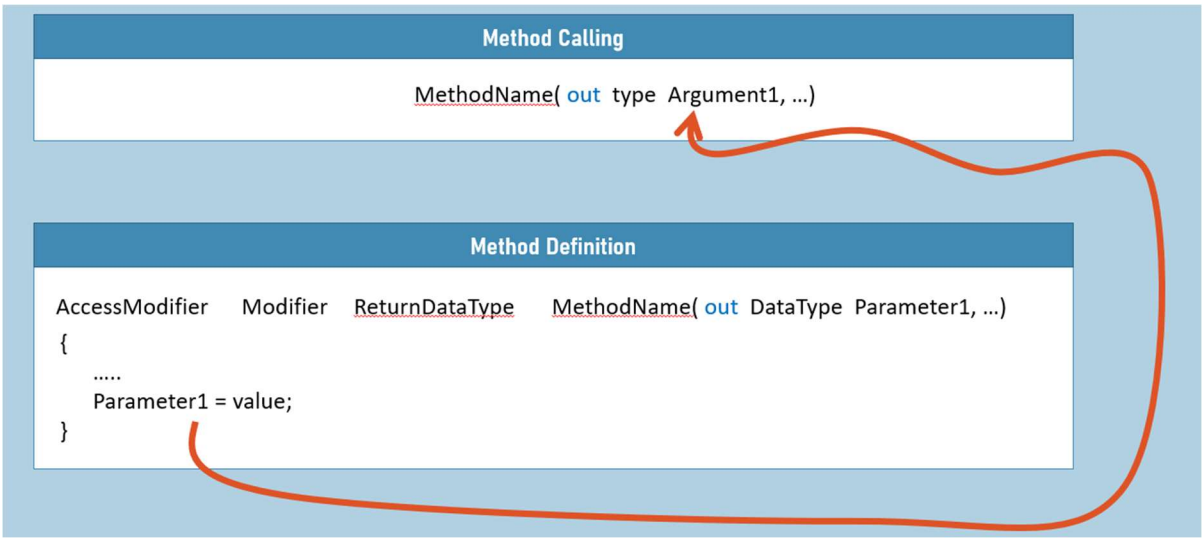
The Argument must be a variable; The Argument can be un-initialized.



'out' variable declaration

You can declare out variable directly while calling the method with 'out' parameter.

New feature in C# 7.0.

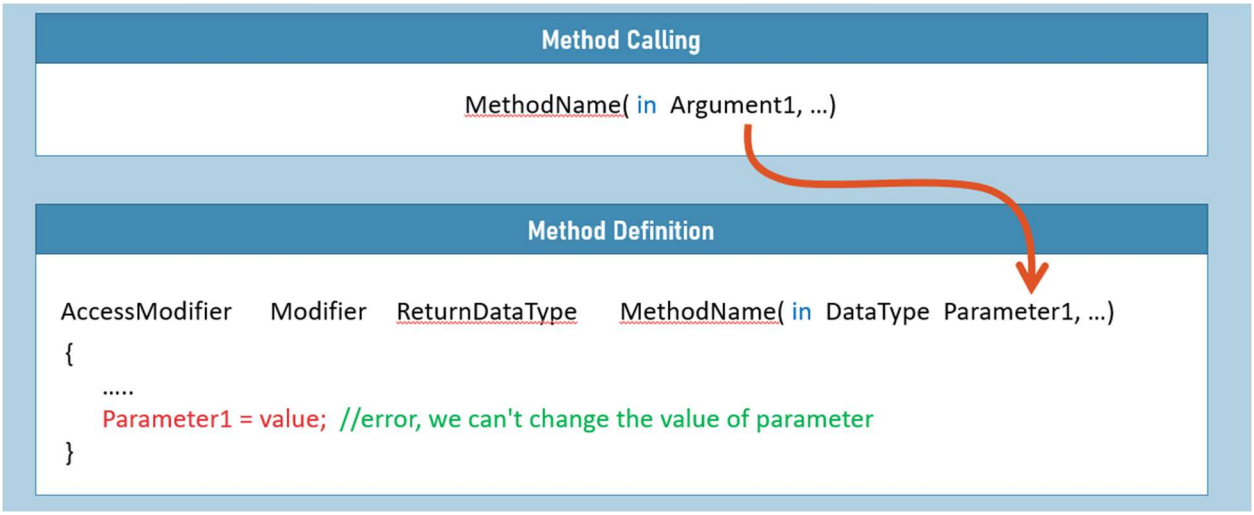


Parameter Modifiers (in)

The "Argument" will be assigned into the "Parameter", but the parameter becomes read-only.

We can't modify the value of parameter in the method; if you try to change, compile-time error will be shown.

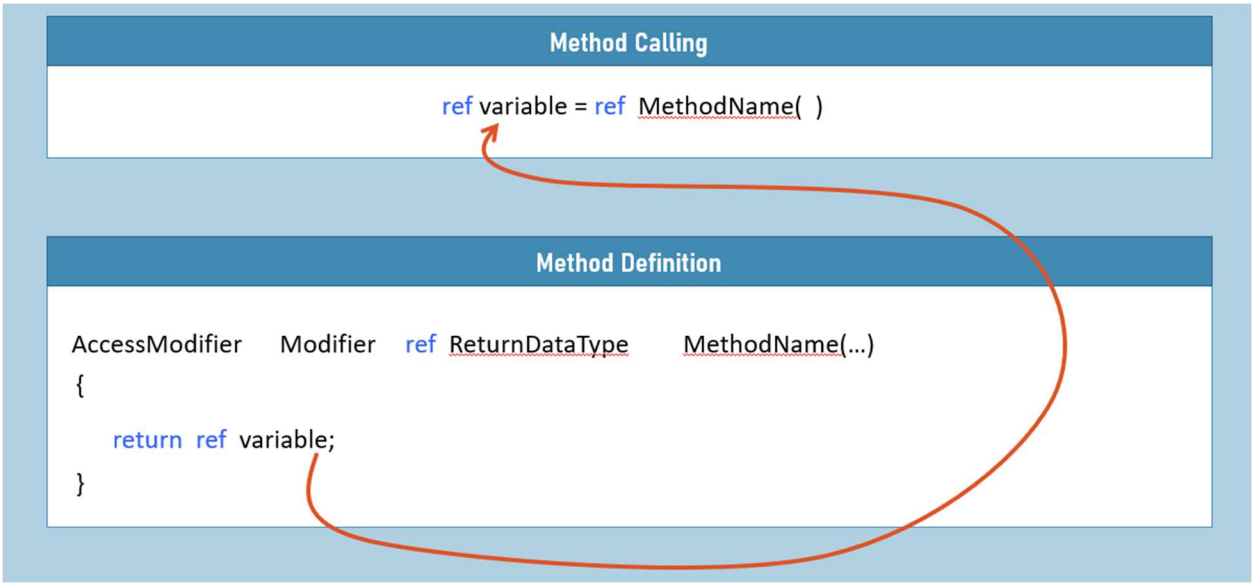
New feature of C# 7.2.



ref returns

The reference of return variable will be assigned to receiving variable.

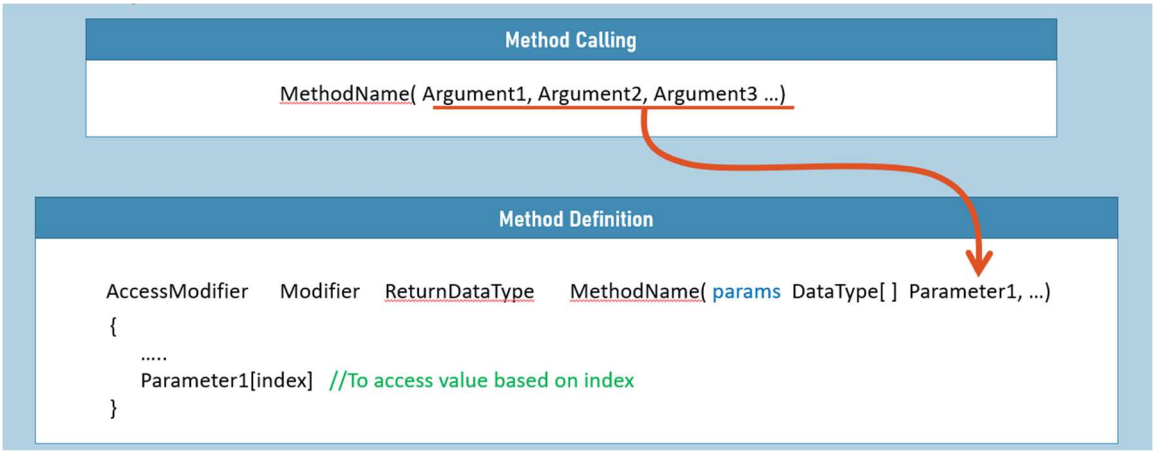
New feature in C# 7.3.



Parameter Modifiers (params)

All the set of arguments will be at-a-time received as an array into the parameter.

The "params" parameter modifier can be used only for the last parameter of the method; and can be used only once for one method.



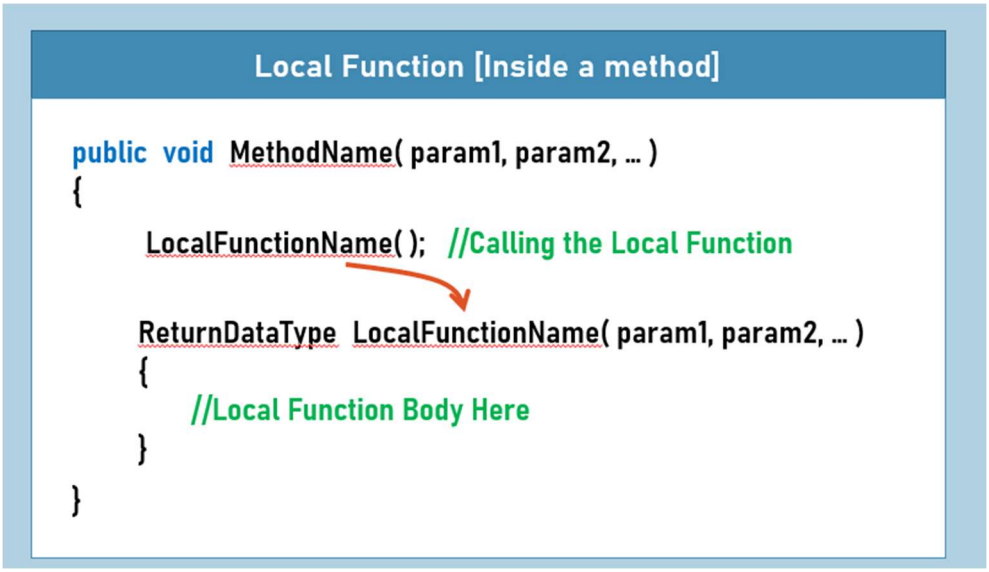
Local Functions

"Local functions" are functions, to do some small process, which is written inside a method.

Local functions are not part of the class; they can't be called directly through reference variable.

Local functions don't support "access modifiers" and "modifiers".

Local functions support parameters, return.

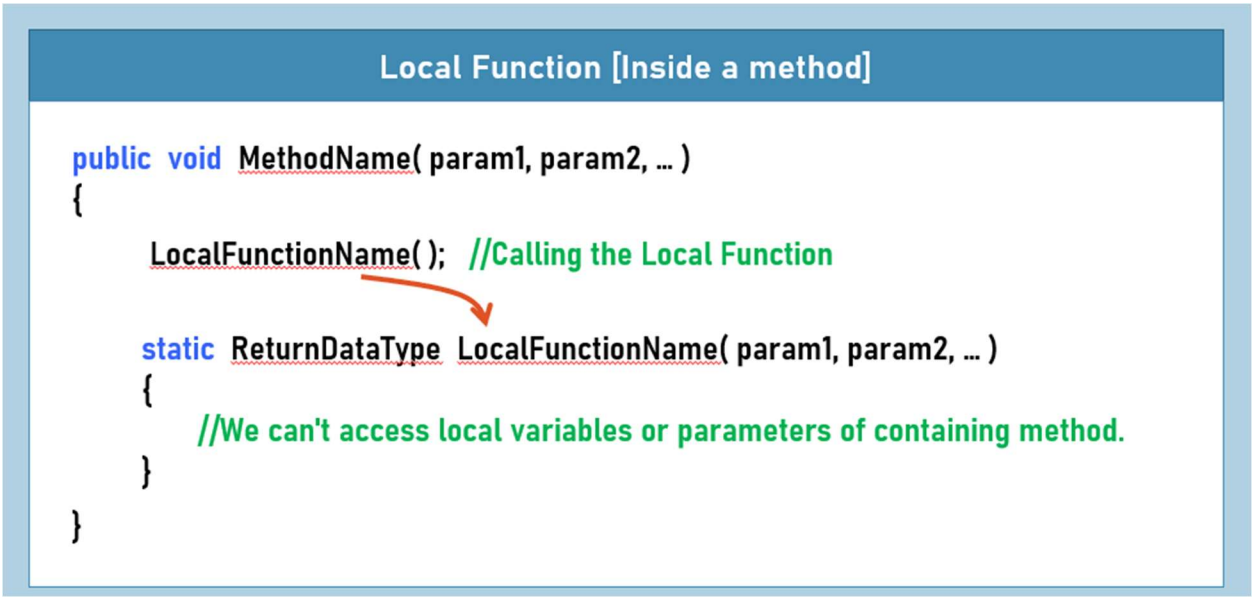


Static Local Functions

"Static Local functions" are functions, same as normal "Local Functions".

Only the difference is, static local functions can't access local variables or parameters of containing method.

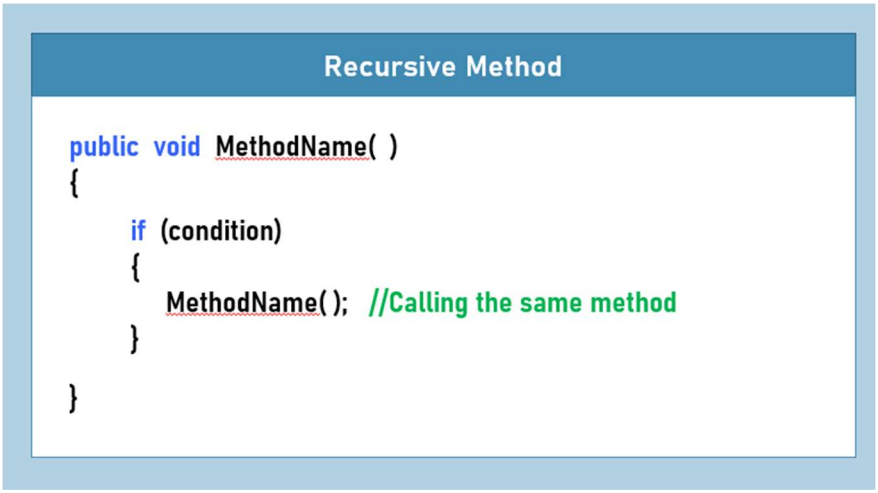
This is to avoid accidental access of local variables or parameters of containing method, inside the local function.



Recursion

A method calls itself.

Useful in mathematic computations, such as finding factorial of a number.



Key Points to Remember

- Method is a part of class, that contains collection of statements to do some process.
- Access modifiers of Methods: private, protected, private protected, internal, protected internal, public.
- Modifiers of Methods: static, virtual, abstract, override, new, partial, sealed
- For each method call, a new stack will be created; all local variables and parameters of the method will be stored in that stack; will be deleted automatically at the end of method execution.
- In instance methods, the 'this' keyword refers to 'current object, based on which the method is called'.
- Instance methods can access & manipulate instance fields & static fields; Static methods can access only static fields.
- But static method can create an object for the class; then access instance fields through that object.
- Using named arguments , you can change order of parameters while calling the method.
- Method Overloading is 'writing multiple methods with same name in the same class with different set of parameters'.
- The 'ref' parameter is used to receive value into the method and also return some value back to the method caller; The 'out' parameter is only used to return value back to the method caller; but not for receiving value into the method.

Type Conversion

'Type Conversion' is a process of convert a value from one type (source type) to another type (destination type).

Eg: int -> long

1. Implicit Casting

(from lower-numerical-type to higher-numerical-type)

2. Explicit Casting

(from higher-numerical-type to lower-numerical-type)

3. Parsing / TryParse

(from string to numerical-type)

4. Conversion Methods

(from any-primitive-type to any-primitive-type and also string along with other types such as DateTime, Base64 etc.)

Implicit Casting

The 'lower-numerical type' can be automatically (implicitly) converted into 'higher-numerical type'.

Conversion From		Conversion To
sbyte	→	short, int, long, float, double, decimal
byte	→	short, ushort, int, uint, long, ulong, float, double, decimal
short	→	int, long, float, double, decimal
ushort	→	int, uint, long, ulong, float, double, decimal
int	→	long, float, double, decimal
uint	→	long, ulong, float, double, decimal
long	→	float, double, decimal
ulong	→	float, double, decimal
float	→	double
double	→	[none]
decimal	→	[none]
char	→	ushort, int, uint, long, ulong, float, double, decimal
bool	→	[none]
string	→	[none]

Explicit Casting

We can manually convert a value from one data type to another data type, by specifying the destination data type within brackets, at left-hand-side of the source value.

Loosy conversion: If the destination type is not sufficient-enough to store the converted value, the value may loose.

Syntax: `(DestinationDataType)SourceValue`

1. At all cases in the table of implicit casting.

2. At the case in the following table of explicit casting.

3. Child class to Parent class.

Conversion From		Conversion To
sbyte	→	byte, ushort, uint, ulong
byte	→	sbyte
short	→	sbyte, byte, ushort, uint, ulong
ushort	→	sbyte, byte, short
int	→	sbyte, byte, short, ushort, uint, ulong
uint	→	sbyte, byte, short, ushort, int
long	→	sbyte, byte, short, ushort, int, uint, ulong
ulong	→	sbyte, byte, short, ushort, int, uint, long
float	→	sbyte, byte, short, ushort, int, uint, long, ulong, decimal
double	→	sbyte, byte, short, ushort, int, uint, long, ulong, float, decimal
decimal	→	sbyte, byte, short, ushort, int, uint, long, ulong, float, double
char	→	sbyte, byte, short, ushort, int, uint, long, ulong, float, double, decimal
bool	→	[none]
string	→	[none]

Parse

The string value can be converted into any numerical data type, by using "Parsing" technique.

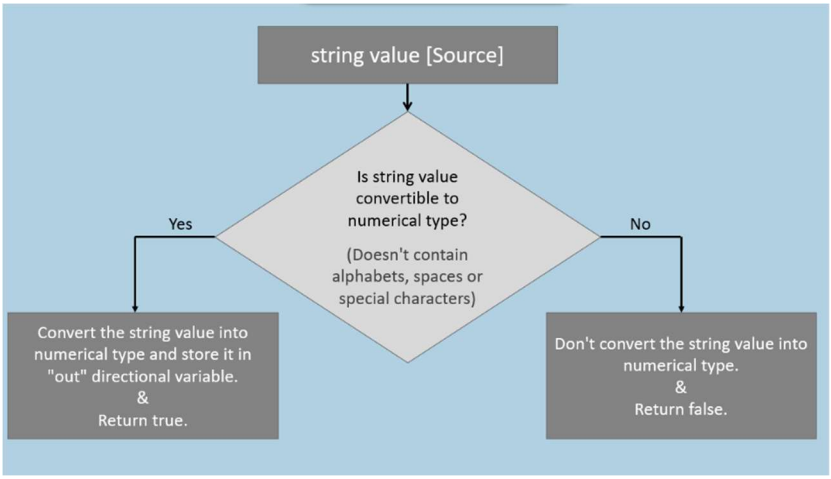
Eg: string à int

The source value must contain digits only; shouldn't contain spaces, alphabets or special characters.

If the source value is invalid, it raises FormatException.

Syntax: `DestinationDataType.Parse(SourceValue)`

TryParse



The string value can be converted into any numerical data type, by using "TryParse" technique (same as "parse"); but it checks the source value before attempting to parse.

Eg: string -> int

If the source value is invalid, it returns false; It doesn't raise any exception in this case.

If the source value is valid, it returns true [indicates conversion is successful]

It avoids FormatException.

```
bool variable = DestinationType.TryParse(SourceValue,
out DestinationVariable)
```

Conversion Methods

Conversion method is a pre-defined method, which converts any primitive type (and also 'string') to any other primitive type (and also 'string').

Eg: string -> int and int -> string

The System.Convert is a class, which contains a set of pre-defined methods.

It raises FormatException, if the source value is invalid.

For each data type, we have a conversion method.

All conversion methods are static methods.

Syntax:

```
type destinationVariable =
Convert.ConversionMethod (SourceValue )
```

Conversion To	Conversion Method
sbyte	System.Convert.ToSByte(value)
byte	System.Convert.ToByte(value)
short	System.Convert.ToInt16(value)
ushort	System.Convert.ToUInt16(value)
int	System.Convert.ToInt32(value)
uint	System.Convert.ToUInt32(value)
long	System.Convert.ToInt64(value)
ulong	System.Convert.ToUInt64(value)
float	System.Convert.ToSingle(value)
double	System.Convert.ToDouble(value)
decimal	System.Convert.ToDecimal(value)
char	System.Convert.ToChar(value)
string	System.Convert.ToString(value)
bool	System.Convert.ToBoolean(value)

Key Points to Remember

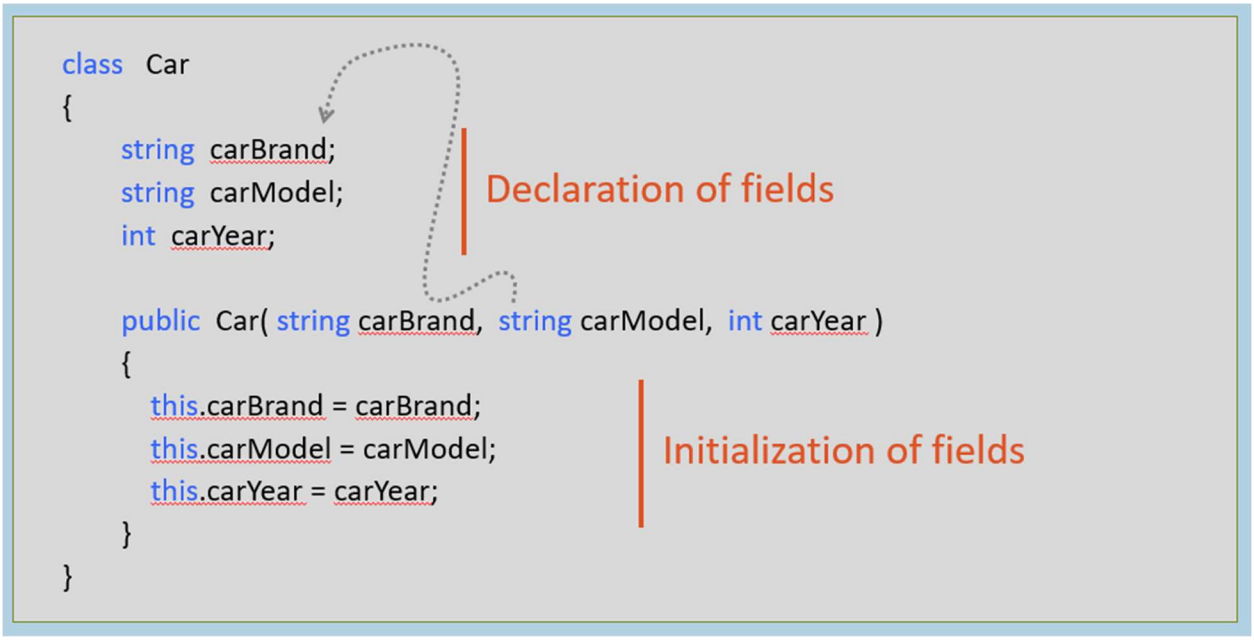
- For all the possible cases of 'implicit casting' and 'explicit casting', it is preferred to use 'explicit casting' or 'conversion methods' always.
- For conversion from 'string' to 'numerical type', use TryParse, instead of 'Parse'; as 'TryParse' avoids exceptions.
- For conversion of value from any-type to any-type, use conversion method.

Constructors

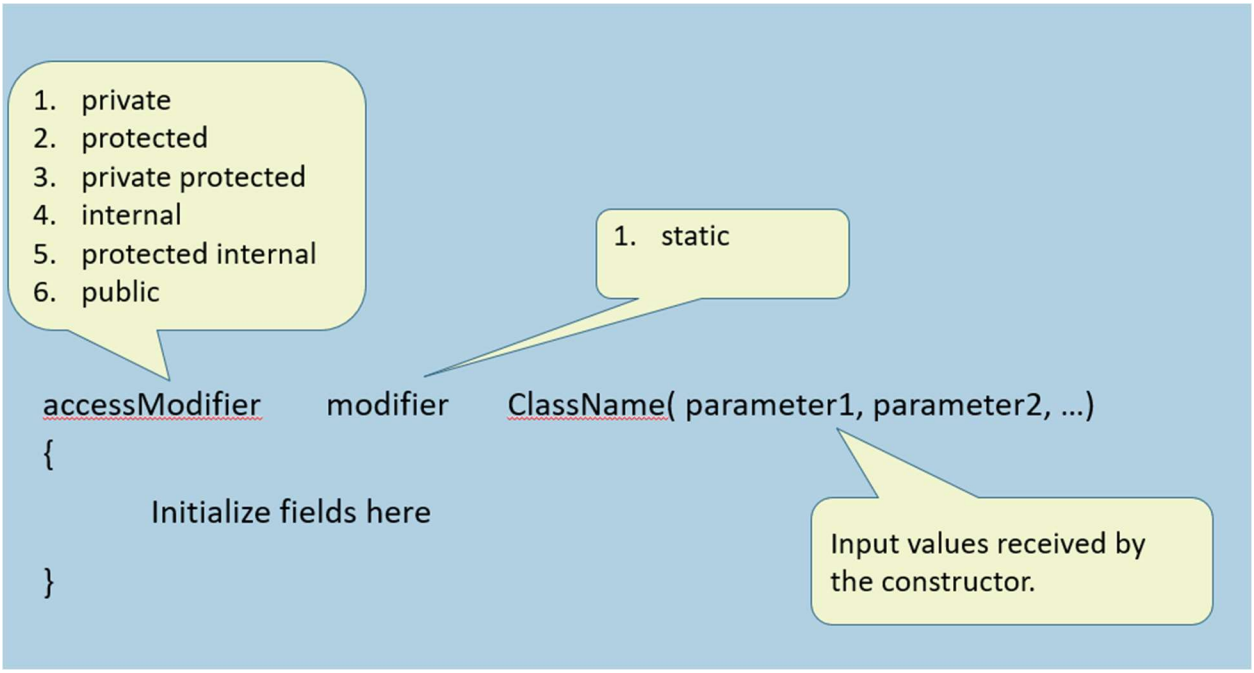
Special method of class, which contains initialization logic of fields.

Constructor initializes the fields and also contains the additional initialization logic (if any).

Eg:



Syntax of Constructor



Rules of Constructors

- Constructor's name should be same as class name.
- Constructor is recommended to be "public" member or "internal" member;
- if it is a "private member", it can be called within the same class only; so you can create object of a class only inside the same class; but not outside the class.
- Constructor can have one or more parameters.
- Constructor can't return any value; so no return type.
- A class can have one or more constructors; but all the constructors of the class must have different types of parameters.

Instance (vs) Static Constructor

Instance Constructor

```
public ClassName( Parameter1, Parameter2, … )
{
    …
}
```

1. Initializes instance fields.
2. Executes automatically every time when a new object is created for the class.
3. "private" by default; We can use any of access modifiers.
4. Can contain any initialization logic, that should be executed every time when a new object is created for the class.

Static Constructor

```
static ClassName( )
{
    …
}
```

1. Initializes static fields.
2. Executes only once, i.e. when first object is created for the class or when the class is accessed for the first time during the execution of Main method.
3. "public" by default; Access modifier can't be changed.
4. Can contain any initialization logic, that should be executed only once i.e. when a new object is created for the class.

Parameter-less (vs) Parameterized Constructor

Parameter-less Constructor

```
public ClassName( )
{
```

```
3.   …
4. }
```

1. Constructor without parameters.
2. It generally initializes fields with some literal values (or) contains some general-initialization logic of object.

Parameterized Constructor

```
1. public ClassName( Parameter1, Parameter2, … )
2. {
3.   …
4. }
```

1. Constructor with one or more parameters.
2. It generally initializes fields by assigning values of parameters into fields.

Implicit (vs) Explicit Constructor

Implicit Constructor (after compilation)

```
1. public ClassName( )
2. {
3. }
```

1. If there is a class without constructor, then the constructor automatically provides an empty constructor, while compilation, which initializes nothing. It is called as "Implicit Constructor" or "Default Constructor".
2. It is just to satisfy the rule "Class should have a constructor".

Explicit Constructor (While coding)

```
1. public ClassName( with or without parameters )
2. {
3.   …
4. }
```

1. The constructor (parameter-less or parameterized) while is created by the developer is called as "Explicit Constructor".
2. In this case, the C# compiler doesn't provide any implicit constructor.

Constructor Overloading

Write multiple constructors with same name in the class, with different set of parameters (just like 'method overloading').

It is recommended to write a parameter-less constructor in the class, in case of constructor overloading.

Constructor Overloading (multiple constructors in the same class)

```
1. public ClassName( )
2. {
3. }
4.
5. public ClassName( parameter1, parameter2, … )
6. {
7.    …
8. }
```

Object Initializer

Special syntax to initialize fields / properties of class, along with creating the object.

Executes after the constructor.

It is only for initialization of fields / properties, after creating object; it can't have any initialization logic.

Execution:


```
new ClassName( ) { field1 = value, field2 = value, … }
```

Use 'object initializer' when:

1. there is no constructor present in the class; but you want to initialize fields / properties.
2. (or) there is a constructor; but it is meant for initializing other set of fields, other than the fields that you want to initialize.

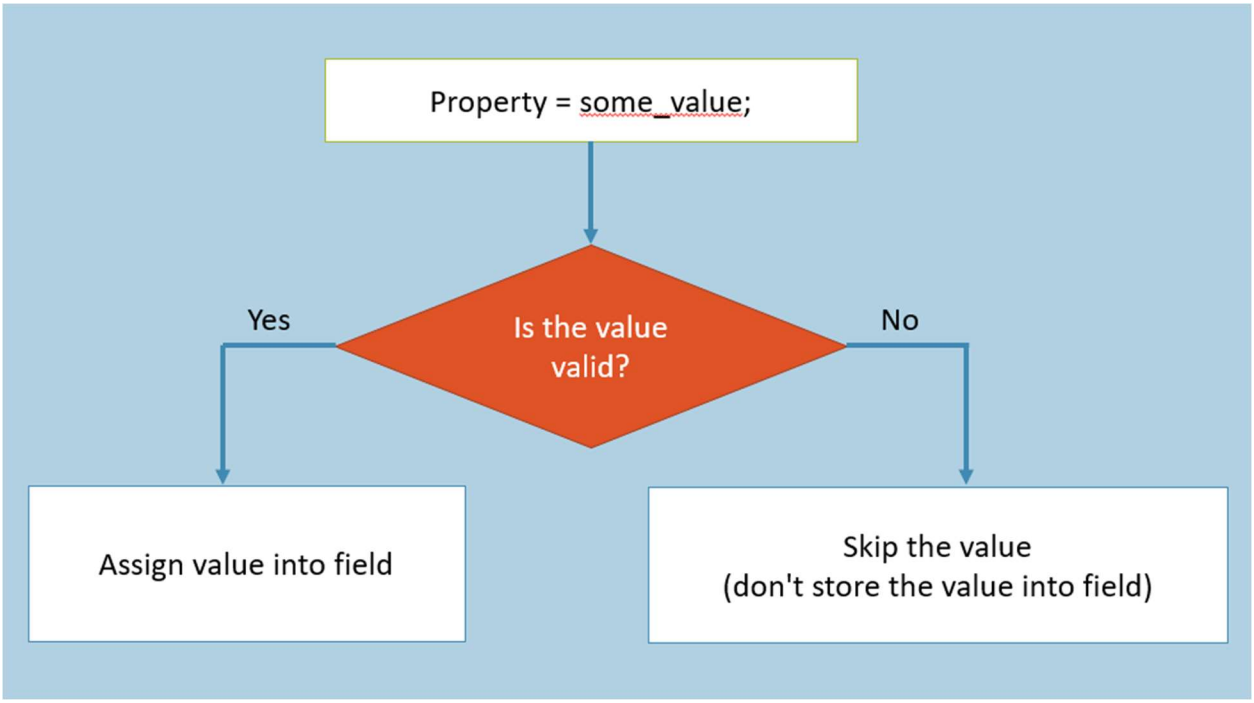
Key Points to Remember

1. 'Instance constructor' initializes 'instance fields'; but also can access 'static fields'.
2. 'Static constructor' initializes 'static fields'; can't access 'instance fields'.
3. Default (empty constructor) is provided automatically by C# compiler, if the developer creates a class without any constructor.
4. It is always recommended to write a parameter-less constructor first, if you are creating parameterized constructor.

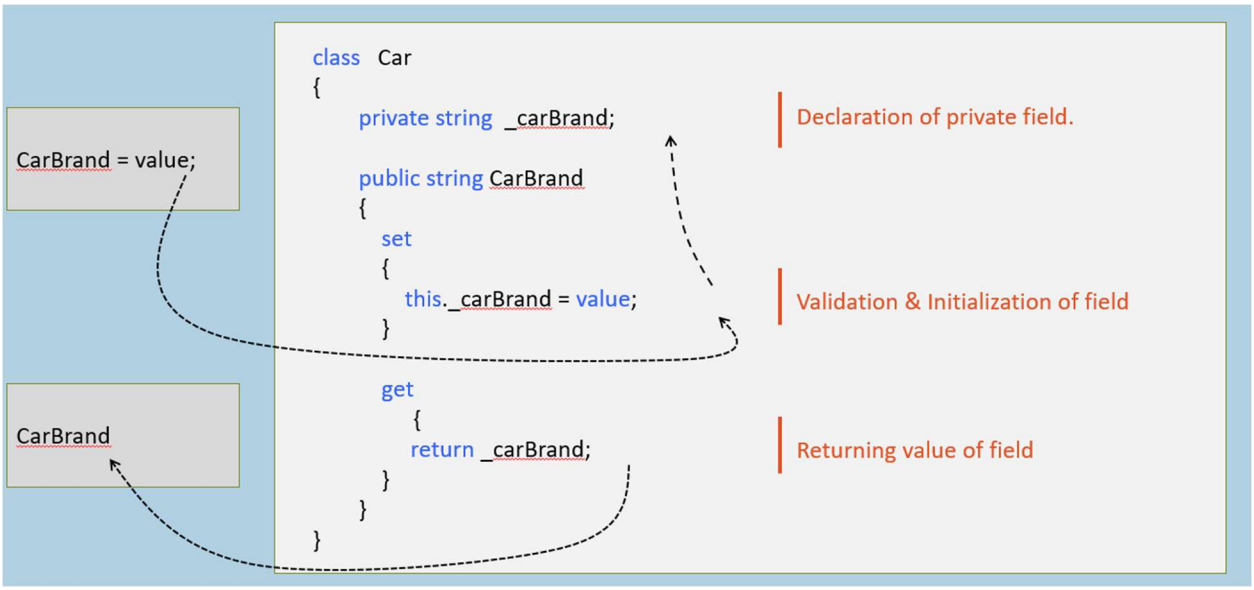
5. Use 'object initializer', if you want to initialize desired fields of an object, as soon as a new object is created.

Properties

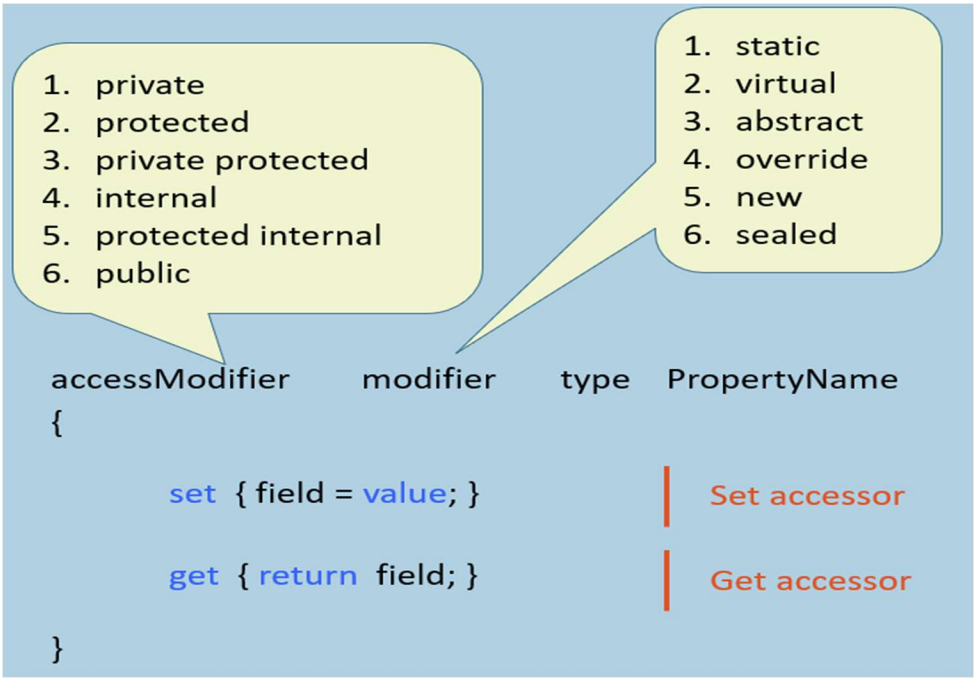
Receive the incoming value; validate the value; assign value into field.



Property is a collection of two accessors (get-accessor and set-accessor).



Syntax of Property



Set Accessor [vs] Get Accessor

Set Accessor

```
1. set
2. {
```

```
3.   field = value;
4. }
```

1. Used to validate the incoming value and assign the same into field.
2. Executes automatically when some value is assigned into the property.
3. Has a default (implicit) parameter called "value", which represents current value i.e. assigned to the property.
4. Can't have any additional parameters.
5. But can't return any value.

Get Accessor

```
1. get
2. {
3.   return field;
4. }
```

1. Used to calculate value and return the same (or) return the value of field as-it-is.
2. Executes automatically when the property is retrieved.
3. Has no implicit parameters.
4. Can't have parameters.
5. Should return value of field.

Features and Advantages of Properties

Properties create a protection layer around fields, preventing assignment of invalid values into properties & also do some calculation automatically when someone has invoked the property.

No memory will be allocated for the property.

Access modifier of accessors:

Access modifier is applicable for the property, set accessor and get accessor individually. But access modifiers of accessors must be more restrictive than access modifier of property.

Eg:

```
1. internal  modifier  data_type  PropertyName
2. {
3.      private  set { property = value; }
4.      protected  get { return property; }
5. }
```

Read-Only [vs] Write-Only Properties

Readonly Property

```
accessModifier type PropertyName
{
  get
  {
    return field;
  }
}
```

1. Contains ONLY 'get' accessor
2. Reads & returns the value of field; but not modifies the value of field.

Write-only Property

```
accessModifier type PropertyName
{
  set
  {
    field = value;
  }
}
```

1. Contains ONLY 'set' accessor.
2. Validates & assigns incoming value into field; but doesn't return the value.

Auto-Implemented Properties

Property with no definition for set-accessor and get-accessor.

Used to create property easily (with shorter syntax).

Creates a private field (with name as _propertyName) automatically, while compilation-time.

Auto-Implemented property can be Read-only (only 'get' accessor) property; but it can't be Write-only (only 'set' accessor).

```
accessModifier  modifier  data_type  PropertyName
{
     accessModifier  set;
     accessModifier  get;
}
```

Useful only when you don't want to write any validation or calculation logic.

Auto-Implemented Property Initializer

New feature in C# 6.0

You can initialize value into auto-implemented property.

```
accessModifier    modifier  type  propertyName  { set;  get; } = value;
```

Properties: Key Points to Remember

It is recommended to use Properties always in real-time projects.

You can also use 'Auto-implemented properties' to simplify the code.

Properties doesn't occupy any memory (will not be stored).

Properties forms a protection layer surrounding the private field that validates the incoming value before assigning into field.

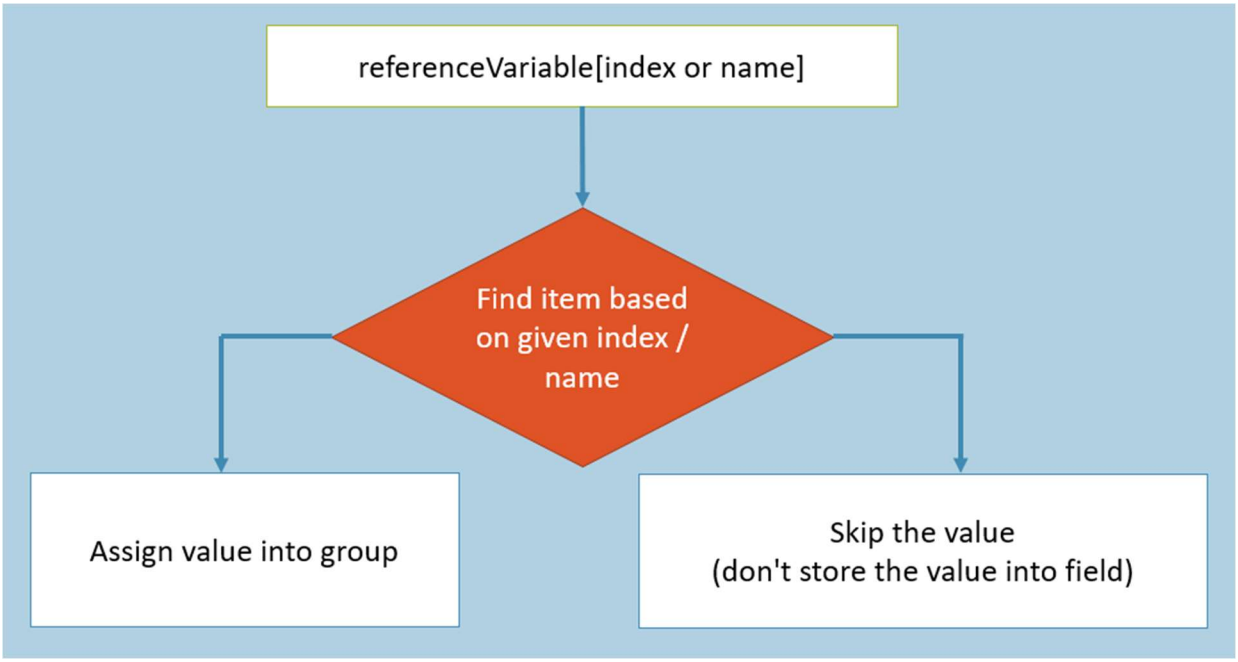
Read-only property has only 'get' accessor; Write-only property has only 'set' accessor.

Properties can't have additional parameters.

Indexers

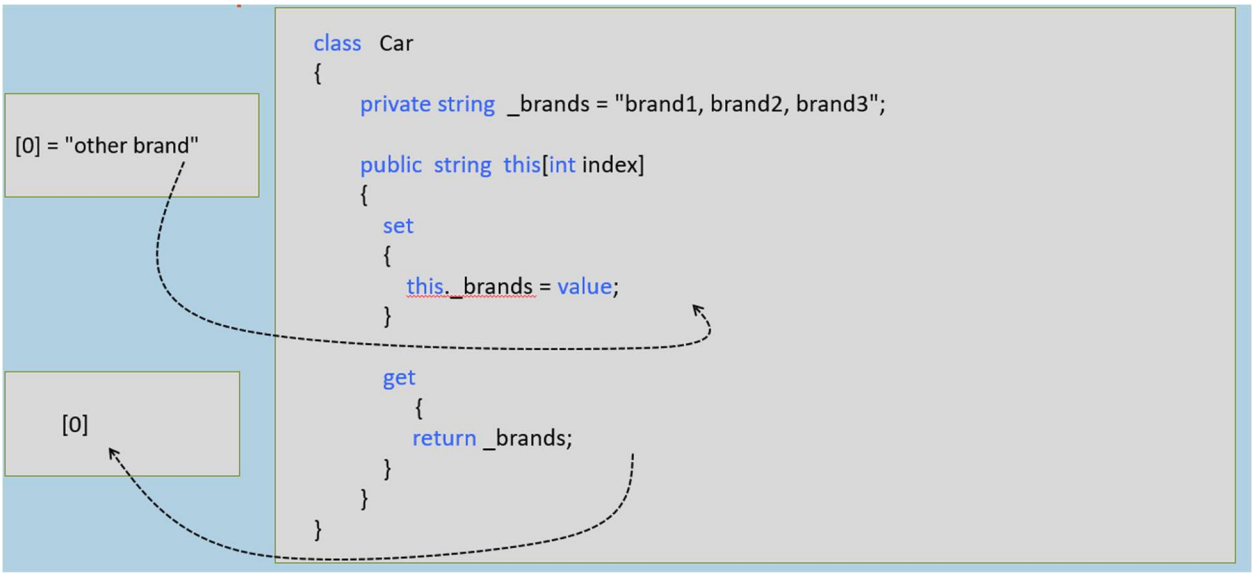
Receive a number / string. Search for the particular item among a group of items; set or get value into the group of items.

It provides shorter syntax to access a group of items.

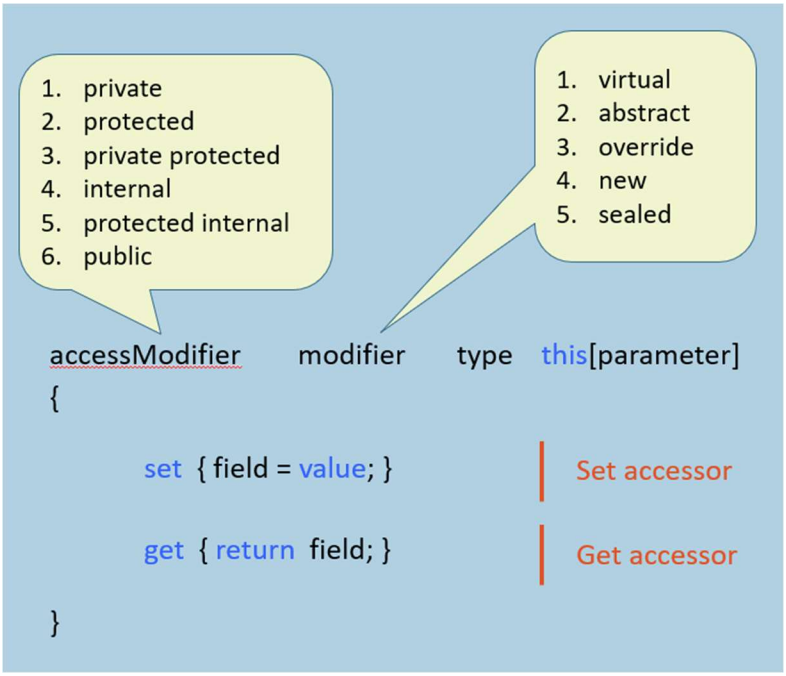


Indexer is a special member of class, which contains set-accessor and get-accessor to access a group of items / elements.

Eg:



Syntax of Indexer

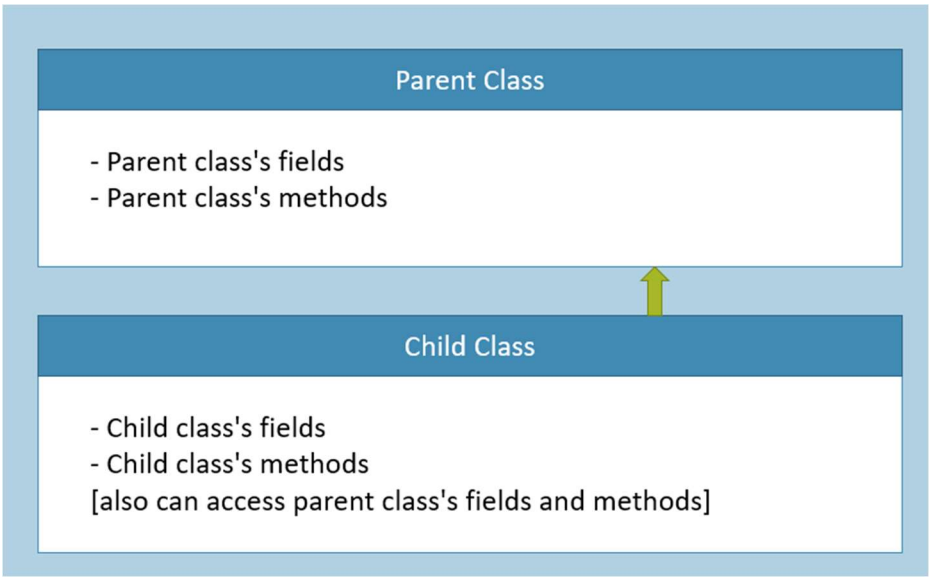


Indexers: Key Points to Remember

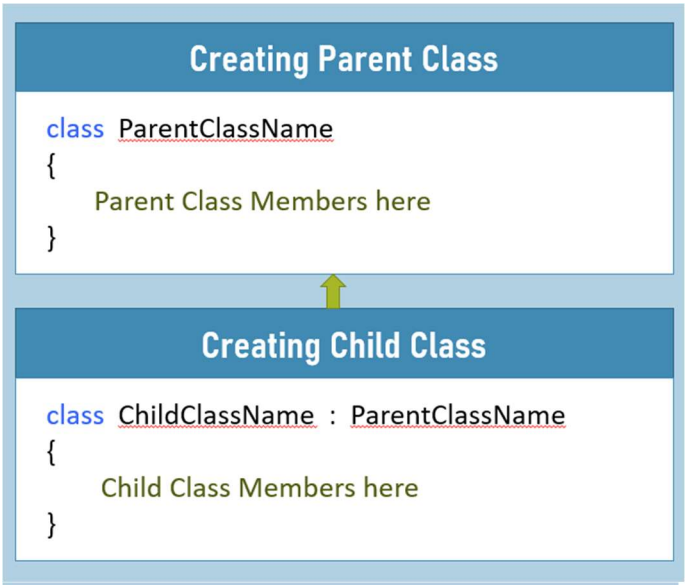
- Indexers are always created with 'this' keyword.
- Indexers are generally used to access group of elements (items).
- Parameterized properties are called indexer.
- Indexers are implemented through get and set accessors along with the [ ] operator.
- Indexer must have one or more parameters.
- ref and out parameter modifiers are not permitted in indexer.
- Indexer can't be static.
- Indexer is identified by its signature (syntax of calling); where as a property is identified it's name.
- Indexer can be overloaded.

Inheritance

- This allows classes to be arranged in a hierarchy that represents "is-a-type-of" relationships.
- The "parent class" acts as a "base type" of "one or more child classes".
- Child classes are derived from parent class.



- Concept of extending the parent class, by creating child class.
- "Child class" extends "parent class".
- The child class's object stores members of both child class and parent class.



Types of Inheritance

1. Single Inheritance

```
1. class ParentClassName
2. {
3. }
4.
5. class ChildClassName : ParentClassName
6. {
7. }
```

One Parent Class, One Child Class.

2. Multi-Level Inheritance

```
1. class ParentClassName
2. {
3. }
4.
5. class ChildClass1 : ParentClassName
6. {
7. }
8.
9. class ChildClass2 : ChildClass1
10.{
11.}
```

One Parent Class, One Child Class; and the Child class has another Child class.

3. Hierarchical Inheritance

```
1. class ParentClassName
2. {
3. }
4.
5. class ChildClass1 : ParentClassName
6. {
7. }
8.
9. class ChildClass2 : ParentClassName
10.{
11.}
```

One Parent Class, Multiple Child Classes.

4. Multiple Inheritance

```
class ParentClass1
{
}

class ParentClass2
{
}

class ChildClass : ParentClass1, ParentClass2
{
}
```

Multiple Parent Classes, One Child class.

5. Hybrid Inheritance

```
class ParentClassName
{
}

class ChildClass1 : ParentClassName
{
}

class ChildClass2 : ChildClass1
{
}

class ChildClass3 : ChildClass1
{
}
```

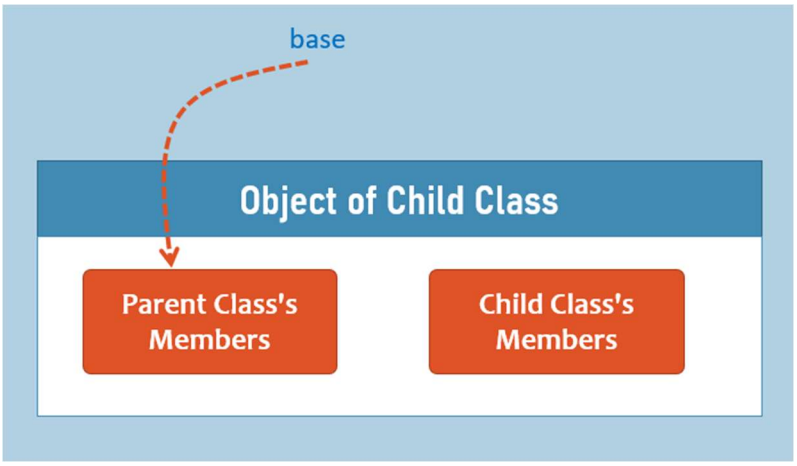
Hierarchical Inheritance + Multi Level Inheritance

'base' keyword

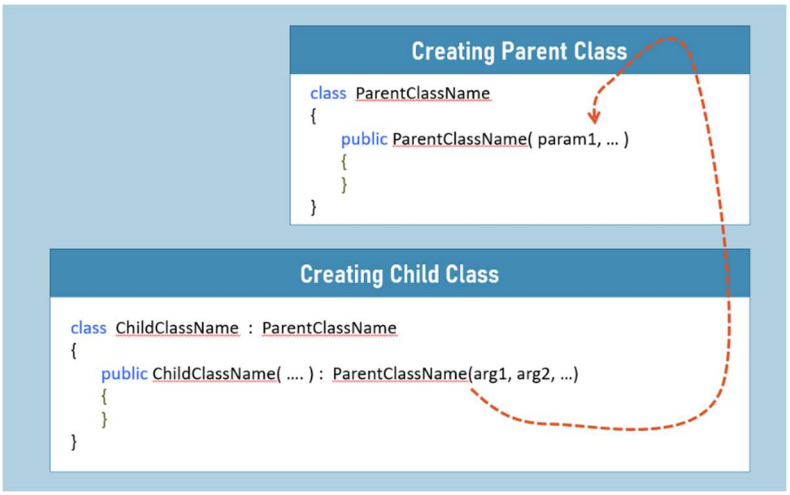
The "base" keyword represents parent class's members in the child class.

It is optional to use, by default.

It is must to use, when there is "name ambiguity" between parent class's member and child class's member.



Parent Class's Constructor

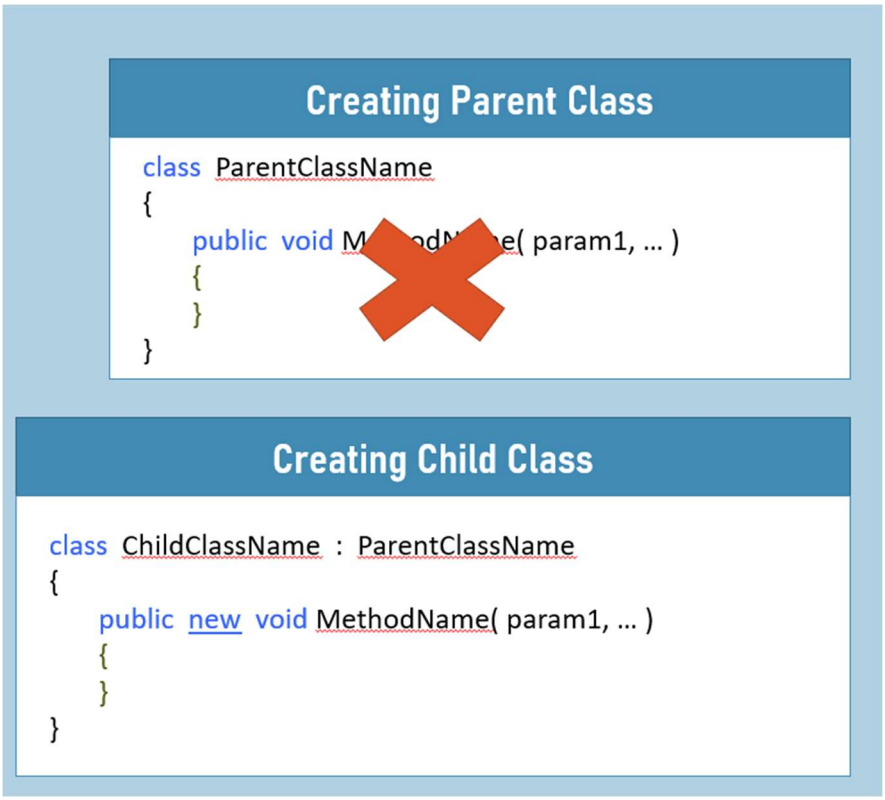


It is OPTIONAL to call "Parent Class's Parameter-less Constructor" from "Child Class".

It is MUST to call "Parent Class's Parameterized Constructor" from "Child Class" and pass necessary arguments.

Method Hiding

It is a concept, which is used to hide (overwrite) the parent class's method, by creating another method in the child class with same name and same parameters.

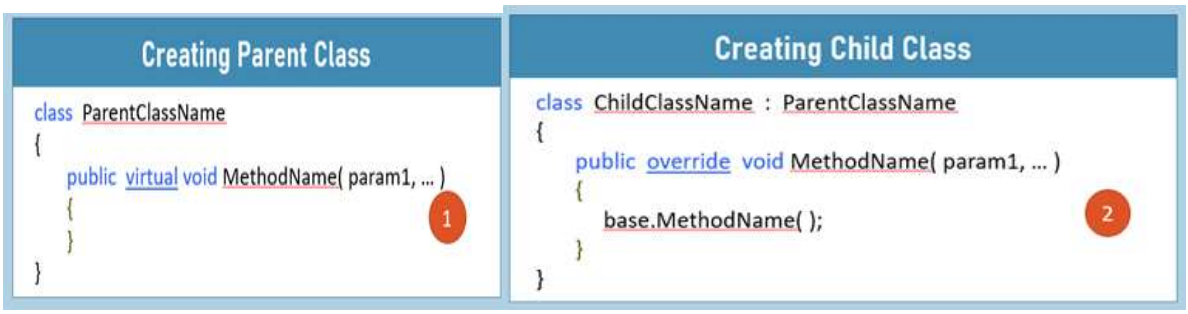


When method hiding is done, if the method is called using child class's object; the child class's method only executes; parent class's method will not be executed.

Method hiding is done automatically; but is recommended to use "new" keyword (but not must).

Method Overriding

It is a concept, which is used to extend the parent class's method, by creating another method in the child class with same name and same parameters.



When method overriding is done, if the method is called using child class's object; the parent class's method first and child's method executed next.

Method Overriding is done with "virtual" keyword at parent class; and "override" keyword at child class's method.

The parent class's method invoked using "base" keyword.

Without 'virtual' keyword are parent class's method; the child class's method can't be 'override'.

Key Points to Remember

- The "parent class" acts as a "base type" of "one or more child classes".
- The 'base' keyword inside the child class, can access the members of parent class.
- The child class's object stores parent class's fields also, automatically; however they are accessible in child class if they are not 'private'.
- Method hiding is a concept of 'overwriting' the parent class's method, by creating another method in child class, with same signature.
- Method overriding is a concept of 'extending' the parent class's method, by creating another method in child class, with same signature.
- C# doesn't support multiple inheritance (with multiple parent classes). That means, a child class can have ONLY ONE parent class; however, a child class can have MULTIPLE parent interfaces; so in this way, C# supports multiple inheritance.

Sealed Classes

Sealed class is a class, which is instantiable; but not inheritable.

Use sealed class, whenever you don't want to let other developers to create child classes for the specific class.

```
1. sealed class Class1
2. {
3. }
4.
5. class Class2 : Class1 //not possible
6. {
7. }
```

Comparison Table: Class [vs] Sealed Class

Class Type	Can Inherit from Other Classes	Can Inherit from Other Interfaces	Can be Inherited	Can be Instantiated
Normal Class	Yes	Yes	Yes	Yes
Sealed Class	Yes	Yes	No	Yes

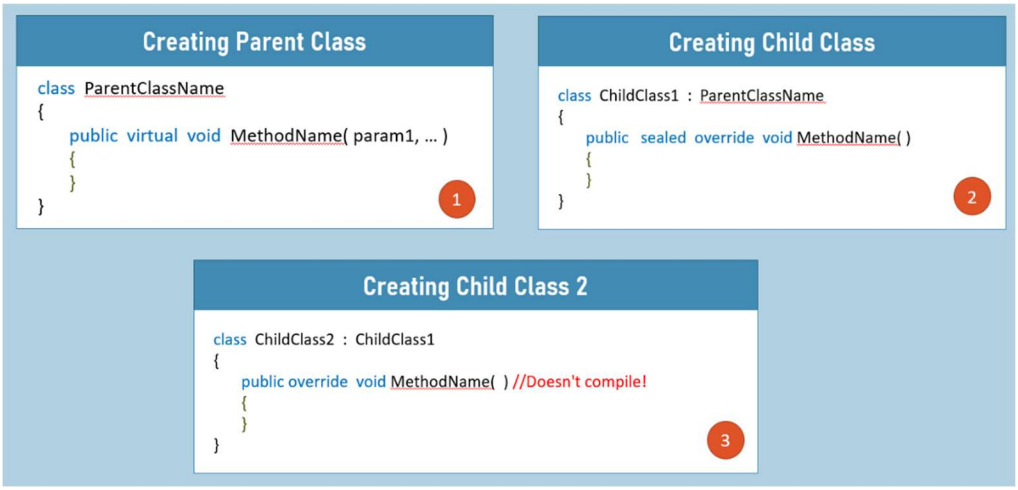
Type	1. Non-Static Fields	2. Non-Static Methods	3. Non-Static Constructors	4. Non-Static Properties	5. Non-Static Events	6. Non-Static Destructors	7. Constants
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Sealed Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Type	8. Static Fields	9. Static Methods	10. Static Constructors	11. Static Properties	12. Static Events	13. Virtual Methods	14. Abstract Methods	15. Non-Static Auto-Impl Properties	16. Non-Static Indexers
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Sealed Class	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes

Sealed Methods

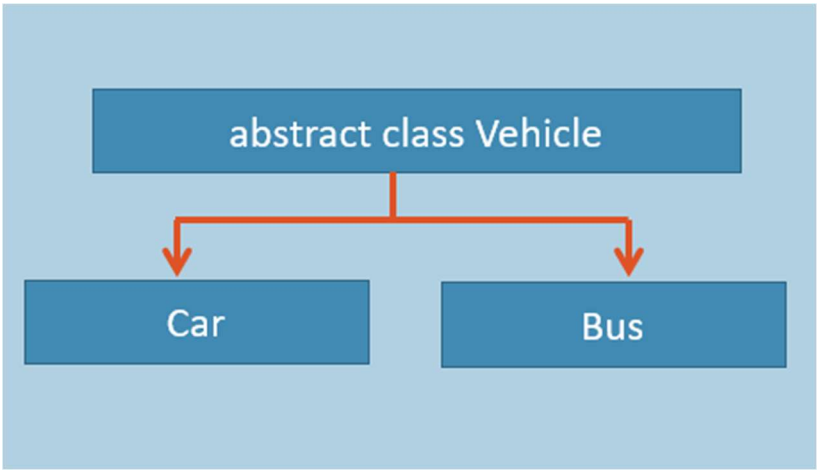
Sealed Methods must be "override methods"; which can't be overridden in the corresponding child classes.

Use sealed methods to prevent overriding that particular methods in the corresponding child classes.



Abstract Classes

Abstract class is a parent class, for which, we can't create object; but we can create child classes.



Parent Class [Abstract Class]

```
1. abstract class AbstractClassName
2. {
3.   //Abstract Class Members here
4. }
```

Child Class of Abstract Class

```
1. class ChildClassName : AbstractClassName
2. {
3.   Child Class Members here
4. }
```

The main intention of abstract class is to provide common set of fields and methods to all of its child classes of a specific group.

Abstract class can contain all types of members (fields, properties, methods, constructors etc.).

We can't create object for abstract class; but we can access its members through child class's object.

So 'creating child class of abstract class' is the only-way to utilize abstract classes.

Use Abstract class concept, for the classes, for which, you feel creating object is not meaningful.

Comparison Table: Class (vs) Abstract Class

Class Type	Can Inherit from Other Classes	Can Inherit from Other Interfaces	Can be Inherited	Can be Instantiated
Normal Class	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	No

Based on members:

Type	1. Non-Static Fields	2. Non-Static Methods	3. Non-Static Constructors	4. Non-Static Properties	5. Non-Static Events	6. Non-Static Destructors	7. Constants
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Type	8. Static Fields	9. Static Methods	10. Static Constructors	11. Static Properties	12. Static Events	13. Virtual Methods	14. Abstract Methods	15. Non-Static Auto-Impl Properties	16. Non-Static Indexers
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Abstract Methods

Abstract methods are declared in parent class, with "abstract" keyword; implemented in child classes, with "override" keyword.

When the parent class don't want to provide the definition of a method;it wants to let child classes to implement the method.

Parent Class [Abstract Class]

```
1. abstract class AbstractClassName
2. {
3.     AccessModifier abstract ReturnDataType MethodName(param1, …);
4. }
```

Child Class of Abstract Class

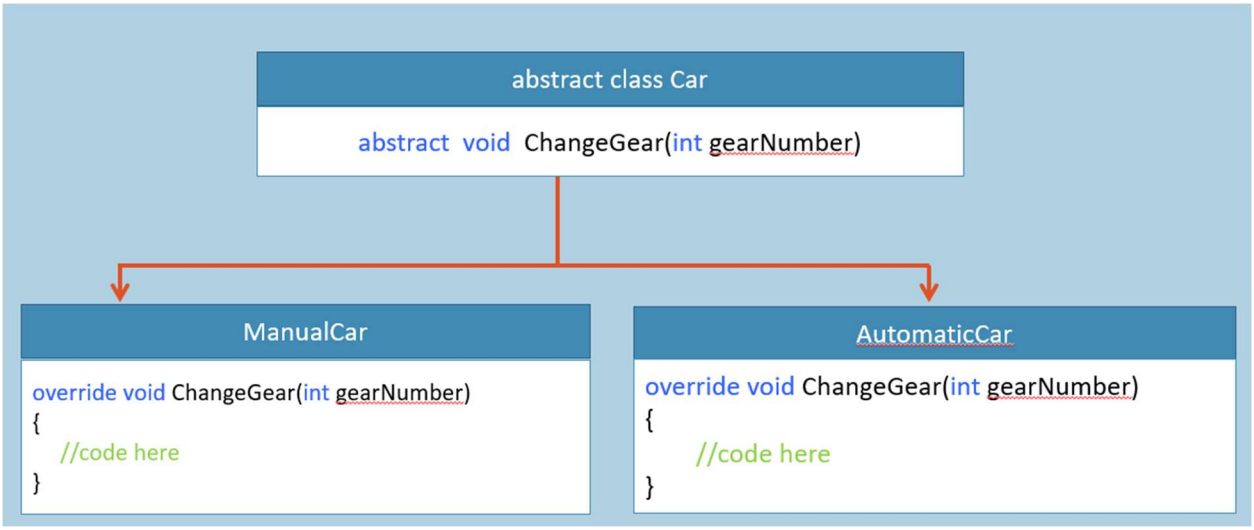
```
1. class ChildClassName : AbstractClassName
2. {
3.     AccessModifier override ReturnDataType MethodName(param1, …)
4.     {
5.     }
```

```
6. }
```

Abstract Methods contain "method declaration" only; but not "method body".

Child class must provide method body for abstract methods.

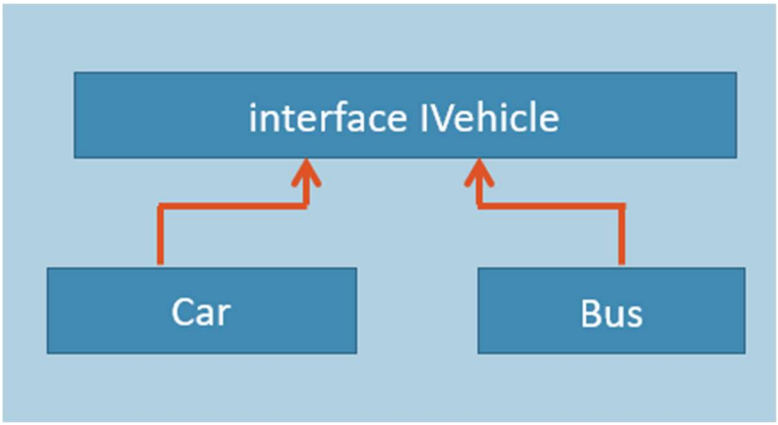
Eg:



Interfaces

Interface is a set of abstract methods, that must be implemented by the child classes.

Eg:



Interface

```
1. interface InterfaceName
2. {
```

```
3.      ReturnDataType MethodName(param1, …);
4. }
```

Child Class of Interface

```
1. class ChildClassName : InterfaceName
2. {
3.      public ReturnDataType MethodName(param1, …)
4.      {
5.      }
6. }
```

- The child class that implements the interface, MUST implement ALL METHODS of the interface.
- Interface methods are by default "public" and "abstract".
- The child class must implement all interface methods, with same signature.
- You can't create object for interface.
- You can create reference variable for the interface.
- The reference variable of interface type can only store the address of objects of any one of the corresponding child classes.
- You can implement multiple interfaces in the same child class [Multiple Inheritance].
- An interface can be child of another interface.

Comparison Table: Abstract Class (vs) Interface

Class Type	Can Inherit from Other Classes	Can Inherit from Other Interfaces	Can be Inherited	Can be Instantiated
Normal Class	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	No
Interface	No	Yes	Yes	No

Based on members:

Type	1. Non-Static Fields	2. Non-Static Methods	3. Non-Static Constructors	4. Non-Static Properties	5. Non-Static Events	6. Non-Static Destructors	7. Constants
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	Yes	No	No

Type	8. Static Fields	9. Static Methods	10. Static Constructors	11. Static Properties	12. Static Events	13. Virtual Methods	14. Abstract Methods	15. Non-Static Auto-Impl Properties	16. Non-Static Indexers
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	No	No	Yes	Yes	No

Polymorphism

Polymorphism provides the ability to the developer, to define different implements for the same method in the same class or different classes.

Compile-time polymorphism:

- Eg: Method Overloading
- Decission will be taken at compilation time.
- Also known as "Early binding" / "Static polymorphism".

Run-time polymorphism:

- Eg: Method Overriding
- Decission will be taken at run time.
- Also known as "Late binding" / "Dynamic polymorphism".

Implementation of different types of Polymorphism

```
1. public void Add(int a, int b);
2. public void Add(int a, int b, int c);
```

Run-Time Polymorphism Example - Method Overriding:

```
1. abstract class ParentClass
2. {
3.   public abstract void Add(int a, int b);
4. }
5.
6. class ChildClass1 : ParentClass
7. {
8.   public override void Add(int a, int b)
9.   {
```

```
        //Implementation for ChildClass1.Add
  }
}

class ChildClass1 : ParentClass
{
  public override void Add(int a, int b)
  {
       //Implementation for ChildClass2.Add
  }
}


ParentClass c1;
c1 = new ChildClass1();
c1.Add(10, 20); //calls ChildClass1.Add

c1 = new ChildClass2();
c1.Add(10, 20); //calls ChildClass2.Add
```

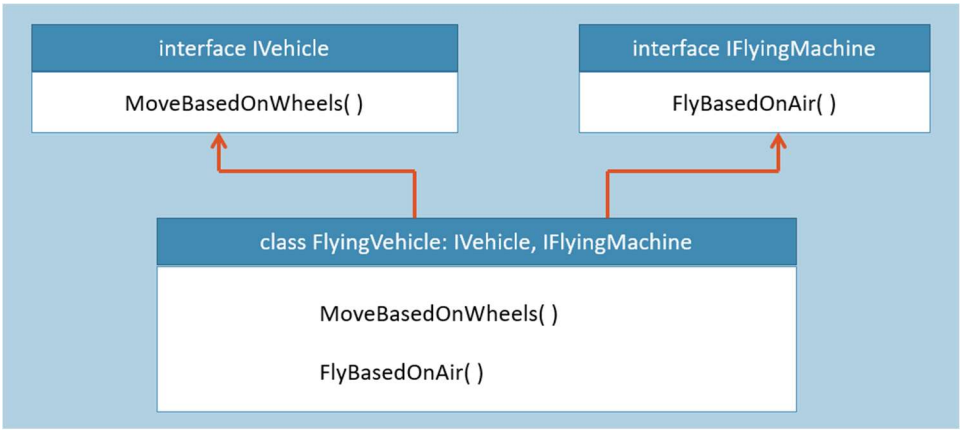
Run-Time Polymorphism Example - With Interfaces:

```
interface InterfaceName
{
  void Add(int a, int b);
}

class ChildClass1 : InterfaceName
{
    public void Add(int a, int b)
    {
         //Implementation for ChildClass1.Add
    }
}

class ChildClass2 : InterfaceName
{
    public void Add(int a, int b)
    {
         //Implementation for ChildClass2.Add
    }
}


InterfaceName c1;
c1 = new ChildClass1();
c1.Add(10, 20); //calls ChildClass1.Add

c1 = new ChildClass2();
c1.Add(10, 20); //calls ChildClass2.Add
```

Multiple Inheritance

In C#, "multiple inheritance" IS POSSIBLE with INTERFACES; that means a child class can have multiple parent interfaces.

Eg:




```
interface  Interface1
{
  void Method1(param1, param2, ...);
}

interface  Interface2
{
  void Method2(param1, param2, ...);
}

class ChildClass : Interface1, Interface2
{
  public void Method1(param1, param2, ...)
  {
    //implementation for ChildClass.Method1
  }

  public void Method1(param1, param2, ...)
  {
    //implementation for ChildClass.Method2
  }
}

Interface1 c1 = new ChildClass();
c1.Method1(...);  //calls ChildClass.Method1

Interface2 c2 = new ChildClass();
c2.Method2(...); //calls ChildClass.Method2
```

"One Child - Multiple Parent Classes / Parent Interfaces" is called as "Multiple Inheritance".

In C#.NET, "multiple inheritance" IS NOT POSSIBLE with CLASSES; that means you can't specify multiple parent classes.

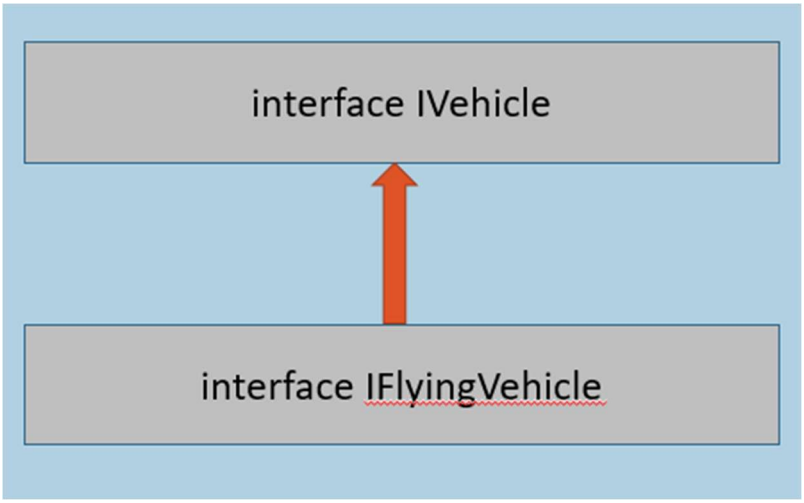
The child class MUST IMPLEMENT all methods of all the interfaces, that are inherited from.

Interface Inheritance

If an interface inherits from another interface, we call it as "Interface Inheritance".

The child class that implements the child interface must implement all the members of both parent interface and child interface too.

Eg:



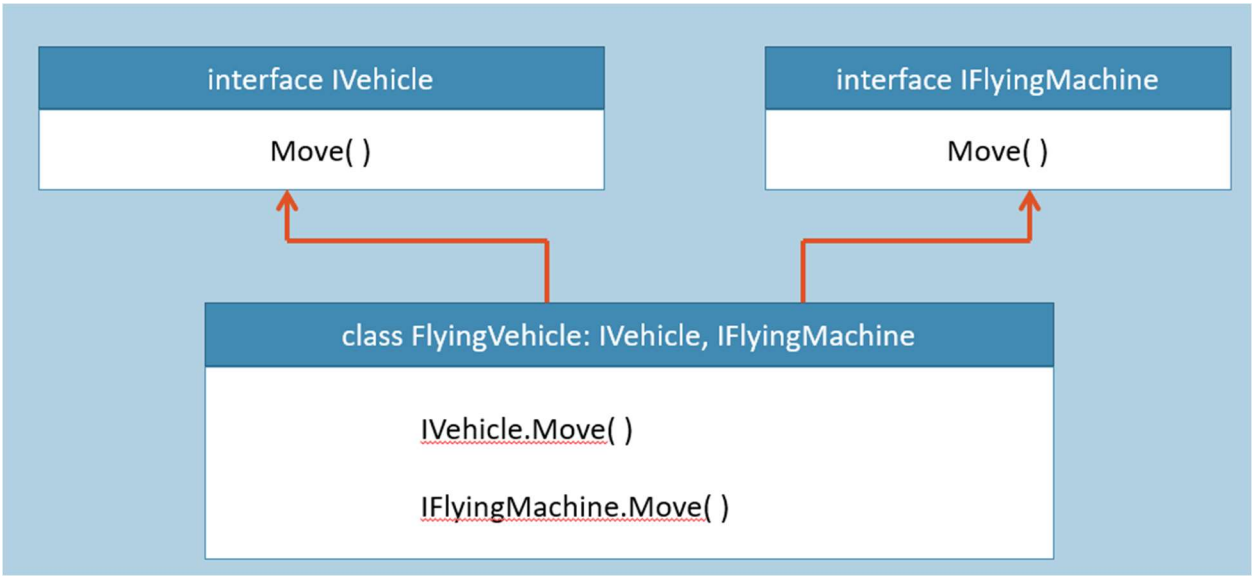

```
1. interface Interface1
2. {
3.     public void Method1(param1, param2, ...);
4. }
5.
6. interface Interface2 : Interface1
7. {
8.     public void Method2(param1, param2, ...);
9. }
10.
11.class ChildClass : Interface2
12.{
13.   public void Method1(param1, param2, ...)
14.   {
15.     //Implementation for ChildClass.Method1
```

```
16.   }
17.
18.   public void Method2(param1, param2, ...)
19.   {
20.     //Implementation for ChildClass.Method2
21.   }
22. }
23.
24. Interface1 c1 = new ChildClass();
25. c1.Method1(...); //calls ChildClass.Method1
26.
27. Interface2 c2 = new ChildClass();
28. c2.Method1(...); //calls ChildClass.Method1
29. c2.Method2(...); //calls ChildClass.Method2
```

Explicit Interface Implementation

"Explicit Interface Implementation" is used to implement an interface method privately; that means the interface method becomes as "private member" to the child class.

Eg:




```
1. interface Interface1
2. {
3.   void Method1(param1, param2, ...);
4. }
5.
6. interface Interface2
7. {
8.   void Method1(param1, param2, ...);
9. }
```

```
10.
11. class ChildClass : Interface1, Interface2
12. {
13.   void Interface1.Method1(param1, param2, ...)
14.   {
15.   }
16.
17.   void Interface2.Method1(param1, param2, ...)
18.   {
19.   }
20. }
21.
22. Interface1 c1 = new ChildClass();
23. c1.Method1(...); //calls Interface1.Method1 at ChildClass
24.
25. Interface2 c2 = new ChildClass();
26. c2.Method1(...); //calls Interface2.Method1 at ChildClass
```

If a child class inherits from two or more interfaces, and there is a duplicate method (having same name and parameters) among those interfaces; then use "Explicit Interface Implementation", to provide different implementations for different interface methods respectively.

You can use "Explicit Interface Implementation" to create private implementation of interface method; so that you can create abstraction for those methods.

Namespaces

Namespaces is a collection of classes and "other types such as interfaces, structures, delegate types, enumerations).

Eg:

In a project for an organization:

```
1. namespace FrontOffice
2. {
3. }
1. namespace Finance
2. {
3. }
1. namespace HR
2. {
3. }
1. namespace Inventory
2. {
3. }
```

Syntax:

```
namespace NamespaceName
{
  Classes
  Interfaces
  Structures
  Delegate Types
  Enumerations
}
```

Namespaces goal is to group-up classes and other types that are related to a particular project-module, into an unit.

Syntax to access a type that is present inside the namespace: `NamespaceName.TypeName`

Nested Namespaces

The namespace which is declared inside another namespace is called as "Nested namespace" or “Inner Namespace".

Use nested namespaces, in order to divide the classes of a larger namespace, into smaller groups.

Syntax to access a type in the inner namespace: `OuterNamespace.InnerNamespace.TypeName`

Syntax to create inner namespace:

```
namespace OuterNamespace
{
  Classes
  Interfaces
  Structures
  Delegate Types
  Enumerations

  namespace InnerNamespace
  {
    Classes
    Interfaces
    Structures
    Delegate Types
    Enumerations
  }
}
```

Importing Namespaces ('using' Directive)

The "using" is a directive statement (top-level statement) that should be placed at the top of the file, which specifies the namespace, from which you want to import all the classes and other types.

Syntax: `using Namespacename;`

When you import a namespace, you can directly access all of its classes and other types (but not inner namespaces).

The "using directives" are written independently for every file.

"One using directive" can import "one namespace" only.

'using' Alias Name

The "using alias" directive allows you to create "alias name" for the namespace.

Syntax:

```
using AliasName = Namespacename;
```

Use "using alias" directive, if you want to access long namespaces with shortcut name.

It is much useful to access specific namespace, when there is namespace name ambiguity (two classes with same name in two different namespaces and both namespaces are imported in the same file).

'using' static

The "using static" directive allows you import a static class directly from a namespace; so that you can directly access any of its methods anywhere in the current file.

Syntax:

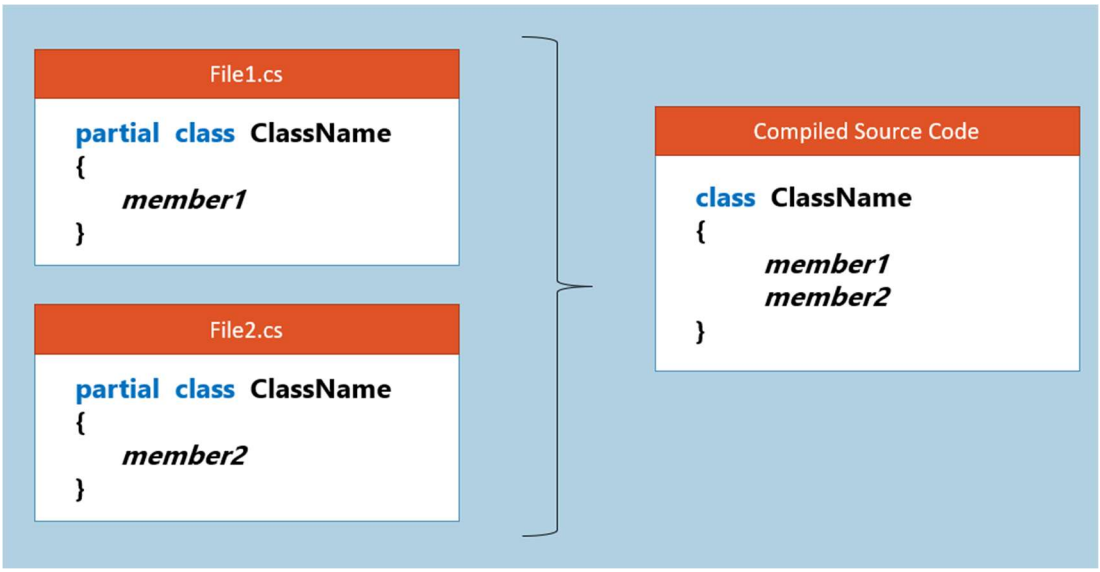
```
using static Namespacename.StaticClassName;
```

Use the "using static" directive to access methods of static class easily, without repeating the class name each time.

Partial Classes

Partial Class is a class that splits into multiple files.

Each file is treated as a "part of the class".



At compilation time, all partial classes that have same name, become as a "single class".

All the partial classes (that want to be a part of a class) should have same name and should be in the same namespace and same assembly & should have same access-modifier (such as 'internal' or 'public').

Duplicate members are not allowed in partial classes.

Any attributes / modifiers (such as abstract, sealed) applied on one partial class, will be applied to all partial classes that have same name.

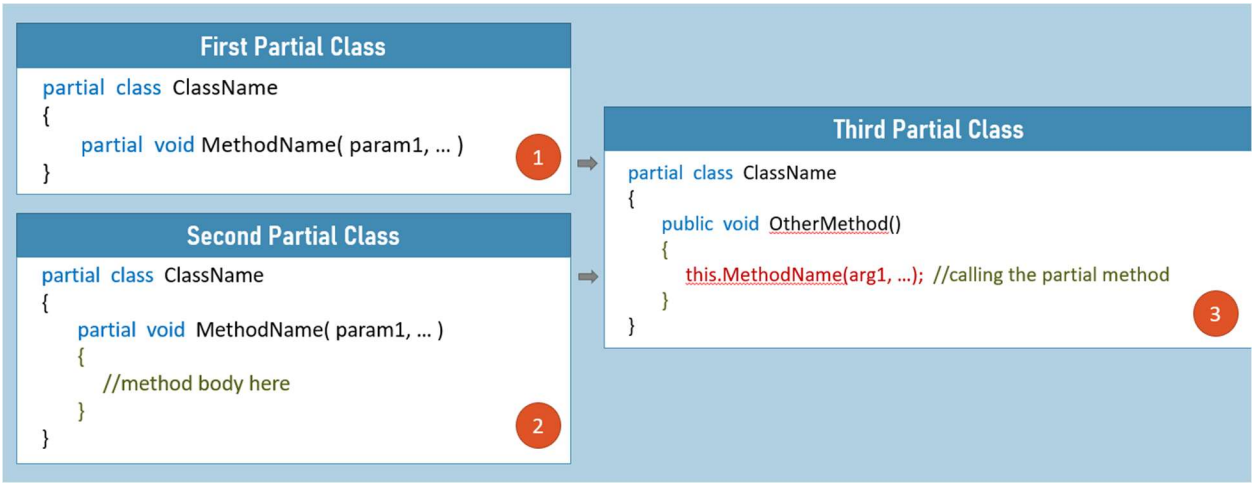
The 'partial' keyword can be used only at before the keywords 'class', 'struct', 'interface' and 'void'.

Each partial class can be developed individually, by different developers / teams.

In WinForms / WebForms, the 'Designer-generated code' will be kept in one partial class; the 'code written by developer' will be kept in another partial class with same name; so both become as a single class at compilation time.

Partial Methods

Partial Methods are "declared in one partial class" (just like abstract method), and "implemented in another partial class", that have same name.



Assume, there are two developers; the first developer develops the first partial class; second developer develops the second partial class.

The partial method lets the first developer to declare a partial method in one partial class; and the second developer implements the partial method in the other partial class.

Partial Methods can only be created in partial class or partial structs.

Partial Methods are implicitly private. It can't have any other access modifier.

Partial Methods can have only "void" return type.

Implementation of partial methods is optional. If there is no implementation of partial methods in any parts of the partial class, the method calls are removed by the compiler, at compilation time.

If you are building large class libraries and decide extension of methods to other developers, partial methods can be used.

Static Classes

Static class is a class that can contain only "static members".

If you don't want even single 'instance member' (non-static member), then use 'static class'.

Advantage: We can avoid accidental creation of object for the class, by making it as "static class".

Syntax to create a static class:

```
1. static class ClassName
2. {
3.   static fields
4.   static methods
```

```
5.    static constructors
6.    static properties
7.    static events
8. }
```

Static Classes [vs] Normal Classes

Static Class:

```
1. static class ClassName
2. {
3.    static fields
4.    static methods
5.    static constructors
6.    static properties
7.    static events
8. }
```

Static class is to create ONLY static members (static fields, static methods etc.)

Static Class:

```
1. class ClassName
2. {
3.    static fields
4.    static methods
5.    static constructors
6.    static properties
7.    static events
8.
9.    instance fields
10.   instance methods
11.   instance constructors
12.   instance properties
13.   instance events
14.   destructors
15.}
```

Normal (non-static class) is to create both instance & static members [or] only instance members.

Comparison Table: Class [vs] Static Class

Class Type	Can Inherit from Other Classes	Can Inherit from Other Interfaces	Can be Inherited	Can be Instantiated
Normal Class	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	No
Interface	No	Yes	Yes	No
Sealed Class	Yes	Yes	No	Yes
Static Class	No	No	No	No

Type	1. Non-Static Fields	2. Non-Static Methods	3. Non-Static Constructors	4. Non-Static Properties	5. Non-Static Events	6. Non-Static Destructors	7. Constants
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	Yes	No	No
Sealed Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Static Class	No	No	No	No	No	No	Yes

Type	8. Static Fields	9. Static Methods	10. Static Constructors	11. Static Properties	12. Static Events	13. Virtual Methods	14. Abstract Methods	15. Non-Static Auto-Impl Properties	16. Non-Static Indexers
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	No	No	Yes	Yes	No
Sealed Class	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Static Class	Yes	Yes	Yes	Yes	Yes	No	No	No	No

Enumerations

Enumeration is a collection of constants.

Enumeration is used to specify the list of options allowed to be stored in a field / variable.

Use enumeration if you don't want to allow other developers to assign other value into a field / variable, other than the list of values specified in the enumeration

Syntax to access member of enum: `EnumerationName.ConstantName`

Syntax to create Enumeration:

```
1. enum EnumerationName
2. {
3.    Constant1, Constant2, …
4. }
```

By default, each constant will be assigned to a number, starts from zero; however you can change the number (integer only).

```
1. enum EnumerationName
2. {
3.   Constant1 = value, Constant2 = value, …
4. }
```

The default data type of enum member is "int". However, you can change its data type as follows.

```
1. enum EnumerationName : datatype
2. {
3.    Constant1 = value, Constant2 = value, …
4. }
```

Value-Types vs Reference-Types

Value Types (Structures, Enumerations)

• Mainly meant for storing simple values.

• Instances (examples) are called as "structure instances" or "enumeration instances".

• Instances are stored in "Stack". Every time when a method is called, a new stack will be created.

Reference Types (string, Classes, Interfaces, Delegates)

• Mainly meant for storing complex / large amount of values.•Instances (examples) are called as "Objects" (Class Instances / Interface Instances / Delegate Instances).

• Instances (examples) are called as "Objects" (Class Instances / Interface Instances / Delegate Instances).

• Instances (objects) are stored in "heap". Heap is only one for entire application.

Structures

Structure is a "type", similar to "class", which can contain fields, methods, parameterized constructors, properties and events.

Structure - Example

```
struct Student
{
  public int studentId;
  public string studentName;

  public string GetStudentName( )
  {
    return studentName;
  }
}
```

Structure - Syntax

```
struct StructureName
{
  fields
  methods
  parameterized constructors
  properties
  events
}
```

- The instance of structure is called as "structure instance" or "structure variable"; but not called as 'object'. We can't create object for structure. Objects can be created only based on 'class'.
- Structure instances are stored in 'stack'.
- Structure doesn't support 'user-defined parameter-less constructor and also destructor.
- Structure can't inherit from other classes or structures.
- Structure can implement one or more interfaces.
- Structure doesn't support virtual and abstract methods.
- Structures are mainly meant for storing small amount of data (one or very few values).
- Structures are faster than classes, as its instances are stored in 'stack'.

Class (vs) Structure

Structures

1. Structures "value-types".
2. Structure instances (includes fields) are stored in stack. Structures doesn't require Heap.Structure instances (includes fields) are stored in stack. Structures doesn't require Heap.
3. Suitable to store small data (only one or two values).
4. Memory allocation and de-allocation is faster, in case of one or two values.
5. Structures doesn't support Parameter-less Constructor.
6. Structures doesn't support inheritance (can't be parent or child).
7. The "new" keyword just initializes all fields of the "structure instance".
8. Structures doesn't support abstract methods and virtual methods.

9. Structures doesn't support destructors.
10. Structures are internally derived from "System.ValueType". System.Object -> System.ValueType -> Structures
11. Structures doesn't support to initialize "non-static fields“, in declaration.
12. Structures doesn't support "protected" and "protected internal" access modifiers.
13. Structure instances doesn't support to assign "null".

Classes

1. Classes are "reference-types".
2. Class instances (objects) are stored in Heap; Class reference variables are stored in stack.
3. Suitable to store large data (any no. of values)
4. Memory allocation and de-allocation is a bit slower.
5. Classes support Parameter-less Constructor.
6. Classes support Inheritance.
7. The "new" keyword creates a new object.
8. Classes support abstract methods and virtual methods.
9. Classes support destructors.Classes are internally and directly derived from "System.Object". System.Object -> Classes
10. Classes supports to initialize "non-static fields“, in declaration.
11. Classes support "protected" and "protected internal" access modifiers.
12. Class's reference variables support to assign "null".

Comparison Table - Class (vs) Structure

Class Type	Can Inherit from Other Classes	Can Inherit from Other Interfaces	Can be Inherited	Can be Instantiated
Normal Class	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	No
Interface	No	Yes	Yes	No
Sealed Class	Yes	Yes	No	Yes
Static Class	No	No	No	No
Structure	No	Yes	No	Yes

Type	1. Non-Static Fields	2. Non-Static Methods	3. Non-Static Constructors	4. Non-Static Properties	5. Non-Static Events	6. Non-Static Destructors	7. Constants
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	Yes	No	No
Sealed Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Static Class	No	No	No	No	No	No	Yes
Structure	Yes	Yes	Yes	Yes	Yes	No	Yes

Type	8. Static Fields	9. Static Methods	10. Static Constructors	11. Static Properties	12. Static Events	13. Virtual Methods	14. Abstract Methods	15. Non-Static Auto-Impl Properties	16. Non-Static Indexers
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	No	No	Yes	Yes	No
Sealed Class	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Static Class	Yes	Yes	Yes	Yes	Yes	No	No	No	No
Structure	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes

Constructors in Structures

C# provides a parameter-less constructor for every structure by default, which initializes all fields.

You can also create one or more user-defined parameterized constructors in structure.

Each parameterized constructor must initialize all fields; otherwise it will be compile-time error.

The "new" keyword used with structure, doesn't create any object / allocate any memory in heap; It is a just a syntax to call constructor of structure.

```
1. public StructureName( datatype parameter)
2. {
3.   field = parameter;
4. }
```

Read-only Structures

Use readonly structures in case of all of these below:

- All fields are readonly.

- All properties have only 'get' accessors (readonly properties).
- There is a parameterized constructor that initializes all the fields.
- You don't want to allow to change any field or property of the structure.
- Methods can read fields; but can't modify.

Readonly Structure - Example

```
1. readonly struct Student
2. {
3.   public readonly int studentId;
4.   public string studentName { get; }
5.   public Student( )
6.   {
7.     studentId = 1;
8.     studentName = "Scott";
9.   }
10.}
```

'Readonly structures' is a new feature in C# 8.0.

This feature improves the performance of structures.

Primitive Types as Structures

All primitive types are structures.

For example, "sbyte" is a primitive type, which is equivalent to "System.SByte" (can also be written as 'SByte') structure.

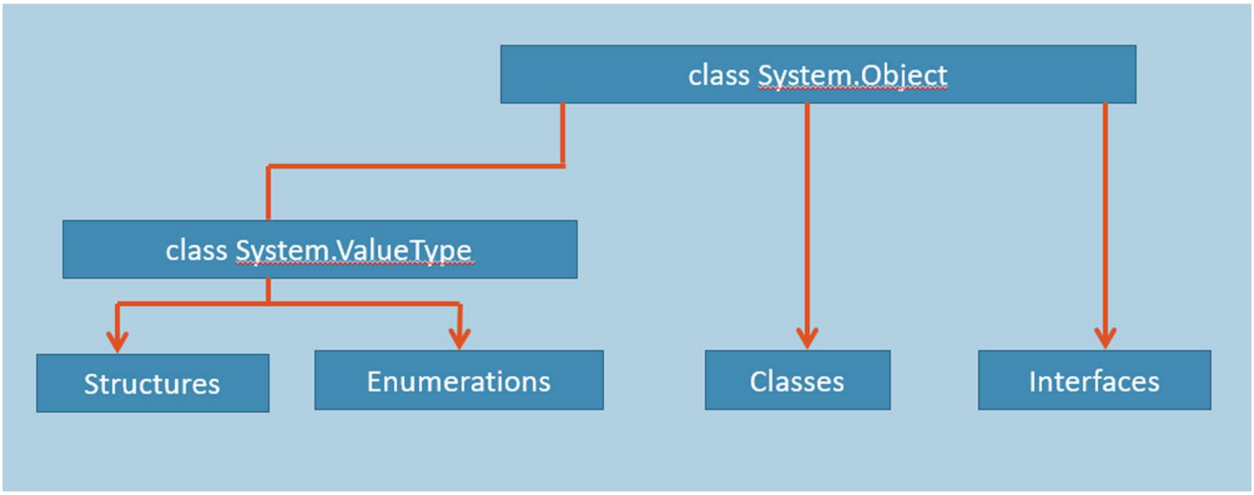
In C#, it is recommended to always use primitive types, instead of structure names.

Data Type	Is it Structure / Class?	Name of Structure / Class	Full Path (with namespace)
sbyte	Structure	SByte	System.SByte
byte	Structure	Byte	System.Byte
short	Structure	Int16	System.Int16
ushort	Structure	UInt16	System.UInt16
int	Structure	Int32	System.Int32
uint	Structure	UInt32	System.UInt32
long	Structure	Int64	System.Int64
ulong	Structure	UInt64	System.UInt64
float	Structure	Single	System.Single
double	Structure	Double	System.Double
decimal	Structure	Decimal	System.Decimal
char	Structure	Char	System.Char
bool	Structure	Boolean	System.Boolean
string	Class	String	System.String

System.Object class

The "System.Object" is a pre-defined class, which is the "Ultimate super class (base class)" in .net.

All the classes and other types are inherited from System.Object directly / indirectly.



All C# classes, structures, interfaces, enumerations are children of System.Object class.

Every method defined in the Object class is available in all objects in the system as all classes in the .NET Framework are derived from Object class.

Derived classes can override Equals, GetHashCode and ToString methods of Object class.

System.Object class is meant for achieving "type safety" in C#.

Methods of System.Object class

System.Object Class [Pre-defined]

```
1. namespace System
2. {
3.   class Object
4.   {
5.     virtual bool Equals( object value );
6.     virtual int GetHashCode( );
7.     Type GetType( );
8.     virtual string ToString( );
9.   }
10.}
```

› **bool Equals(object value)**

Compares the current object with the given argument object; returns true, if both are same objects; returns false, if both are different objects.

› **int GetHashCode(object value)**

Returns the a number that represents the object. It is not guarantee that the hash code is unique, by default.

› **Type GetType( )**

Returns the name of the class (including namespace path), based on which, the object is created.

› **string ToString( )**

By default, it returns the name of the class (including namespace path), based on which, the object is created.

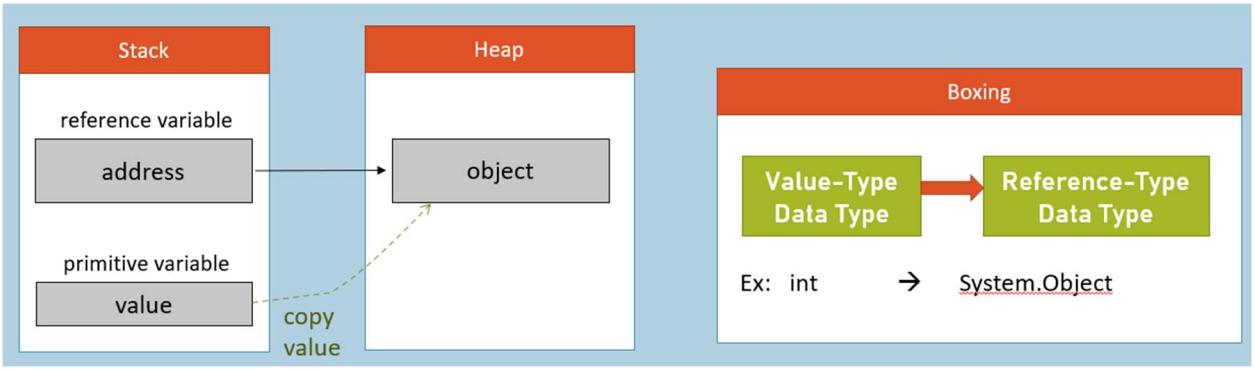
It is virtual method, which can be overridden in the child class.

Boxing & Unboxing

Boxing

It is a process of converting a value from "Value-Type Data Type" to "Reference-Type Data Type", if they are compatible data types.

This can be done automatically [no need of any syntax].



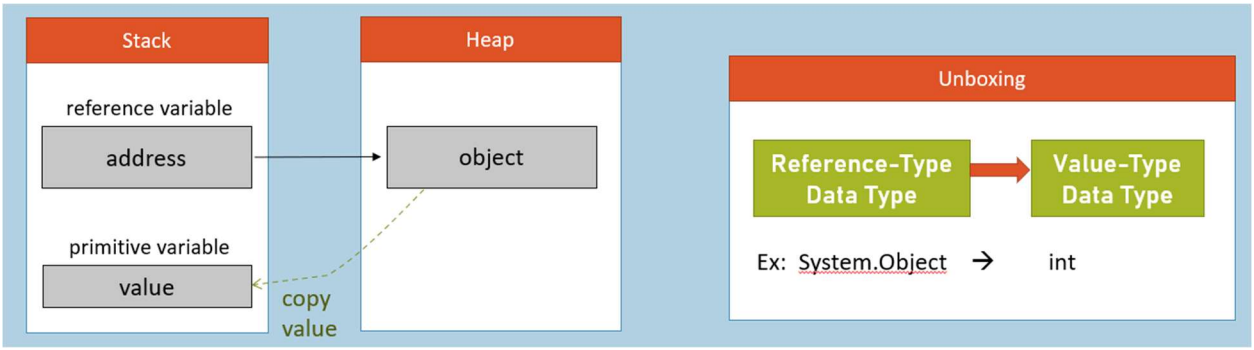
Unboxing

It is a process of converting a value from "Reference-Type Data Type" to "Value-Type Data Type", if they are compatible data types.

This should be done explicitly (by using explicit casting).

Syntax: `(DestinationDataType)SourceValue`

Example: `(short)100`



Generics

Generic class is a class, which contains one or more "type parameters".

You must pass any data type (standard data type / structure / class), while creating object for the generic class.

Generic Class - Example

```
1. class ClassName<T>
2. {
3.    public T FieldName;
4. }
```

Object of Generic Class - Example

```
ClassName<int> referenceVariable = new ClassName<int> ( );
```

- The same field may belong to different data types, w.r.t. different objects of the same class.
- You will decide the data type of the field, while creating the object, rather than while creating field in the class.

- It helps you in code reuse, performance and type-safety.
- You can create your own generic-classes, generic-methods, generic-interfaces and generic-delegates.
- You can create generic collection classes. The .NET framework class library contains many new generic collection classes in System.Collections.Generic namespace.
- The generic type parameter (T) acts as "temporary data type", which represents the actual data type, provided by the user, while creating object.
- You can have multiple "generic type parameters" in the same class (for use for different fields.
- Generics are introduced in C# 2.0.

Generic Constraints

Generic Constraints are used to specify the types allowed to be accepted in the "generic type parameter".

- where T : class
- where T : struct
- where T : ClassName
- where T : InterfaceName
- where T : new( )

Generic Constraints - Example

```
1. class ClassName<T> where T : class
2. {
3.    public T FieldName;
4. }
```

Object of Generic Class - Example

```
ClassName<int> referenceVariable = new ClassName<int> ( );
//error
```

Understanding Generic Constraints

Advantage: You can restrict what type of data types (class names) allowed to be passed while creating object.

In C#, constraints are used to restrict a generics to accept only particular type or its derived types.

By using 'where' keyword, we can apply constraints on generics.

You can apply multiple constraints on generic classes or methods based on your requirements.

Eg: `where T : class where T2 : class`

Generic Methods

Generic Method is a method that has one or more generic parameter(s).

You can restrict what type of data types to be allowed to be passed to the parameter while calling the method.

Generic Method - Example

```
1. public void MethodName<T>
2. {
3. }
```

Calling Generic Method - Example

```
MethodName<datatype>( valueHere );
```

Nullable Types

Value Types (structures, enumerations)

- Value Types are by default non-nullable types.
- Non-nullable types doesn't support 'null' values to be assigned to its variables.

Reference Types (classes, interfaces)

- Reference Types are by default nullable types.
- Nullable types support 'null' values assigned to its variables.
- They don't require the following syntax.

Cnverting Value-Types to Nullable-Types

```
1. Nullable<int> x = null;
2. [or]
3. int? x = null;
```

Accessing value of a nullable type:

```
variable.Value
```

```
variable.HasValue
```

What is null?

Represents 'blank' value.

Eg: In Employee class, the 'int CreditCardNumber' can be 'null'.

Null coalescing operator

The 'null coalescing operator' checks whether the value is null or not.

It returns the left-hand-side operand if the value is not null.

It returns the right-hand-side operand if the value is null.

Advantage: Simplifying the syntax of 'if statement' to check if the value is null.

Syntax:

```
variableName ?? valueIfNull
```

Null Propagation Operator

The "Null Propagation Operator (?.) and (? []) checks the value of left-hand operand whether it is null or not.

It returns the right-hand-side operand (property or method), if the value is not null.

It returns null, if the value is null.

It accesses the property or method, only if the reference variable is "not null"; just returns "null", if the reference variable is "null".

Null Propagation Operator (?.) - Syntax

```
referenceVariable?.fieldName;
```

-- is same as --

```
(referenceVariable == null)? null : referenceVariable.fieldName;
```

Advantage: We can invoke desired member (property or method) after checking if null.

Extension Methods

Extension method is a method injected (added) into an existing class (or struct or interface), without modifying the source code of that class (or struct or interface).

Existing Class

```
1. class ClassName
2. {
3. }
```

Static Class for Extension Method

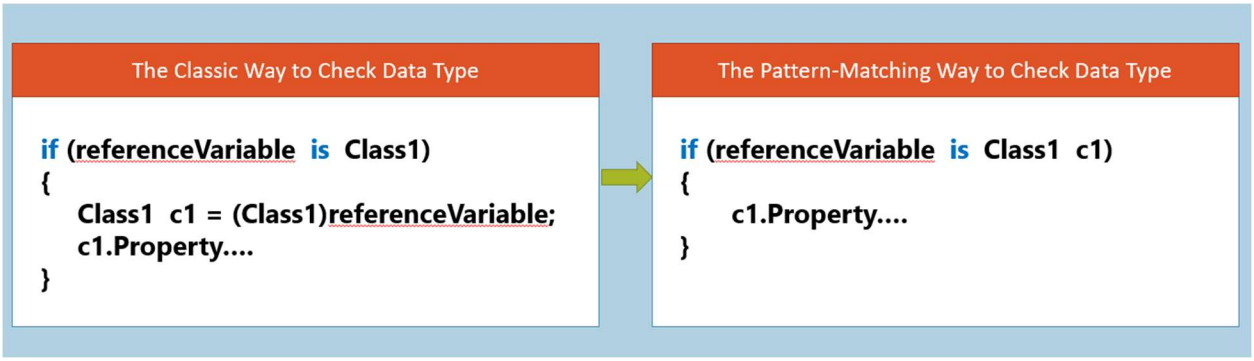
```
1. static class ClassName
2. {
3.   public static ReturnType MethodName(this ClassName ParameterName, …)
4.   {
5.     method body here
6.   }
7. }
```

- The developer of ClassLibrary, creates a class with a set of methods. The consumer of ClassLibrary, can add additional methods to the same class, without modifying the source code of the ClassLibrary.
- You can add additional methods to pre-defined classes / structures such as String, Int32, Console etc.
- You must create a static class with a static method; that it will be added as a non-static method to the specified class.
- This feature is introduced in C# 3.0.
- The first parameter of extension must be having "this" keyword; followed by the class name / structure name, to which you want to add the extension method. Eg: this ClassName parameter
- The parameter (with 'this' keyword) represents the current object, just like "this" keyword in the instance methods.
- Extension method can have any no. of additional parameters, where the "this" keyword parameter is must.
- Extension method does not support method overriding. That means, extension method's signature can't be same as any existing method.

- You can also add extension methods to sealed class.
- 'Extension Methods' concept can't be used to create fields, properties, or events.
- The static class of extension method can't be inner class.
- The namespace in which the static class of extension method is created, must be imported in order to call the extension method as non-static method.

Pattern Matching

It allows you to declare a variable, while checking the data type (class) of a reference variable, and automatically type-casts the reference variable into the specified data type (class).



Advantage: Simplified syntax to perform multiple checks of data types and type-casts.

Implicitly-Typed Variables

The variables that are declared with 'var' keyword are called as 'implicitly-typed variables' (a.k.a type-inference).

Implicitly-typed variables are declared without specifying the 'type' explicitly; so that the C# compiler automatically identifies the appropriate data type at compilation-time, based on the value assigned at the time of declaration.

Syntax: `var variableName = value;`

While declaration, the 'type' of implicitly-typed variables is fixed. It is not possible to change the type of that variable or assign "other type of values" into the implicitly typed variables, after declaration.

Implicitly Typed Variables can only be "local variables"; can't be used for method parameters, return type or fields.

Implicitly Typed Variables must be initialized along with declaration.

It is not possible to declare multiple implicitly typed variables in the same statement. Eg: var x = 10, y = 20; //error

It is not possible to assign "null" into implicitly typed variables (while declaration). Eg: var x = null; //error

Dynamically-Typed Variables

Dynamically Typed Variables are the variables that are declared with 'dynamic' keyword.

Declared without specifying the type explicitly.

There is no fixed type for the variable.

You can assign any type of value to these variables.

C# compiler skips "type-checking" at compilation time; instead, it resolves the data types of its values, at run time.

Syntax: `dynamic variableName = value;`

The "dynamic" type variables are converted as "object" type in most cases. **Eg:** `dynamic dynamicVariable = 100; -> object dynamicaVariable = 100;`

The Dynamically Typed Variable can change its data type, any no. of times, at run time.

Methods and other members of 'dynamically typed variables' will not be checked by the compiler at compilation time; will be checked by CLR at run time.

If the method or other member not available, it would not cause compile-time error; it raises run-time error, when the execution flow encountered that particular statement. **Eg:** `dynamicVariable.NonExistingMethod( ); //run-time error (exception)`

The Dynamically Typed Variables need not be initialized, while declaration.

The Dynamically Typed Variable doesn't have "Intellisense" in Visual Studio.

The "dynamic" keyword is allowed for local variables, method parameters, fields, properties, return types etc.

Inner Classes

"Inner Class" (a.k.a. Nested Class) is a class, which is created in another class (outer-class or containing-class).

Syntax:

```
1. class ClassName
2. {
3.   class InnerClassName
4.   {
5.     Members here
6.   }
7. }
```

Advantage: We can create all inter-related classes of a class, "inner classes".

Syntax to access inner classes: `OuterClassName.InnerClassName`

By default, inner class is "private"; so it is accessible within the same outer class. To make it available to outside of the outer class, you can use other access modifiers such as "protected", "private protected", "internal", "protected internal" or "public".

A nested class can be declared as a private (default), public, protected, internal, protected internal, or private protected.

Outer class can't access the members of inner class directly, without object.

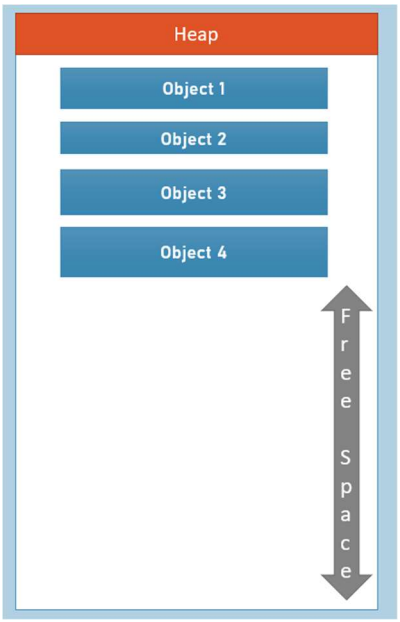
Inner class can't access the members of outer class directly, without object.

You are allowed to create objects of inner class in outer class; and vice versa; but you can't do both; if you create objects vice-versa, it causes StackOverflowException.

You can create a child class for the inner class, outside the outer class.

Garbage Collection

Garbage Collection is a process of deleting objects from memory, to free-up memory; so the same memory can be re-used.



How Garbage Collection Works?

- CLR automatically allocates memory for all objects created any where in the application, whenever it encounters "new ClassName( )" statement. This process is called as "Memory Management", which is done by "Memory Manager" component of CLR.
- All objects are stored in "Heap" (a.k.a. virtual memory).
- Heap is only-one for the entire application life time.
- The default heap size 64 MB (approx.), and extendable.
- When CLR can't find space for storing new objects, it performs a process called "Garbage Collection" automatically, which includes "identification of un-referenced objects and deleting them from heap; so that making room for new objects". This process is done by "Garbage Collector (GC)" component of CLR.

How GC decides if objects are alive?

› GC checks belongs information from the MSIL code:

› It collects references of an object.

› The objects that are referenced by at least one reference variable (or reference field) are "alive objects"; others are "dead objects" or "un-used objects".

When GC gets triggered?

There are NO specific timings for GC to get triggered.

GC automatically gets trigged in the following conditions:

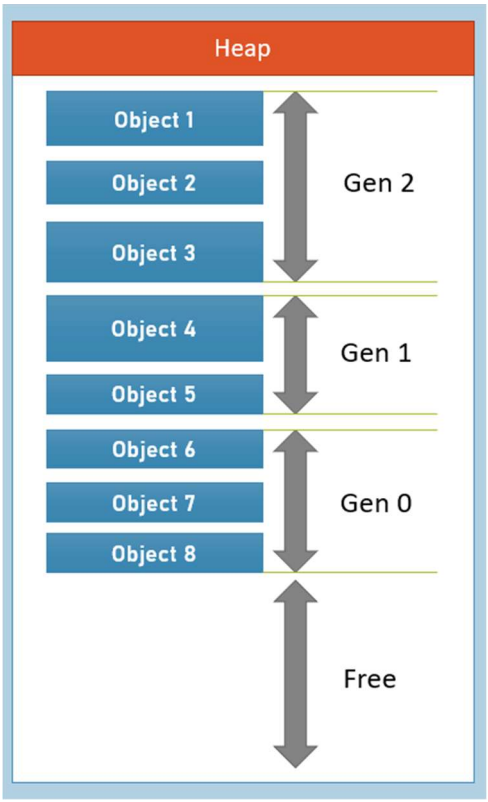
- When the "heap" is full or free space is too low.

- When we call GC.Collect() explicitly.

Generations in GC

Heap contains three segments (called generations):

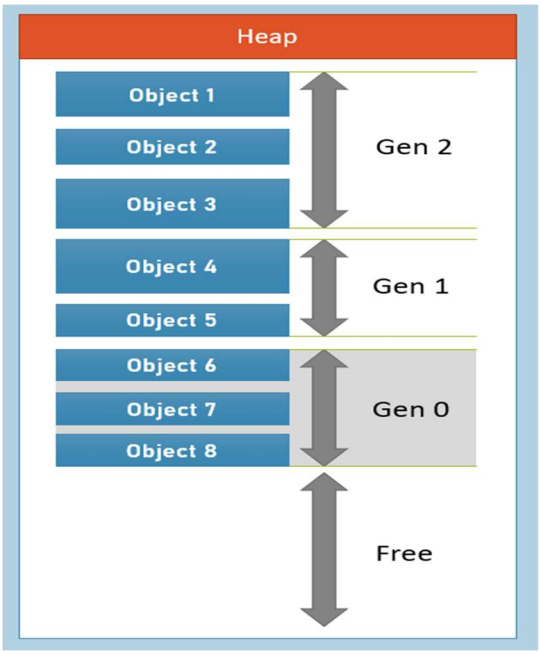
- Generation 2 [Long-Lived Generation]
- Generation 1 [Survival Generation]
- Generation 0 [Short-Lived Generation]



Generation 0

The "Generation 0" is the youngest generation and contains newly created short-lived objects and collected at first priority. The objects survive longer, are promoted to "Generation 1".

Eg: The newly created objects.

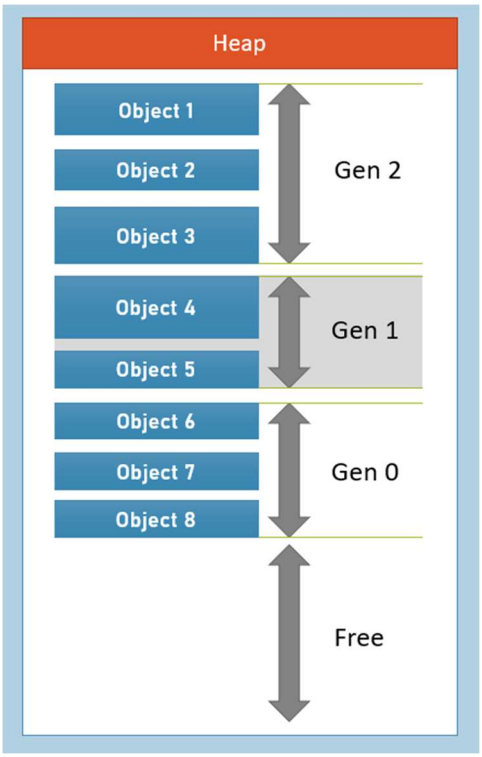


Generation 1

The "Generation 1" is buffer between "Generation 0" and "Generation 2".

The "Generation 1" mainly contains frequently-used and longer-lived objects.

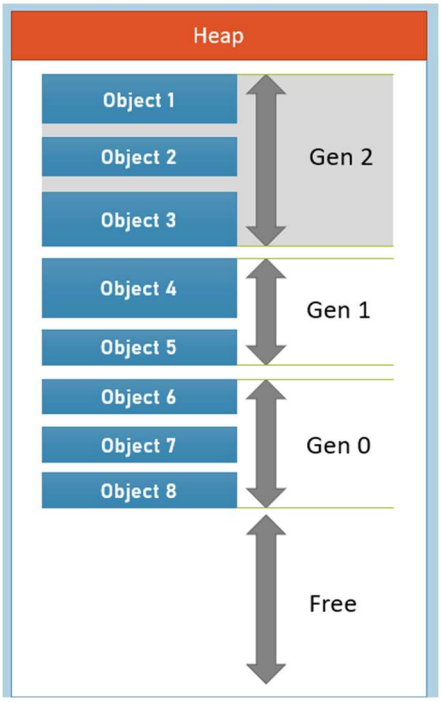
Eg: The objects created in the previously-executed methods, but still accessible.



Generation 2

The "Generation 2" contains the longest-lived objects that were created long-back and still is being used, by different statements in the program.

Eg: The objects that referenced with static fields.



Managed (vs) Unmanaged Resources

Managed Resources

The objects that are created by CLR are called as "Managed Resources".

These will participate in "Garbage Collection" process, which is a part of .NET Framework.

Unmanaged Resources

The objects that are not created by CLR and not managed by CLR are called as "Un-managed resources".

Eg: File streams, database connections

Clearing Unmanaged Resources

Destructor

Clears unmanaged resources just before deleting the object; i.e. generally at the end of application execution.

Dispose

Clears unmanaged resources after the specific task (work) is completed; so no need to wait till end of application execution.

Destructor

Destructor is a special method of the class, which is used to close un-managed resources (such as database connections and file connections), that are opened during the class execution.

Syntax:

```
1. ~ClassName( )
2. {
3.   //body here…
4. }
```

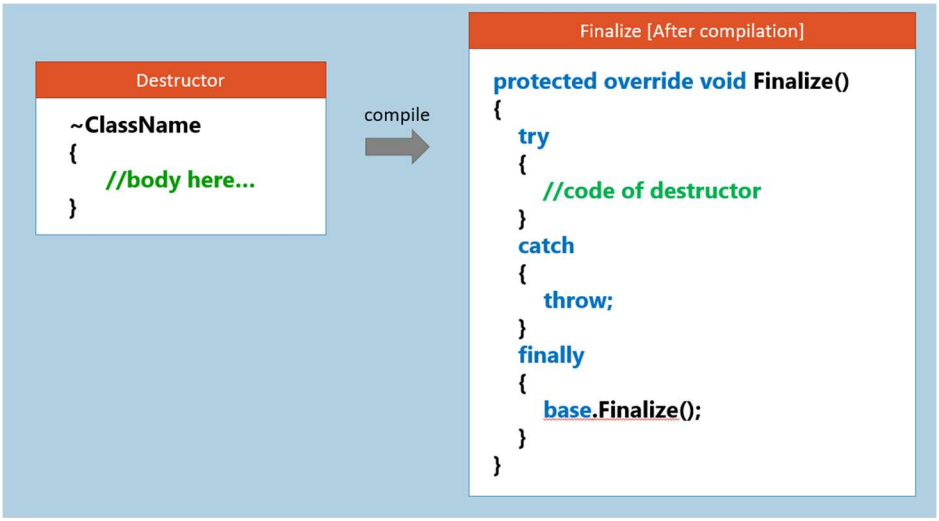
We close database connections and file connections; so no memory wastage or leakage.

Destructor doesn't de-allocate any memory; it just will be called by CLR (.net runtime engine) automatically, just before a moment of deleting the object of the class.

Rules of Destructor

- Destructor's name should be same as class name, started with ~ (tilde) character.
- A Destructor is unique to its class i.e. there cannot be more than one destructor in a class.
- Destructor can't have parameters or return value.
- Destructor is "public" by default, we can't change its access modifier.
- Destructor doesn't support any other modifiers such as "virtual", "abstract", "override" etc.
- Destructors can be defined only in classes; but not in structs, interfaces etc.
- Destructors can't be overloaded or inherited.
- Destructors are usually called at the end of program execution.

Destructor (vs) Finalize method



Internally, destructor is compiled as the "Finalize" method.

The "destructor" is a term belongs to C# language; the "Finalize" method belongs to .net framework generally; and both are same (interchangeable).

The compiled Finalize method calls the Finalize method of corresponding base class.

IDisposable and Dispose

The "IDisposable" interface of "System" namespace, has a method called "Dispose", which is used to close un-managed resources that are created during the life-time of the object.

Implementing System.IDisposable interface

```
1. class ClassName : System.IDisposable
2. {
3.   public void Dispose( )
4.   {
5.     //Close un-managed resources here
6.   }
7. }
```

Using block (creating object with IDisposable)

```
1. using (ClassName referenceVariable = new ClassName( ) )
2. {
3.   //your code here
4. }
```

The un-managed resources include file streams and database connections.

At the end of "using" statement, automatically "Dispose" method will be called.

Dispose is better than Destructor, because we need wait till 'end of application execution' to clear unmanaged resources; we clear them immediately after usage.

Using Declaration

You can prefix "using" keyword before the local variable declaration, in order to call "Dispose" method when that variable goes out of scope.

New feature introduced in C# 8.0

Creating object

```
1. public void Method( )
2. {
3.   using ClassName referenceVariable = new ClassName( );
4.
5.   //do work here
6.
7. } //Dispose will be called automatically here
```

Delegates

"Delegate type" is a "type" that represents methods that have specific parameters and return type.

The "delegate" (a.k.a. delegate object), is an object that stores reference (address) of a specific method of a specific class, with compatible parameters and return type, which is already defined in the delegate type.

1. Creating Delegate Type

```
public delegate ReturnType  DelegateTypeName(param1, param2);
```

2. Creating Delegate Object

```
DelegateTypeName  ReferenceVariable = new
DelegateTypeName(MethodName);
```

3. Invoke Method using Delegate Object

```
ReferenceVariable.Invoke(arg1, arg2, ...);
```

Rules of Delegates

- You can invoke the methods using 'delegate objects' (or) 'delegates'.

- Delegates are used to pass methods as arguments to other methods.
- The method signature (parameters and return type) must match between the "method" and "delegate".
- Delegates can be used as "parameter type" or "return type" of a method.
- You can store references of non-static method or static method in the delegate object.
- The methods, which reference is stored in the "single-cast delegate object", can have return value.
- The methods, which reference is stored in the "multi-cast delegate object", can't have return value; in case, if they have return value, the return value of lastly-executed method only can be received; others will be ignored.
- All delegate types are derived from "System.Delegate" class.

Types of Delegates

Single-Cast Delegates

- Contains reference of only one method.
- When called, it directly invokes the referenced method.

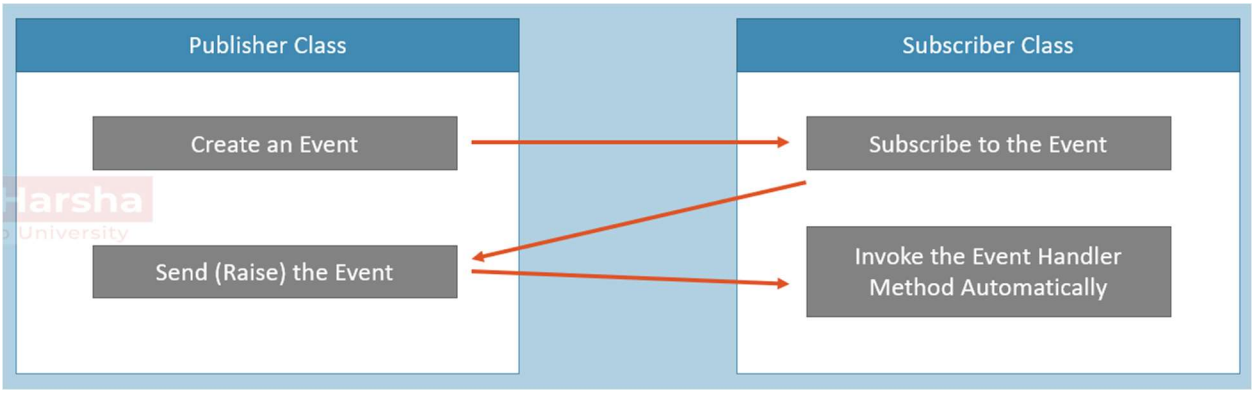
Multi-Cast Delegates

- Contains references of multiple methods.
- When called, it invokes all the referenced methods, one-by-one in a sequence.
- All methods' parameters and return type should be same.

Events

Event is a multi-cast delegate that stores one or more methods; and invoke them every time when the event is raised (called).

The event can be raised only in the same class, in which it is created.



- Publisher class is a class that sends (or raises) events (notifications), is called as "publisher class".
- Publisher class sends events; then Subscriber class receives events.

- Subscriber class is a class that receives (or subscribes or handles) events (notifications), is called as "subscriber class".

Events enable a class to send notifications to other classes, when something occurs.

Publisher class sends events; Subscriber class receives events.

Process of Events

1. The Publisher class creates an event.
2. The Subscriber class subscribes to the event; that means an "event handler" method is created in the subscriber class. The "event handler" method is nothing but, the method which is dedicated to be executed when the event is raised.
3. The publisher class can send (raise) events.
4. Every time, when the event is raised by the publisher, the corresponding "event handler" method executes automatically.

Creating Events

1. Create a Delegate

```
public delegate ReturnType DelegateTypeName(param1, param2, ...);
```

2. Create an Event in Publisher class

```
class Publisher
{
  private DelegateTypeName eventVariable;

  public event DelegateTypeName EventName
  {
    add
    {
      eventVariable += value;
    }

    remove
    {
      eventVariable -= value;
    }
}
```

3. Raise the event in Publishder class

```
if (EventName != null)
   EventName(arg1, arg2, ...);
```

4. Create Event Handler Method in Subscriber class

```
class Subscriber
{
  public ReturnType EventHandlerMethodName(param1, param2, ...)
  {
    Method body here
  }
}
```

5. Subscribe to the Event (Inside or outside the subscriber class)

```
EventName += EventHandlerMethodName;
```

Rules of Events

- The event should be created based on the delegate. That means, the event accepts the methods that are having specific parameters and return type, defined in the delegate.
- An event can have multiple subscribers.
- A subscriber can subscribe multiple events from multiple publishers.
- Events are basically signals to inform to other classes, that some important thing happened in the publisher class.
- Events are special kind of "multi-cast delegates", which can be raised only within the same class, in which they are created.
- Events can be static, virtual, sealed and abstract.
- Events will not be raised (throws exception), if there is no at least one subscriber.
- Events can be defined in interfaces.
- It's not a good idea to return value in events.

Auto-Implemented Events

"Auto-Implemented Events" provide a shortcut syntax to create events with less code.

In this case, you need not create "add" and "remove" accessors; the compiler does the same automatically.

Create an Auto-Implemented Event in Publisher Class

```
class Publisher
{
}
```

You also not required to create a private multi-cast delegate; the compiler does the same automatically.

Disadvantage / Limitation: We can't define custom logic for "add accessor" and "remove accessor".

Anonymous Methods

Anonymous methods are "name-less methods", that can be invoked by using the delegate variable or an event.

Subscribe to Event with Anonymous Method:

```
1. EventName += delegate(param1, param2, …)
2. {
3.   //method body here
4. }
```

Anonymous methods can be used anywhere within the method, to create methods instantly, without define a method at the class level.

Advantage: We need not create a "named method (normal method)" to quickly handle an event.

Rules:

- It can't be called without a delegate or event.
- It can't contain jump statements like goto, break, continue.
- It can access local variables and parameters of outer method.
- It can be passed as a parameter to any method; in this case, the delegate acts as data type for the anonymous method.
- It can't access ref or out parameter of an outer method.
- It is mainly used for event handlers.

Lambda Expressions

"Lambda Expressions" (a.k.a. Statement Lambda) are "name-less methods", that can be invoked by using the delegate variable or an event, much like anonymous methods.

Handle Event with Lambda Expressions:

```
1. EventName += (param1, param2, …) =>
2. {
3.   //method body here
4. }
```

Lambda Expressions can be used anywhere within the method, to create methods instantly, without define a method at the class level.

Advantage: It provides more easier and convenient syntax than "Anonymous methods".

=> operator is called as "goes to" or "goes into" operator.

Inline Lambda Expressions

"Inline Lambda Expressions" (a.k.a. Expression Lambda) are the lambda expressions, which performs a small calculation or condition check and returns a value.

Inline lambdas can receive one or more arguments and must return a value.

Advantage: It provides more easier and convenient syntax to create smaller methods that performs a single calculation or condition check.

Handle Event with Inline Lambda Expressions:

```
EventName += (param1, param2, …) => condition or calculation
```

Expression Bodied Members

"Expression Bodied Members" concept allows the developer to use "Inline Lambda Expressions" to create methods, property accessors, constructors, destructors, indexers in a class.

Method using Expression Bodied Members - without return value:

```
public ReturnType MethodName( ) => statement;
```

Method using Expression Bodied Methods - with return value

```
public ReturnType MethodName( ) => AnyValue;
```

Expression Bodied Members may have or parameters; may / may not have return value.

Expression Bodied Members can have only one statement.

Advantage: It provides more easier and convenient syntax to create smaller methods that performs a single calculation or condition check.

Expression Bodied Members - Usage

Constructor with Expression Bodied Members:

```
public ClassName(param1) => field = param1;
```

Property with Expression Bodied Members:

```
1. public type Property
2. {
3.   set => field = value;
4.   get => field;
5. }
```

Switch Expression

```
1. sourceVariable switch
2. {
3.   value1 => result1,
4.   value2 => result2,
5.   …
6.   _ => defaultResult
7. }
```

The variable used in switch expression is now coming before the switch keyword.

Colon (:) and case keyword are replaced with arrows (=>). Which makes the code more compact and readable.

The default case is now replaced with a discard(_).

And the body of the switch is expression, not a statement.

'Switch Expression' returns the result value based on the matching case.

It is only meant for getting a specific result value; doesn't let you to write multiple statements.

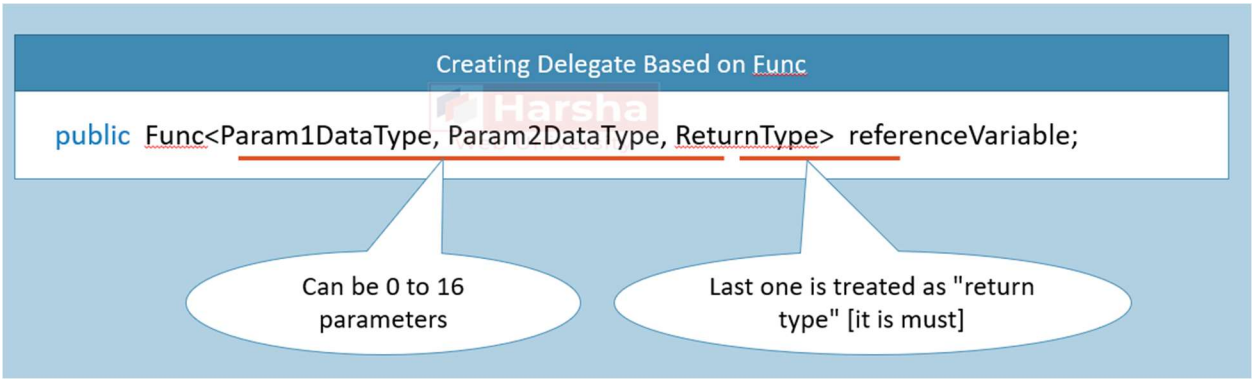
Func

"Func" is a pre-defined generic-delegate, which can be used to create events quickly.

Func supports parameters and return value also.

- Func must have 0 to 16 parameters.

- Func must have return value.



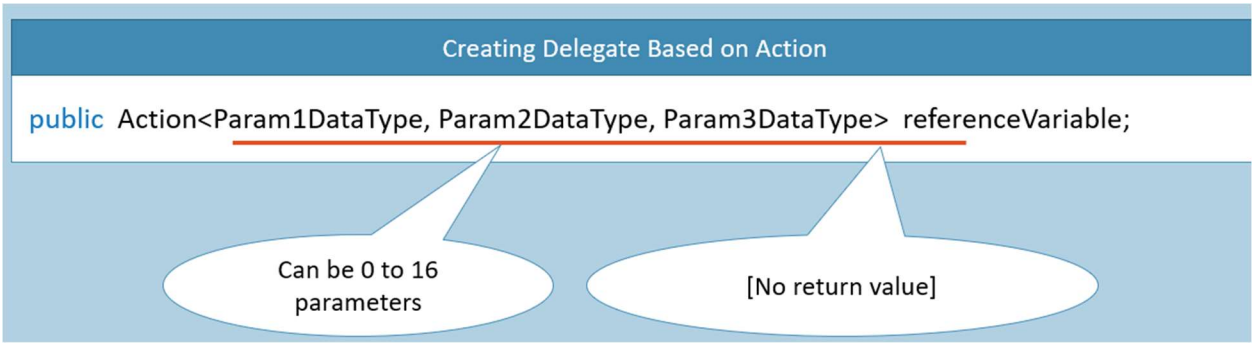
Action

"Action" is a pre-defined delegate, which can be used to create events quickly, similar to "Func".

The difference is:

1. Func must have return value; Action don't have return value.

2. Action must have 0 to 16 parameters.



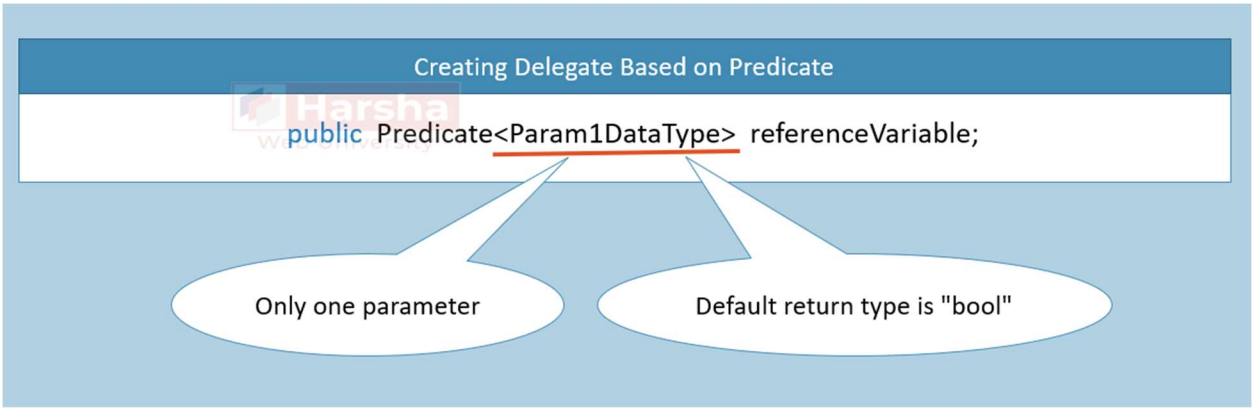
Predicate

"Predicate" is a pre-defined delegate, which can be used to create events quickly, similar to "Func".

The difference is:

1. Func must have return value of any type; Action don't have return value; Predicate must have return value of "bool" type.

2. Func can have 0 to 16 parameters of any type; Action can have 0 to 16 parameters of any type; Predicate must have only one parameter of any type.



EventHandler

'EventHandler' is a pre-defined delegate type, which has two parameters called "object sender" and "EventArgs e"; and no return.

object sender: Represents the source object, where the from where the event is originally raised.

EventArgs e: Represents additional parameters to pass to 'event handler method'. It is recommended to create a child class for 'EventArgs' class.

Creating Event Based on EventHandler:

```
public event EventHandler EventName;
```

Expression Trees

Expression Tree is a collection of delegates represented in tree-like structure.

Expression Tree only executes when we compile and execute it.

Expression Trees support all delegate types such as Func, Action, Predicate or custom delegate types.

1. Creating Expression Tree based on Func:

```
Expression< Func<type1, type2, …> > referenceVariable;
```

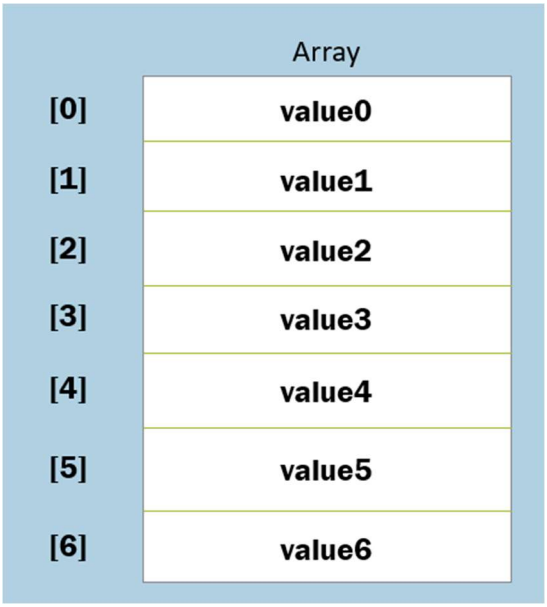
2. Compile and Execute Expression Tree:

```
1. Func<type1, type2, …> referenceVariable2 = referenceVariable.Compile( );
2. referenceVariable2.Invoke(arg1, arg2, …);
```

Arrays

Array is a group of multiple values of same type.

Arrays are stored in continuous-memory-locations in 'heap'.



Syntax:

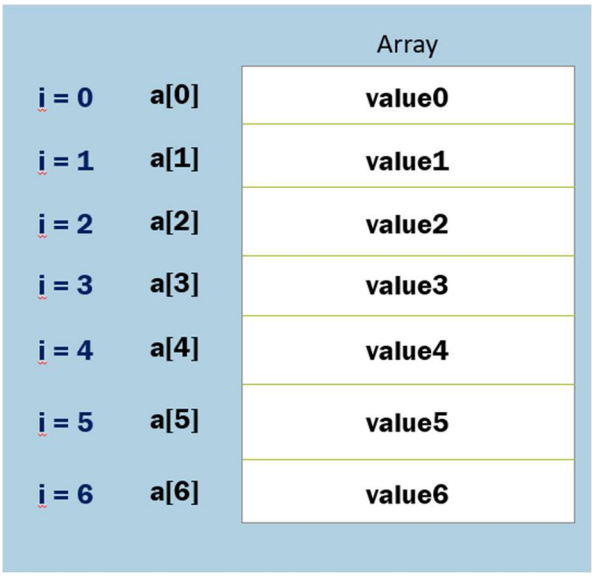
```
type[ ] arrayReferenceVariableName = new type[ size ];
```

- Each value of array is called "element".
- All the elements are stored in continuous memory locations.
- The address of first element of the array will be stored in the "array reference variable".
- The "Length" property stores count of elements of array. Index starts from 0 (zero).
- Arrays are treated as objects of "System.Array" class; so arrays are stored in heap; its address (first element's address) is stored in reference variable at stack.

Array 'for' loop

For loop starts with an "initialization"; checks the condition; executes the loop body and then performs incrementation or decrementation;

Use "for loop" to read elements from an array, based on index.



Syntax:

```
for (int i = 0; i < arrayRefVariable.Length; i++)
{
  arrayRefVariable[i]
}
```

Pros of Array 'for' loop

- You can read any part of the array (all elements, start from specific element, end with specific element).
- You can read array elements in reverse.

Cons of Array 'for' loop:

- A bit complex syntax.

Array 'foreach' loop

Foreach loop starts contains a "iteration variable"; reads each value of an array or collection and assigns to the "iteration variable", till end of the array / collection.

"Foreach loop" is not based on index; it is based on sequence.

Syntax:

```
foreach (DataType iterationVariable in arrayVariable)
{
   iterationVariable
}
```

Pros of Array 'Foreach' loop

- Simplified Syntax
- Easy to use with arrays and collections.
- It internally uses "Iterators".

Cons of Array 'Foreach' loop

- Slower performance, due to it treats everything as a collection.
- It can't be used to execute repeatedly, without arrays or collections.
- It can't read part of array / collection, or read array / collection reverse.

'System.Array' class

Every array is treated as an object for System.Array class.

The System.Array class provides a set of properties and methods for every array.

Properties of 'System.Array' class

1. Length

Methods of 'System.Array' class

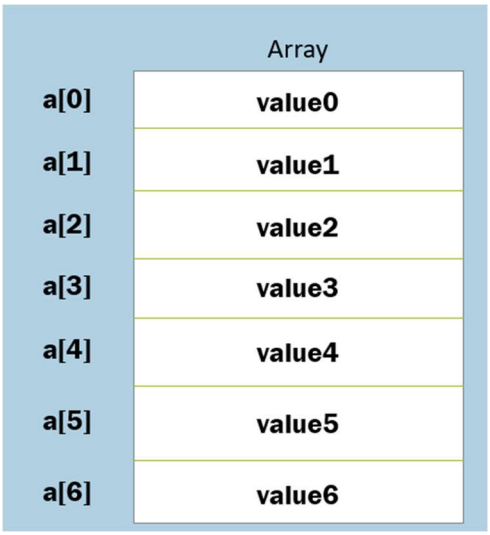
1. IndexOf
2. BinarySearch
3. Clear
4. Resize
5. Sort
6. Reverse
7. CopyTo
8. Clone

Array - IndexOf( ) method

This method searches the array for the given value.

- If the value is found, it returns its index.

- If the value is not found, it returns -1.



Signature:

```
static int Array.IndexOf( System.Array array, object value)
```

Example:

```
Array.IndexOf(array, value3) = 3
```

The “IndexOf” method performs linear search. That means it searches all the elements of an array, until the search value is found. When the search value is found in the array, it stops searching and returns its index.

The linear search has good performance, if the array is small. But if the array is larger, Binary search is recommended to improve the performance

Parameters:

array: This parameter represents the array, in which you want to search.

value: This parameter represents the actual value that is to be searched.

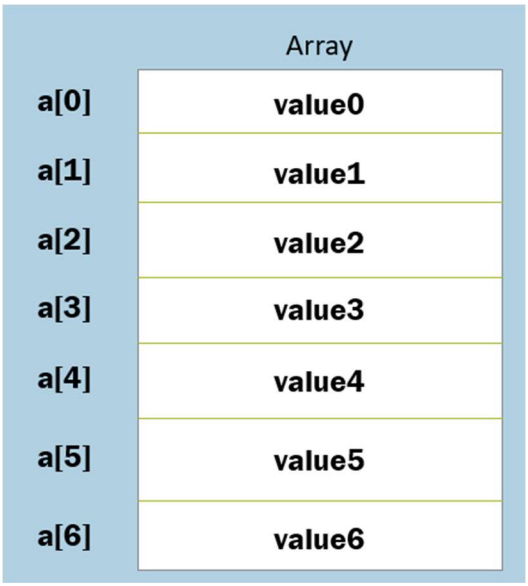
startIndex: This parameter represents the start index, from where the search should be started.

Array - BinarySearch( ) method

This method searches the array for the given value.

- If the value is found, it returns its index.

- If the value is not found, it returns -1.



Signature:

```
static int Array.BinarySearch( System.Array array, object value)
```

Example:

```
Array.BinarySearch(array, value3) = 3
```

The “Binary Search” requires an array, which is already sorted. On unsorted arrays, binary search is not possible.

It directly goes to the middle of the array (array size / 2), and checks that item is less than / greater than the search value.

If that item is greater than the search value, it searches only in the first half of the array.

If that item is less than the search value, it searches only in the second half of the array.

Thus it searches only half of the array. So in this way, it saves performance

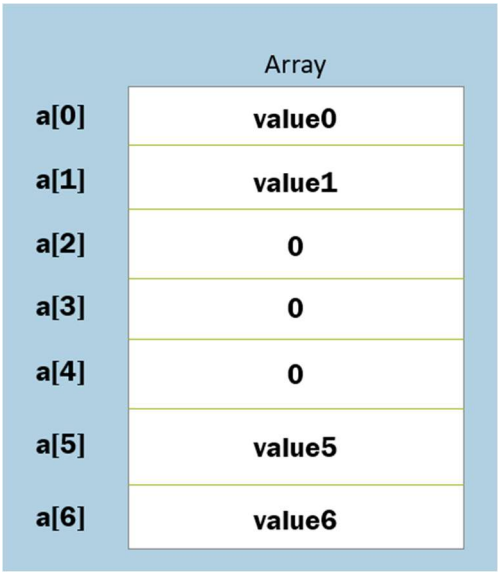
Parameters:

array: This parameter represents the array, in which you want to search.

value: This parameter represents the actual value that is to be searched

Array - Clear( ) method

This method starts with the given index and sets all the “length” no. of elements to zero (0)



Signature:

```
static void Array.Clear( System.Array array, int index, int length)
```

Example:

```
Array.Clear(a, 2, 3)
```

Parameters:

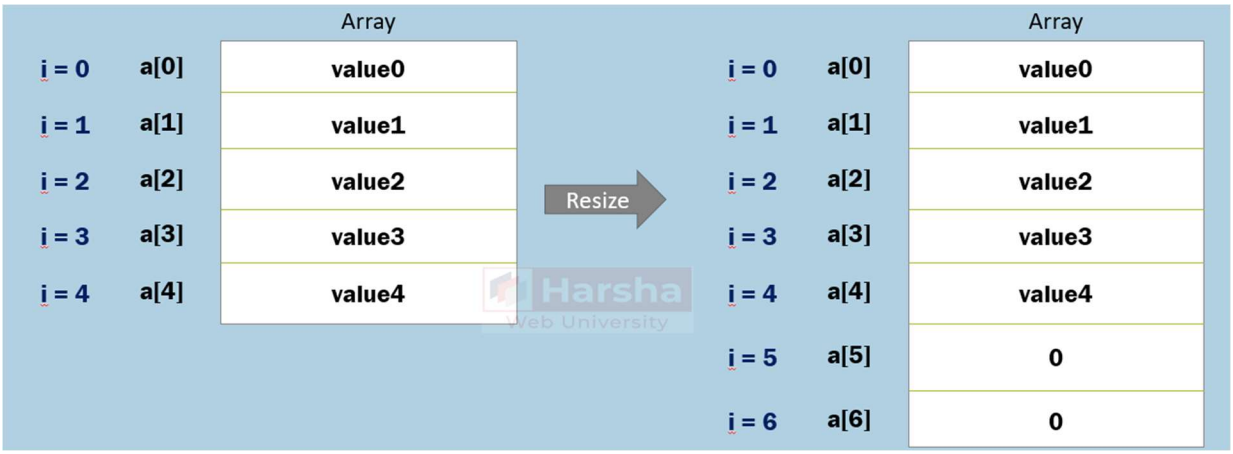
array: This parameter represents the array, in which you want to clear the elements.

index: This parameter represents the index, from which clearing process is to be started.

length: This parameter represents the no. of elements that are to be cleared.

Array - Resize( ) method

This method increases / decreases size of the array.



Signature:

```
static void Array.Resize(ref System.Array array, int newSize)
```

Example:

```
Array.Resize(a, 6)
```

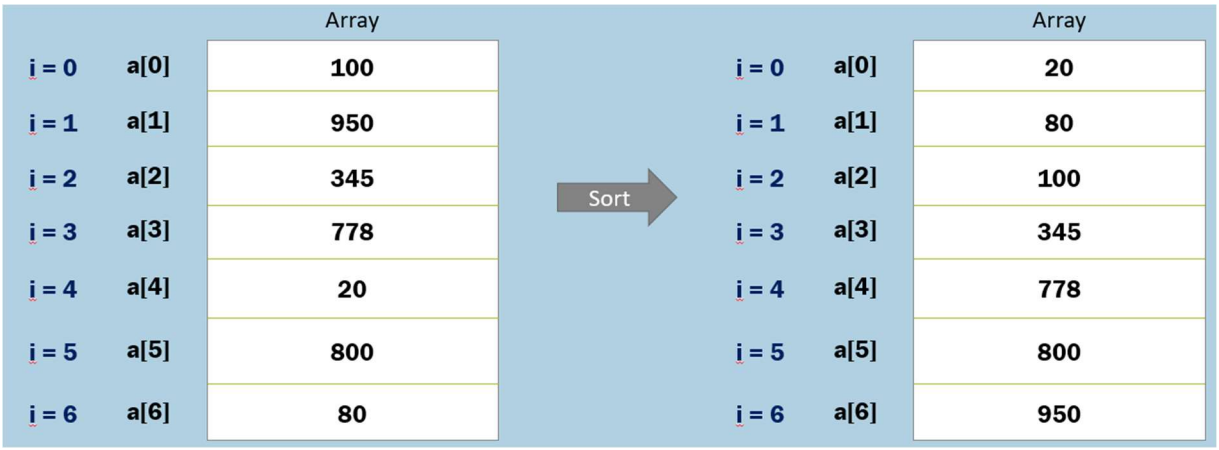
Parameters:

array: This parameter represents the array, which you want to resize.

newSize: This parameter represents the new size of the array, how many elements you want to store in the array. It can be less than or greater than the current size.

Array - Sort( ) method

This method sorts the array in ascending order.



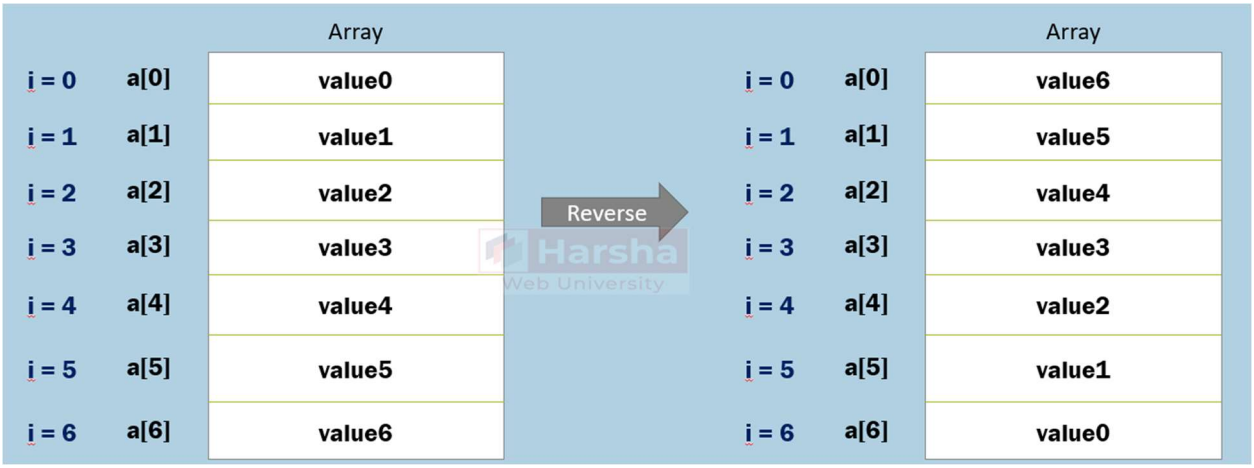
Signature:

```
static void System.Array.Sort( System.Array array )
```

Example:

```
Array.Sort(a)
```

Array - Reverse( ) method



Signature:

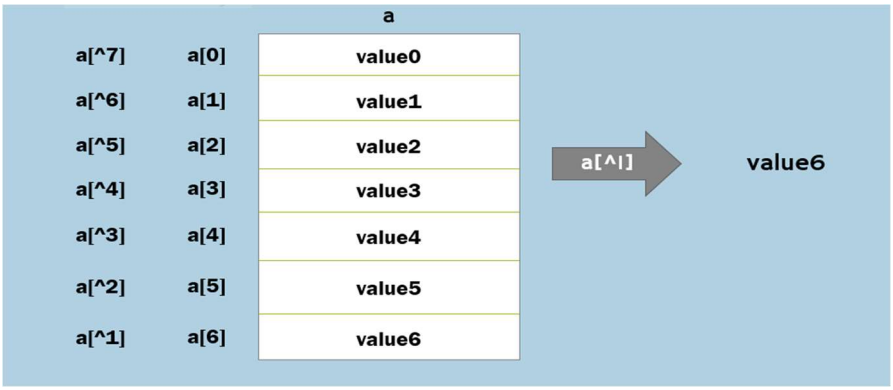
```
static void System.Array.Reverse( System.Array array )
```

Example:

```
Array.Reverse(a)
```

'IndexFromEnd' operator

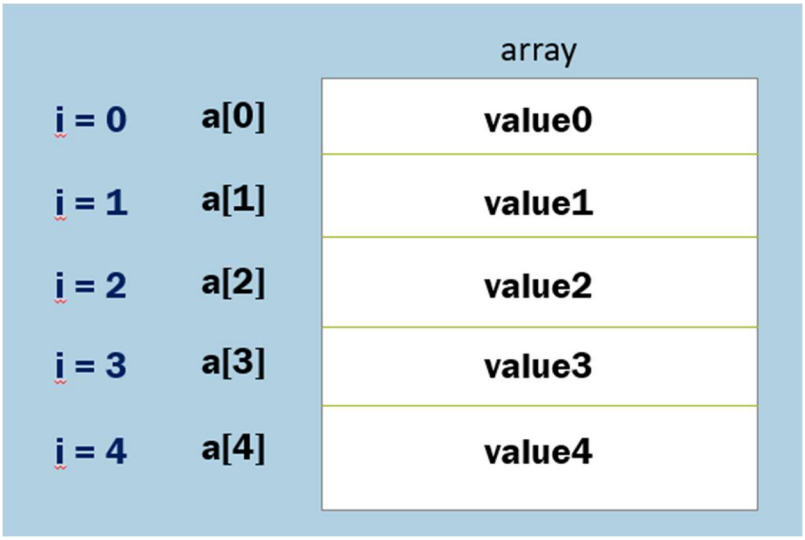
This operator returns the index from end of the array (last element is treated as index '0').



Types of Arrays

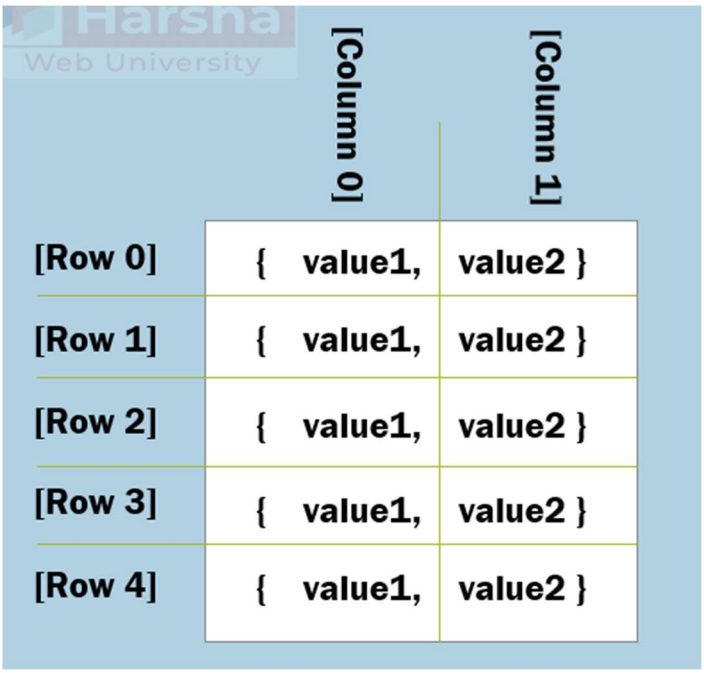
Single-Dim Arrays

Group of multiple rows; each row contains a single value.



Multi-Dim Arrays

Group of multiple rows; each row contains a group of values.

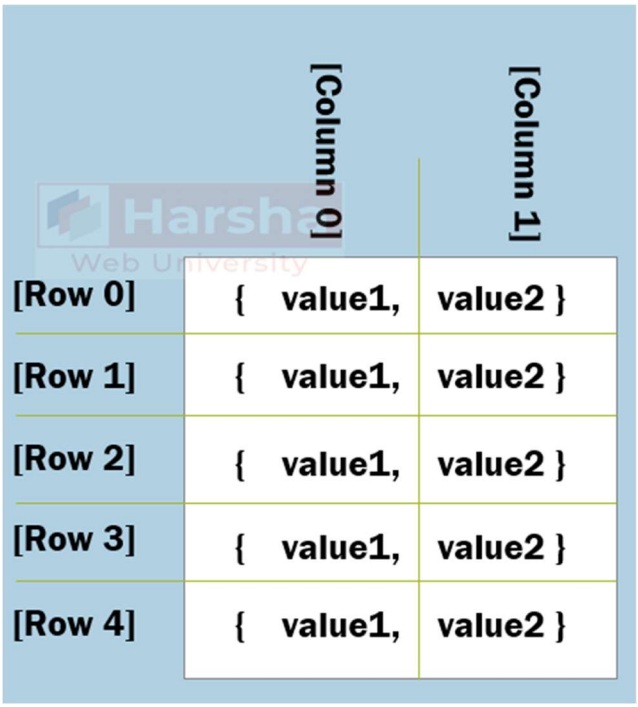


Multi-Dim Arrays

Stores elements in rows & columns format.

Every row contains a series of elements.

You can create arrays with two or dimensions, by increasing the no. of commas (,).



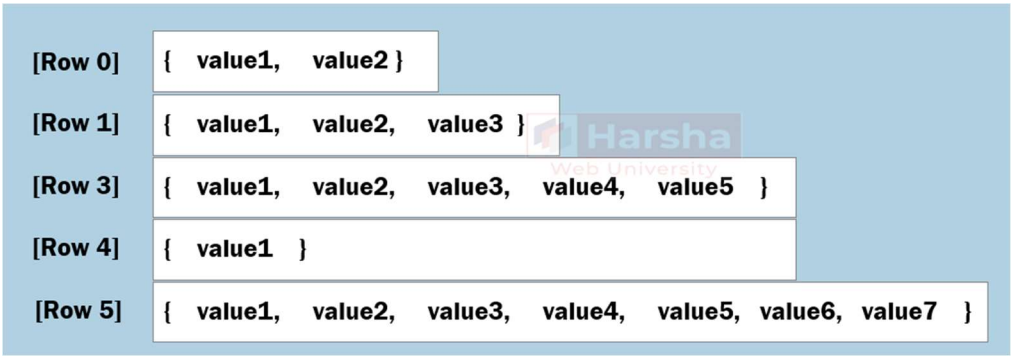
Syntax:

```
type[ , ] arrayReferenceVariable = new type[ rowSize, columnSize ];
```

Jagged Arrays

Jagged Array is an “array of arrays”.

The member arrays can be of any size.

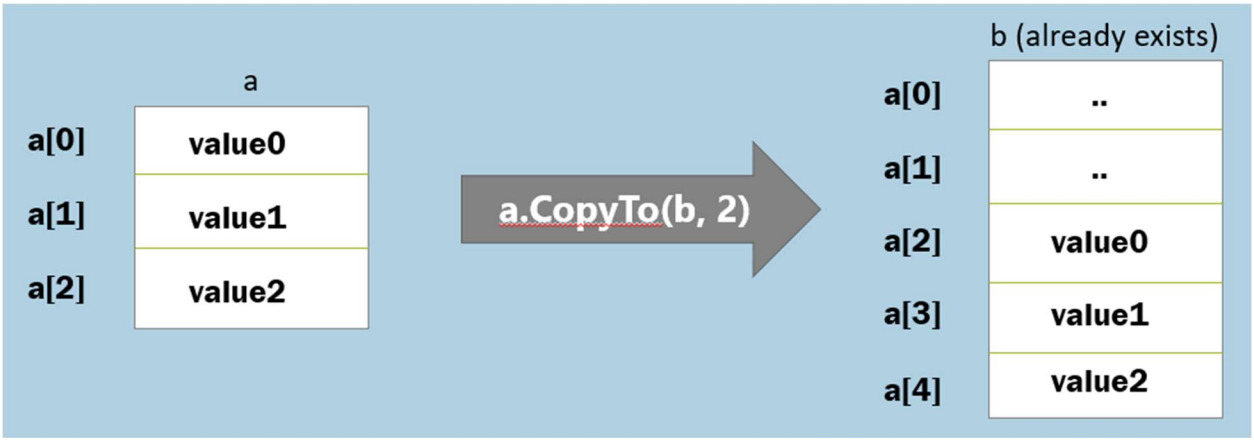


Syntax:

```
1. type[ ] [ ] arrayReferenceVariable = new type[ rowSize ] [ ];
2. arrayReferenceVariable[index] = new type[ size ];
```

Array - CopyTo( ) method

This method copies (shallow copy) all the elements from source array to destination array, starting from the specified 'startIndex'.



Signature:

```
void Array.CopyTo(System.Array destinationArray, int startIndex)
```

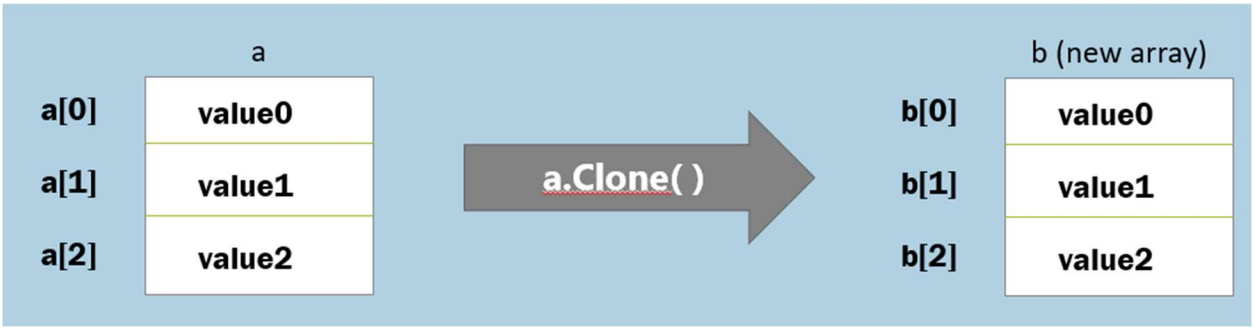
Parameters:

sourceArray: This parameter represents the array, which array you want to copy.

destinationArray: This parameter represents the array, into which you want to copy the elements of source array. The destination array must exist already and should be large enough to hold new values.

startIndex: This parameter represents the index of the element, from which you want to start copying.

Array - Clone( ) method



Signature:

```
object System.Array.Clone()
```

Array.CopyTo()

1. CopyTo() equires to have an existing destination array; and the destination array should be large enough to hold all elements from the source array, starting from the specified startIndex.
2. CopyTo() allows you to specify the startIndex at destination array.
3. The result array need not be type-casted explicitly.

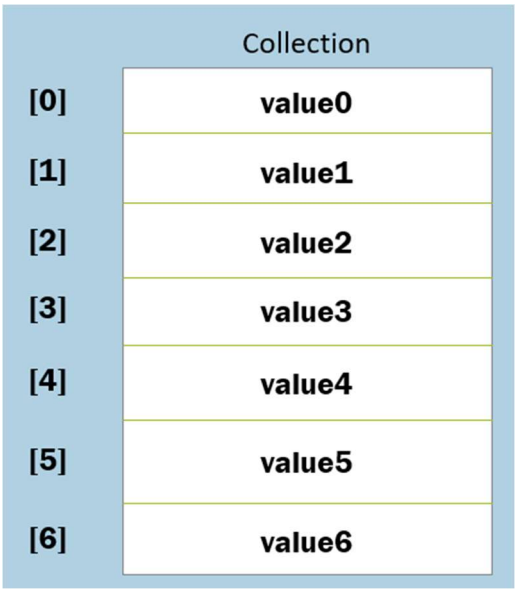
Array.Clone()

1. Clone() creates a new destination array; you need not have an existing array.
2. Clone() doesn't allow you to specify the startIndex at destination array.
3. The result array will be returned as 'object' type; so need to be type-casted to array type.

Collections

Collections are the standard-way to store and manipulate group of elements (primitive values or objects).

Collections are internally objects of specific 'collection classes' such as List, Dictionary, SortedList etc.



Syntax:

```
List<type> referenceVariable = new List<type>( );
```

Collections can store unlimited elements.

- You can add, remove elements at any time.

- You need not specify the size (no. of elements) while creating collection.

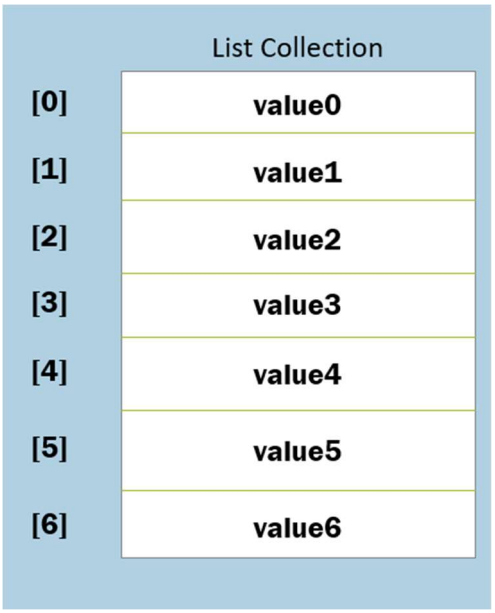
- You can search, sort, copy collections using various built-in methods.

'List' Collection

List collection contains a group of elements of same type.

Full Path: System.Collections.Generic.List

The 'List' class is a generic class; so you need to specify data type of value while creating object.



Syntax:

```
List<type> referenceVariable = new List<type>( );
```

Features of 'List' class

- It is dynamically sized. You can add, remove elements at any time.
- It allows duplicate values.
- It is index-based. You need to access elements by using zero-based index.
- It is not sorted by default. The elements are stored in the same order, how they are initialized.
- It uses arrays internally; that means, recreates array when the element is added / removed. The 'Capacity' property holds the number of elements that can be stored in the internal array of the List. If you add more elements, the internal array will resized to the 'Count' of elements.

Properties of 'List' class

1. Count
2. Capacity

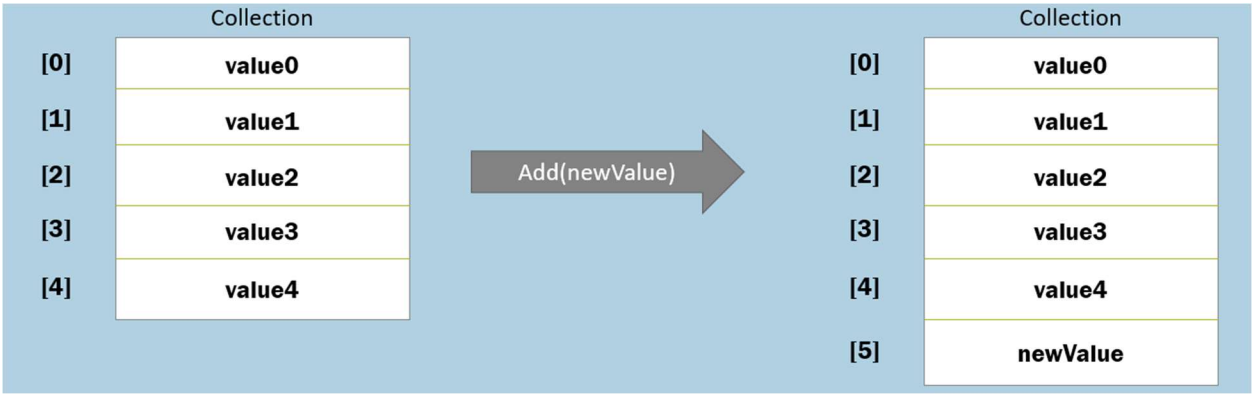
Methods of 'List' class

1. Add(T)
2. AddRange(IEnumerable<T>)
3. Insert(int, T)
4. InsertRange(int, IEnumerable<T>)
5. Remove(T)
6. RemoveAt(int)

7. RemoveRange(int, int)
8. RemoveAll(Predicate<T>)
9. Clear()
10. IndexOf(T)
11. BinarySearch(T)
12. Contains(T)
13. Sort()
14. Reverse()
15. ToArray()
16. ForEach(Action<T>)
17. Exists(Predicate<T>)
18. Find(Predicate<T>)
19. FindIndex(Predicate<T>)
20. FindLast(Predicate<T>)
21. FindLastIndex(Predicate<T>)
22. FindAll(Predicate<T>)
23. ConvertAll(T)

List - Add( ) method

This method adds a new element to the collection.



Syntax:

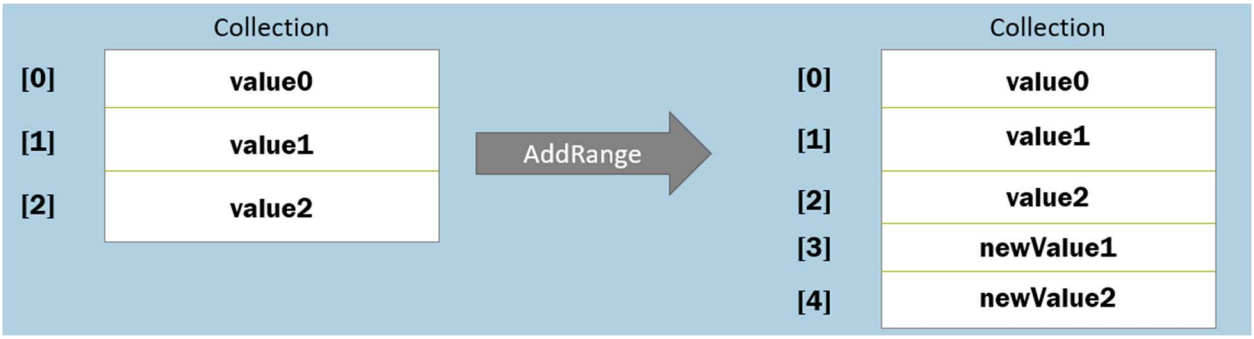
```
void List.Add(T newValue)
```

Syntax:

```
List.Add(10)
```

List - AddRange( ) method

This method adds a new set of elements to the collection.



Syntax:

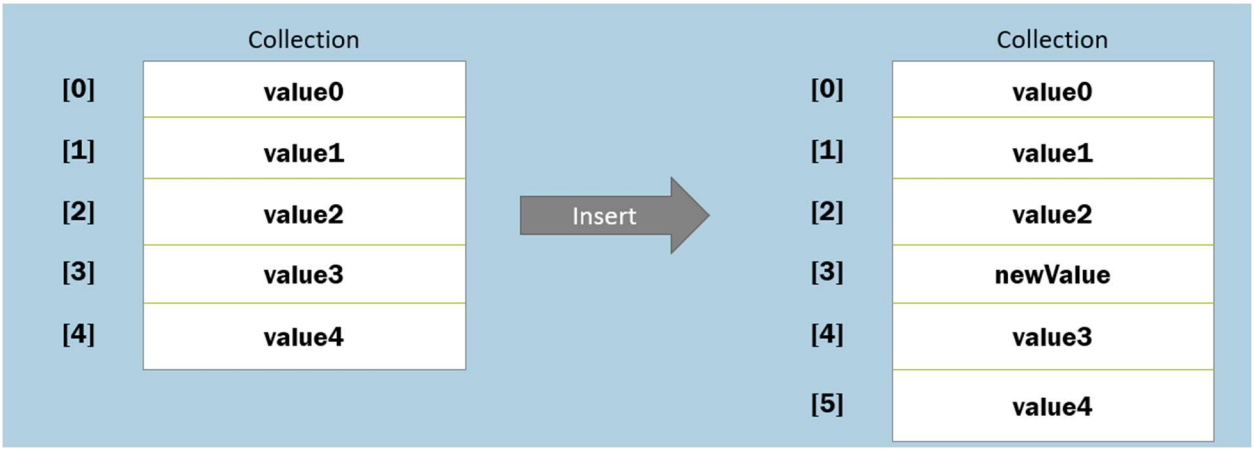
```
void List.AddRange(IEnumerable<T> newValue)
```

Example:

```
List.AddRange(new List<int>( ) { newValue1, newValue2 } )
```

List - Insert( ) method

This method adds a new element to the collection at the specified index.



Syntax:

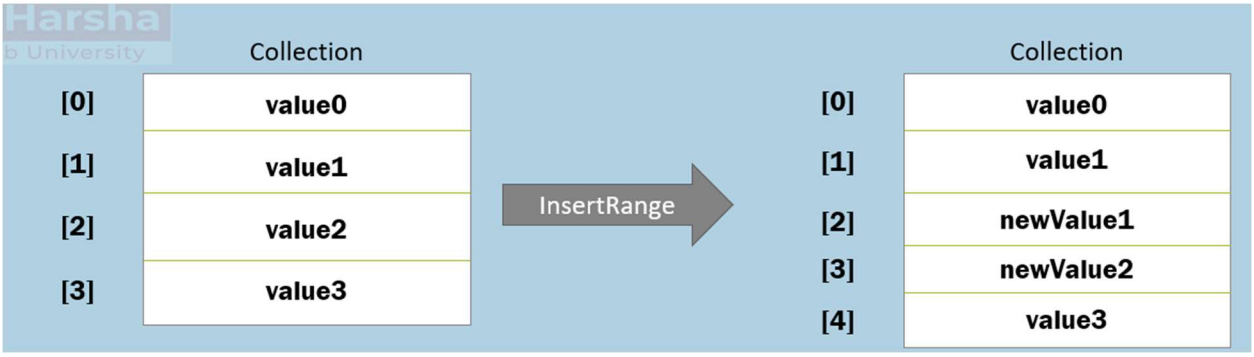
```
void List.Insert(int index, T newValue)
```

Example:

```
List.Insert(3, newValue)
```

List - InsertRange( ) method

This method adds a new set of elements to the collection at the specified index.



Syntax:

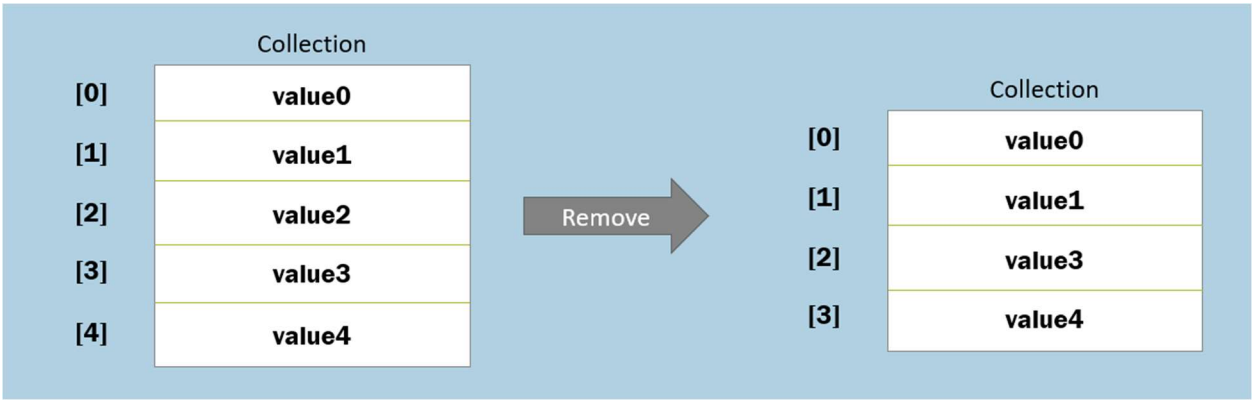
```
void List.InsertRange(int index, IEnumerable<T> newValue)
```

Example:

```
List.InsertRange(2, new List<int>() { newValue1, newValue2 } )
```

List - Remove( ) method

This method removes the specified element from the collection.



Syntax:

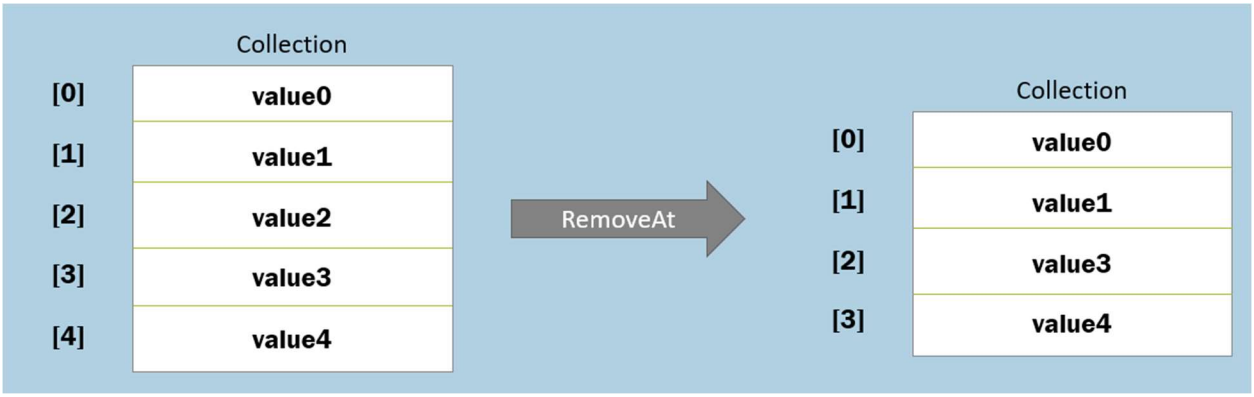
```
void List.Remove(T newValue
```

Example:

```
List.Remove(value2)
```

List - RemoveAt( ) method

This method removes an element from the collection at the specified index.



Syntax:

```
void List.RemoveAt(int index)
```

Example:

```
List.RemoveAt(2)
```

List - RemoveRange( ) method

This method removes specified count of elements starting from the specified startIndex.

Syntax:

```
void List.RemoveRange(int index, int count)
```

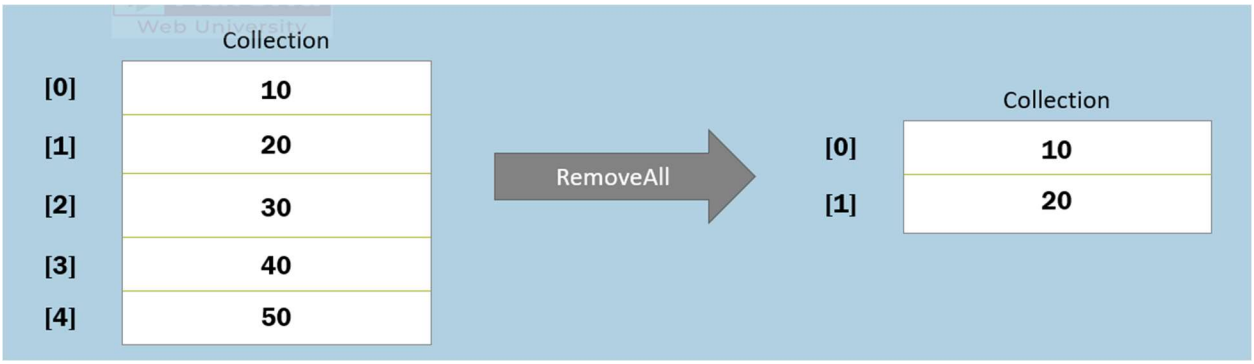
Example:

```
List.RemoveRange(1, 2 )
```

List - RemoveAll( ) method

This method removes all the elements that are matching with the given condition.

You can write your condition in the lambda expression of Predicate type.



Syntax:

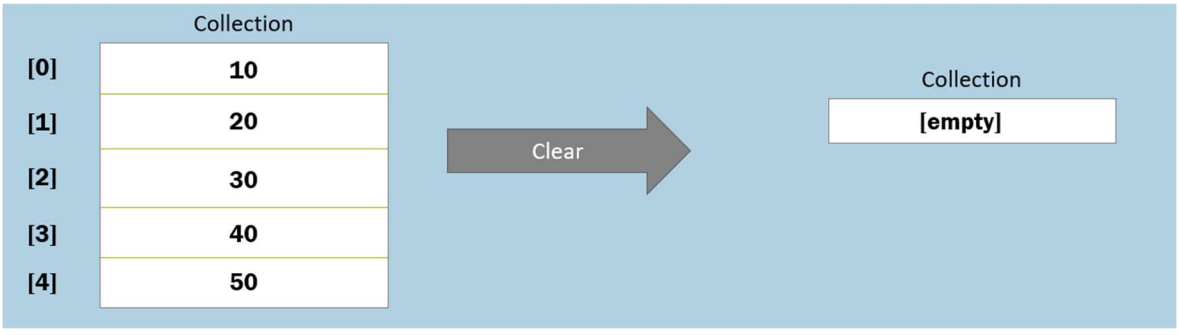
```
void List.RemoveAll(value => condition)
```

Example:

```
List.RemoveAll( n => n >= 30 )
```

List - Clear( ) method

This methods removes all elements in the collection.



Syntax:

```
void List.Clear( )
```

Example:

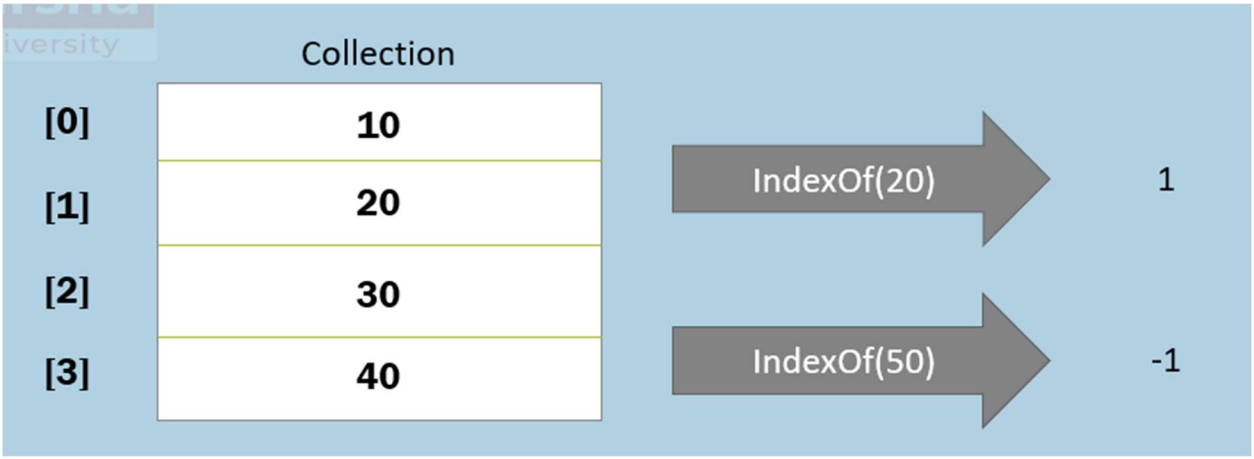
```
List.Clear( )
```

List - IndexOf( ) method

This method searches the collection for the given value.

If the value is found, it returns its index.

If the value is not found, it returns -1.



Syntax:

```
int List.IndexOf(T value, int startIndex)
```

Example:

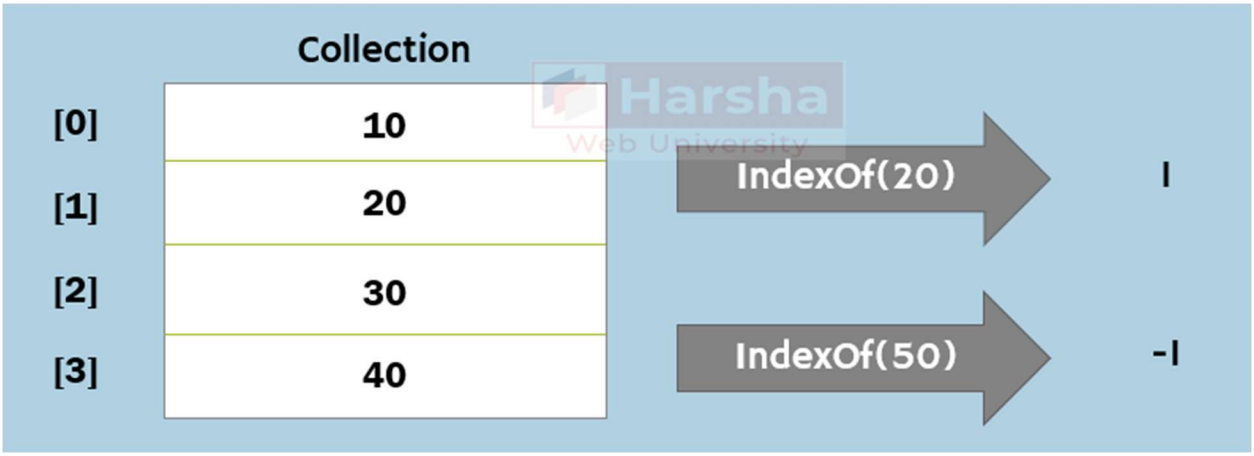
```
List.IndexOf(20)
```

List.IndexOf( ) method

This method searches the collection for the given value.

- If the value is found, it returns its index.

- If the value is not found, it returns -1.



Syntax:

```
int List.IndexOf(T value, int startIndex)
```

Example:

```
List.IndexOf(20)
```

The “IndexOf” method performs linear search. That means it searches all the elements of the collection, until the search value is found. When the search value is found in the collection, it stops searching and returns its index.

The linear search has good performance, if the collection is small. But if the collection is larger, Binary search is recommended to improve the performance.

Parameters:

value: This parameter represents the actual value that is to be searched.

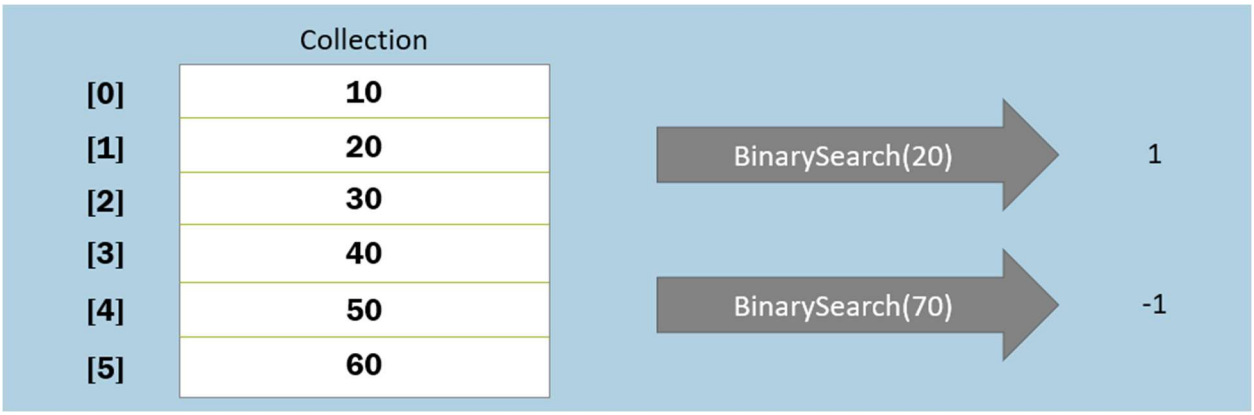
startIndex: This parameter represents the start index, from where the search should be started.

List - BinarySearch( ) method

This method searches the array for the given value.

- If the value is found, it returns its index.

- If the value is not found, it returns -1.



Syntax:

```
int List.BinarySearch( T value)
```

Example:

```
List.BinarySearch(20)
```

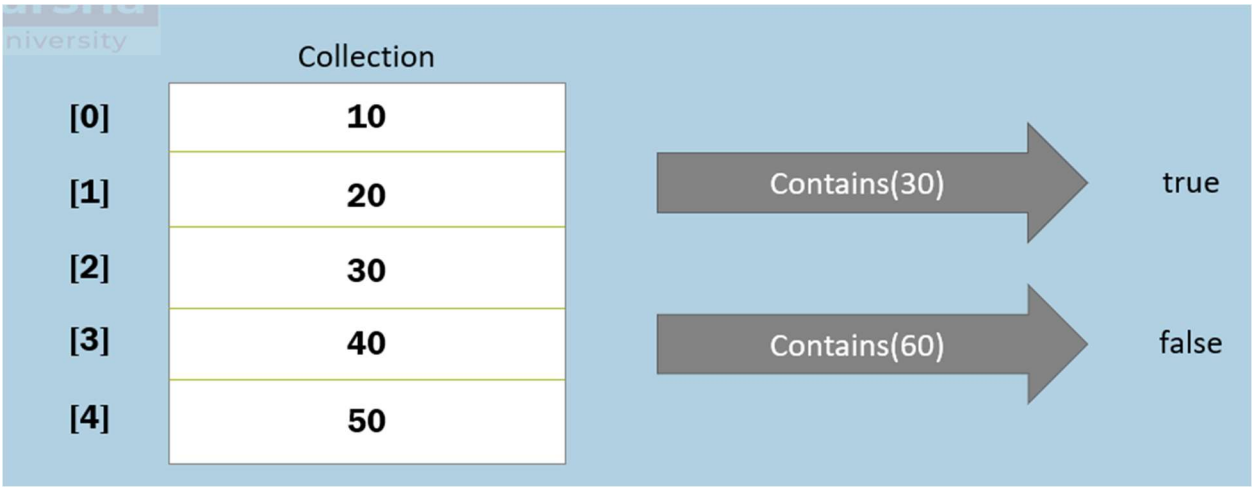
- The “Binary Search” requires a collection, which is already sorted.
- On unsorted collections, binary search is not possible.
- It directly goes to the middle of the collection (collection size / 2), and checks that item is less than / greater than the search value.
- If that item is greater than the search value, it searches only in the first half of the collection.
- If that item is less than the search value, it searches only in the second half of the array.
- Thus it searches only half of the array. So in this way, it improves performance

Parameters:

value: This parameter represents the actual value that is to be searched.

List - Contains( ) method

This method searches the specified element and returns 'true', if it is found; but returns 'false', if it is not found.



Syntax:

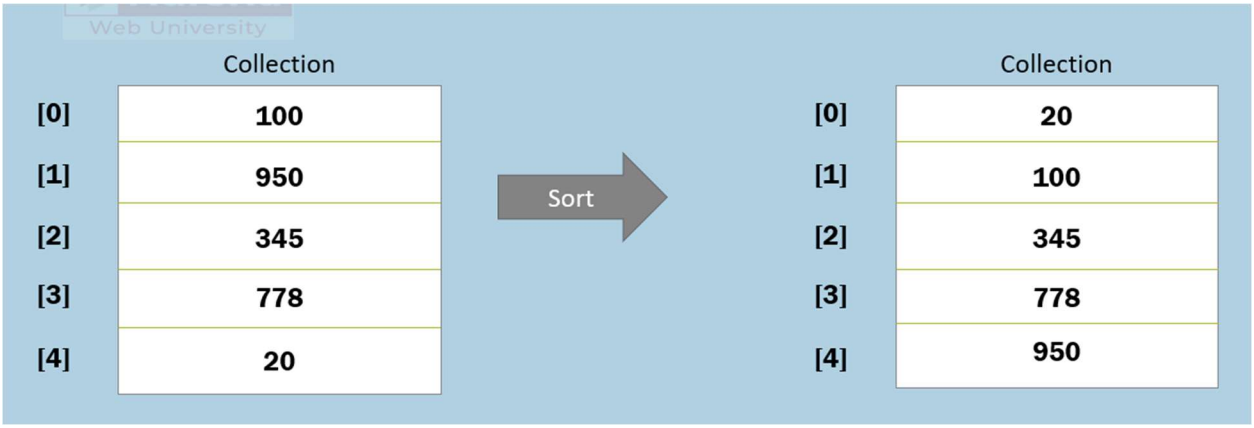
```
bool List.Contains( T value )
```

Example:

```
List.Contains(30)
```

List - Sort( ) method

This method sorts the collection in ascending order.



Syntax:

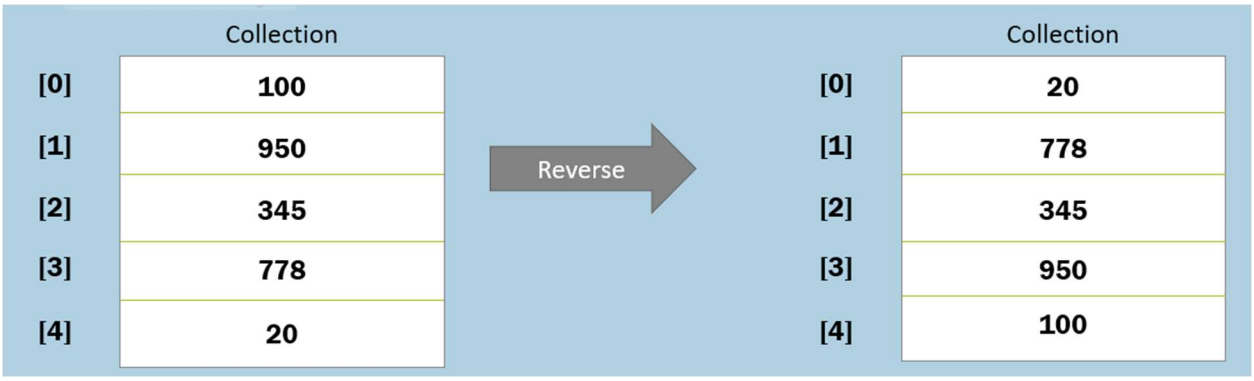
```
void List.Sort( )
```

Example:

```
List.Sort( )
```

List - Reverse( ) method

This method reverses the collection.



Syntax:

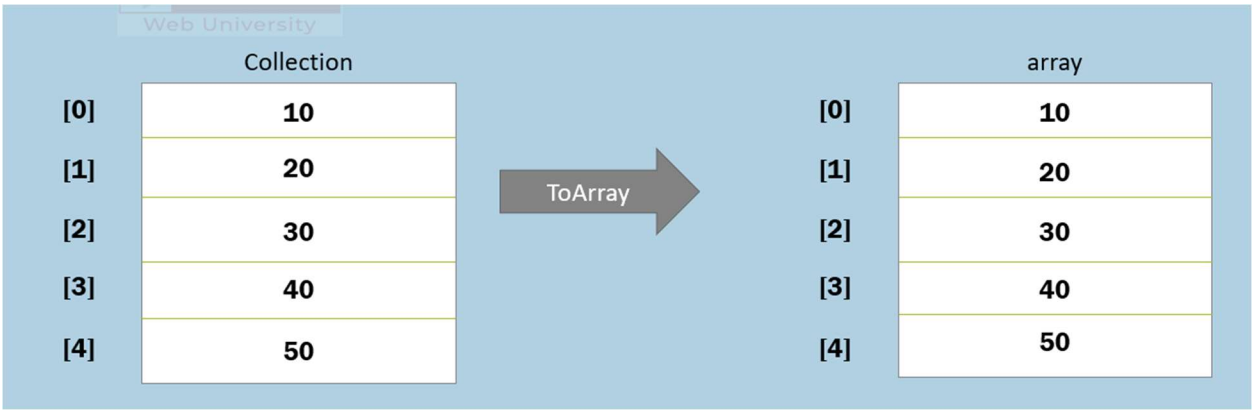
```
void List.Reverse( )
```

Example:

```
List.Reverse( )
```

List - ToArray( ) method

This method converts the collection into an array with same elements.



Syntax:

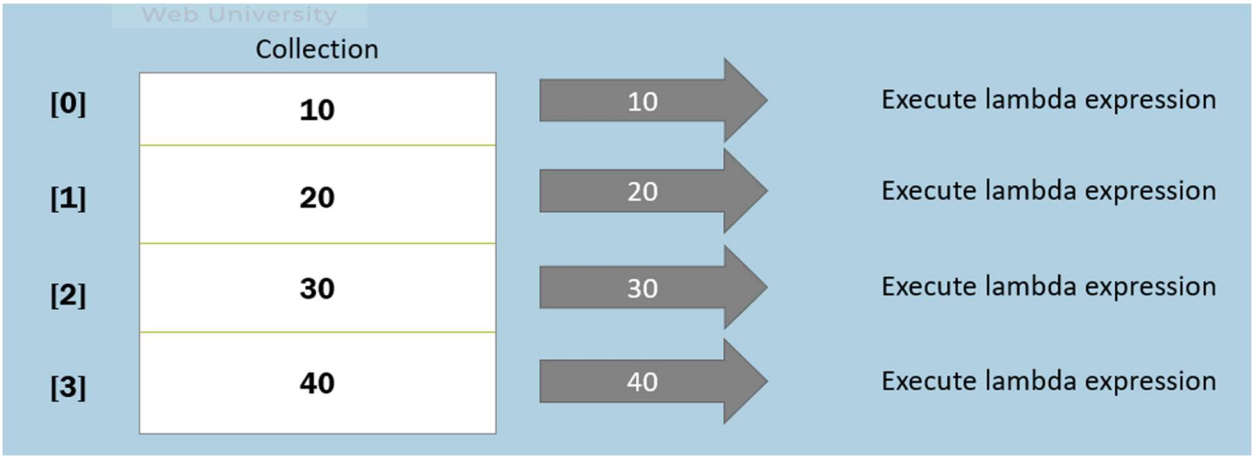
```
T[ ] List.ToArray( )
```

Example:

```
List.ToArray( )
```

List - ForEach( ) method

This method executes the lambda expression once per each element.



Syntax:

```
void List.ForEach( Action<T> )
```

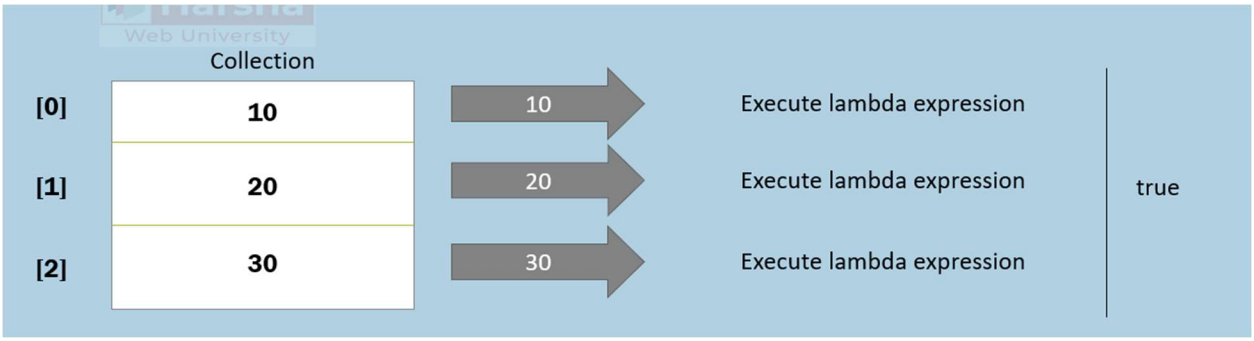
Example:

```
List.ForEach( n => { Console.WriteLine(n); } )
```

List - Exists( ) method

This method executes the lambda expression once per each element.

It returns true, if at least one element matches with the given condition; but returns false, if no element matches with the given condition.



Syntax:

```
bool List.Exists( Predicate<T> )
```

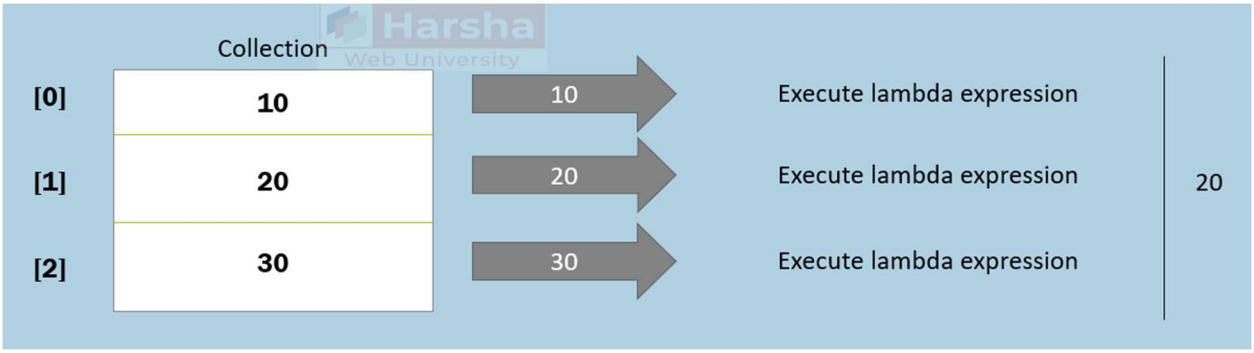
Example:

```
List.Exists( n => n > 15 )
```

List - Find( ) method

This method executes the lambda expression once per each element.

It returns the first matching element, if at least one element matches with the given condition; but returns the default value, if no element matches with the given condition.



Syntax:

```
T List.Find( Predicate<T> )
```

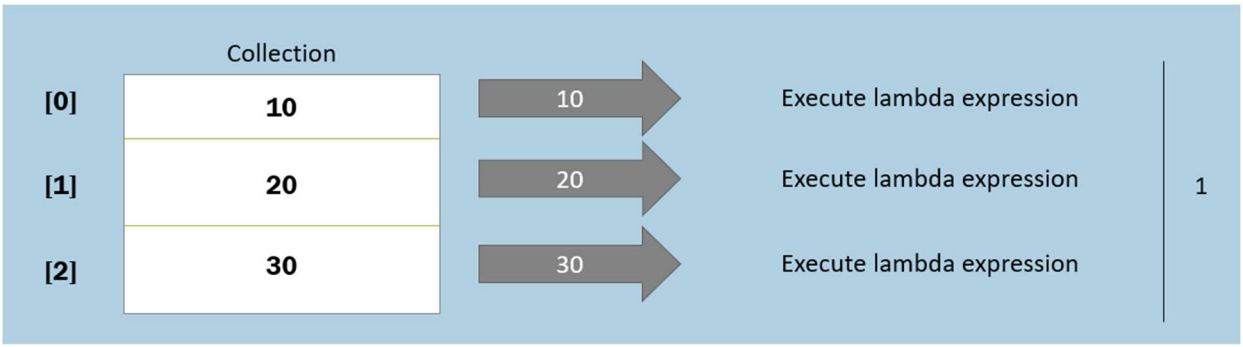
Example:

```
List.Find( n => n > 15 )
```

List - FindIndex( ) method

This method executes the lambda expression once per each element.

It returns index of the first matching element, if at least one element matches with the given condition; but returns -1, if no element matches with the given condition.



Syntax:

```
int List.FindIndex( Predicate<T> )
```

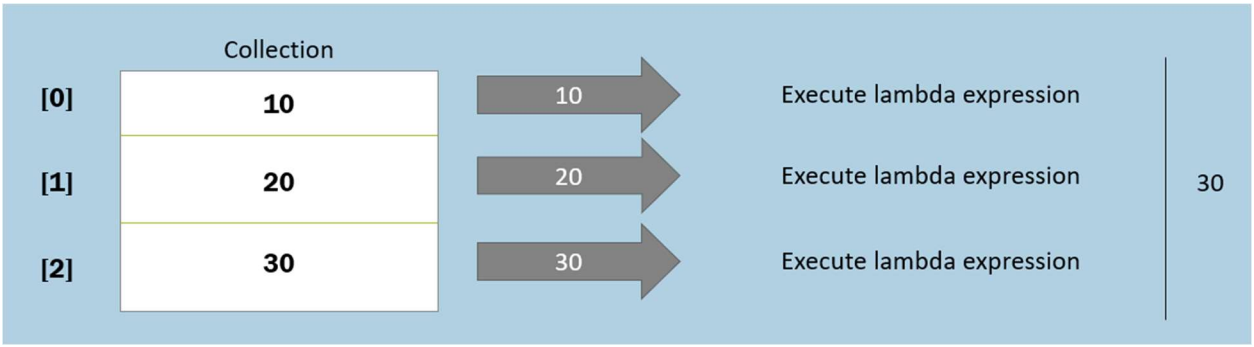
Example:

```
List.FindIndex( n => n > 15 )
```

List - FindLast( ) method

This method executes the lambda expression once per each element.

It returns the last matching element, if at least one element matches with the given condition; but returns the default value, if no element matches with the given condition.



Syntax:

```
T List.FindLast( Predicate<T> )
```

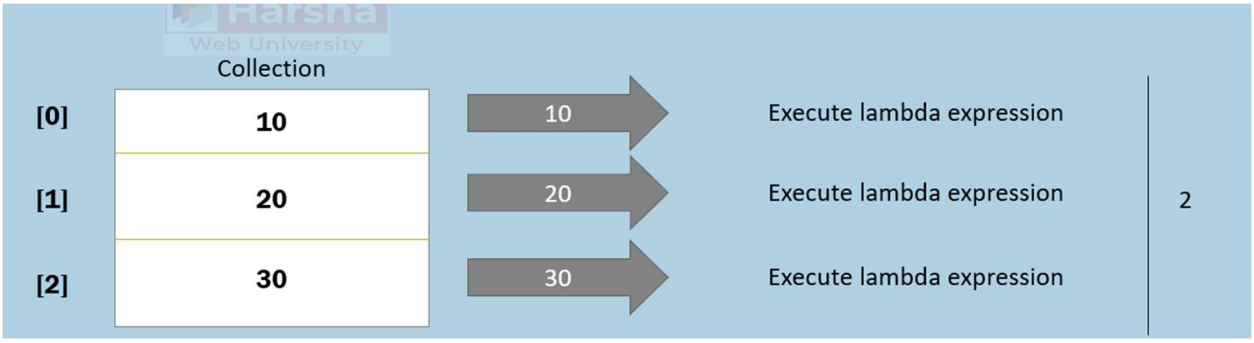
Example:

```
List.FindLast( n => n > 15 )
```

List - FindLastIndex( ) method

This method executes the lambda expression once per each element.

It returns index of the last matching element, if at least one element matches with the given condition; but returns -1, if no element matches with the given condition.



Syntax:

```
int List.FindLastIndex( Predicate<T> )
```

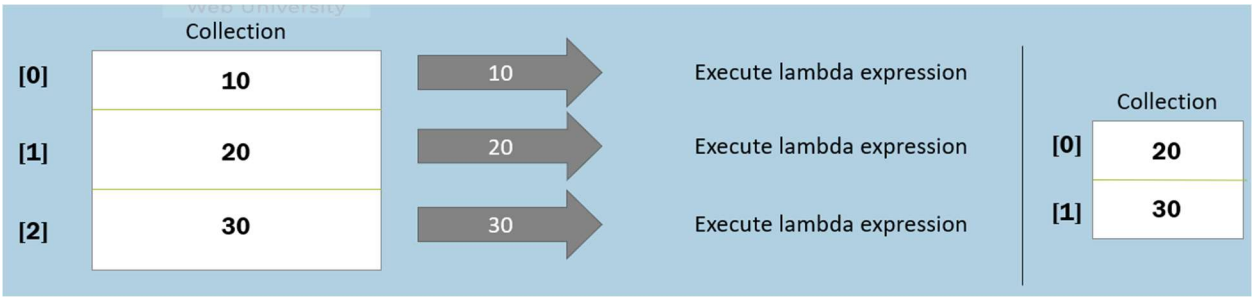
Example:

```
List.FindLastIndex( n => n > 15 )
```

List - FindAll( ) method

This method executes the lambda expression once per each element.

It returns all matching elements as a collection, if there are one or more matching elements; but returns empty collection if no matching elements.



Syntax:

```
List<T> List.FindAll( Predicate<T> )
```

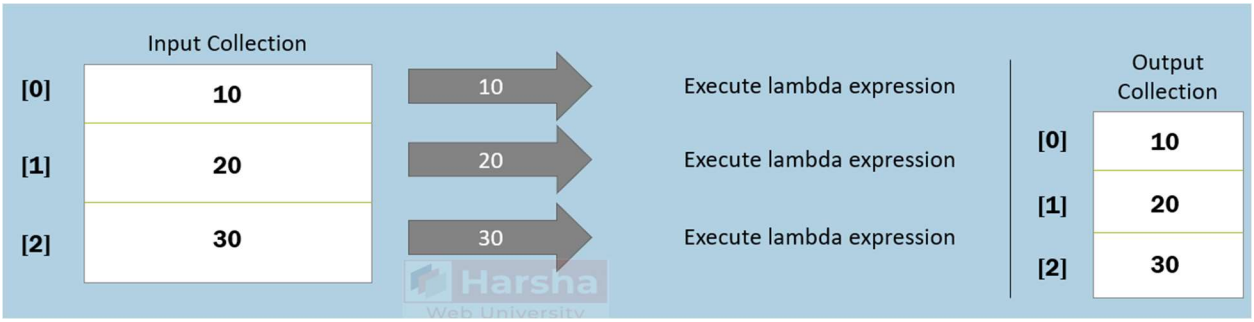
Example:

```
List.FindAll( n => n > 15 )
```

List - ConvertAll( ) method

This method executes the lambda expression once per each element.

It adds each returned element into a new collection and returns the same at last; thus it converts all elements from the input collection as output collection



Syntax:

```
List<TOutput> List.ConvertAll( Converter<TInput, TOutput> )
```

Example:

```
List.ConvertAll( n => Convert.ToDouble(n) )
```

'Dictionary' Collection

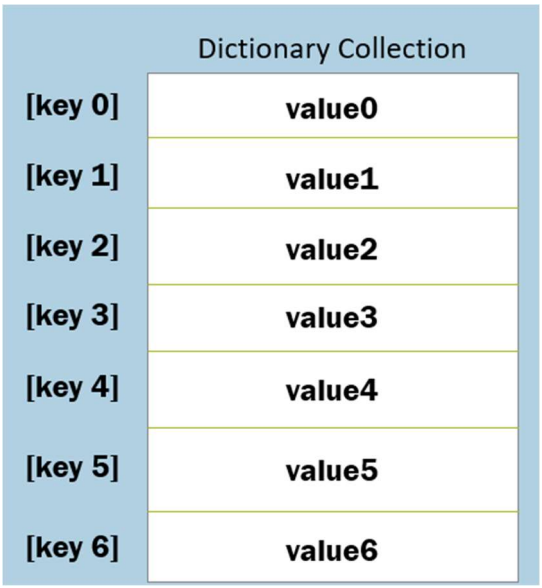
Dictionary collection contains a group of elements of key/value pairs.

Full Path: System.Collections.Generic.Dictionary

The "Dictionary" class is a generic class; so you need to specify data type of the key and data type of the value while creating object.

You can set / get the value based on the key.

The key can't be null or duplicate.




```
Dictionary<TKey, TValue> referenceVariable = new Dictionary<TKey, TValue>( );
```

Features of 'Dictionary' class

- It is dynamically sized. You can add, remove elements (key/value pairs) at any time.
- Key can't be null or duplicate; but value can be null or duplicate.
- It is not index-based. You need to access elements by using key.
- It is not sorted by default. The elements are stored in the same order, how they are initialized.

Properties of 'Dictionary' class

Count : Returns count of elements.

[TKey] : Returns value based on specified key.

Keys : Returns a collection of key (without values).

Values : Returns a collection of values (without keys).

Methods of 'Dictionary' class

void Add(TKey, TValue) : Adds an element (key/value pair).

bool Remove(TKey) : Removes an element based on specified key.

bool ContainsKey(TKey) : Determines whether the specified key exists.

bool ContainsValue(TValue) : Determines whether the specified value exists.

void Clear() : Removes all elements.

'SortedList' Collection

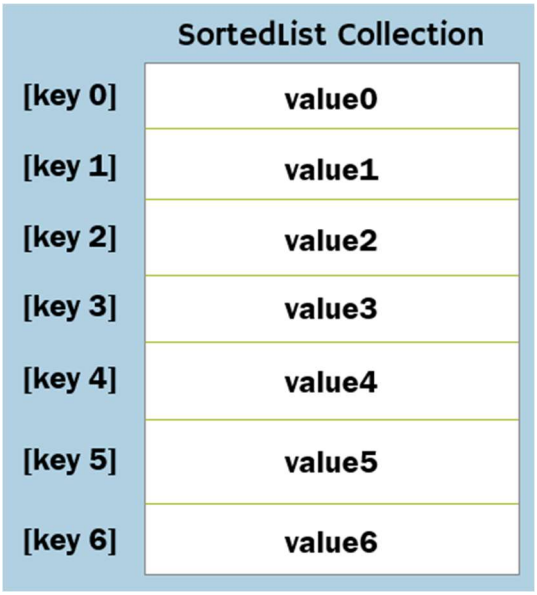
SortedList collection contains a group of elements of key/value pairs.

Full Path: System.Collections.Generic.SortedList

The "SortedList" class is a generic class; so you need to specify data type of the key and data type of the value while creating object.

You can set / get the value based on the key.

The key can't be null or duplicate.




```
SortedList<TKey, TValue> referenceVariable = new SortedList<TKey, TValue>( );
```

Features of 'SortedList' class

- It is dynamically sized. You can add, remove elements (key/value pairs) at any time.
- Key can't be null or duplicate; but value can be null or duplicate.
- It is not index-based. You need to access elements by using key.
- It is sorted by default. The elements are stored in the sorted ascending order, according to the key.
- Each operation of adding element, removing element or any other operation might be slower than Dictionary, because internally it resorts the data based on key.

Properties

Count : Returns count of elements.

[TKey] : Returns value based on specified key.

Keys : Returns a collection of key (without values).

Values : Returns a collection of values (without keys).

Methods

void Add(TKey, TValue) : Adds an element (key/value pair).

bool Remove(TKey) : Removes an element based on specified key.

bool ContainsKey(TKey) : Determines whether the specified key exists.

bool ContainsValue(TValue) : Determines whether the specified value exists.

int IndexOfKey(TKey) : Returns index of the specified key.

int IndexOfValue(TValue) : Returns index of the specified value.

void Clear() : Removes all elements.

'Hashtable' Collection

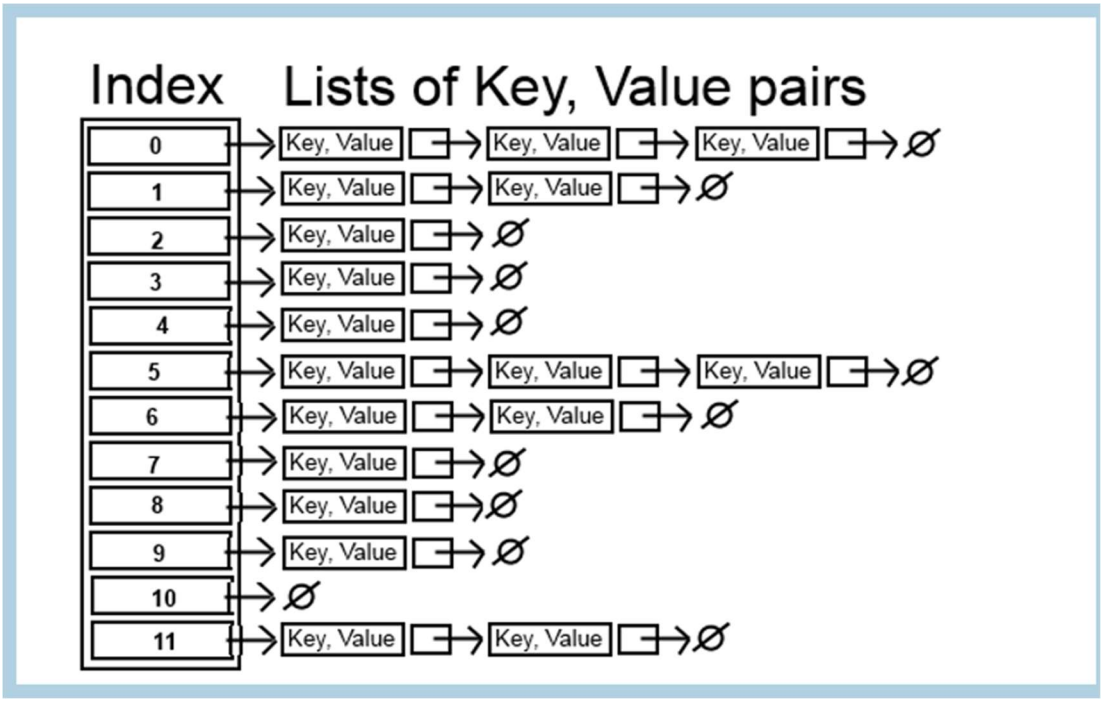
Hashtable collection contains a group of elements of key/value pairs stored at respective indexes.

Full Path: System.Collections.Hashtable

Process of adding an element:

1. Generate index based on the key. Ex: index = hash code % size of hashtable

2. Add the element (key and value) next to the linked list at the generated index.




```
Hashtable referenceVariable = new Hashtable( );
```

- It is dynamically sized. You can add, remove elements (key/value pairs) at any time.
- Key can't be null or duplicate; but value can be null or duplicate.
- The "Hashtable" class is not a generic class.
- You can set / get the value based on the key.
- It is not index-based. You need to access elements by using key.

Properties

Count :Returns count of elements.

[TKey] : Returns value based on specified key.

Keys :Returns a collection of key (without values).

Values : Returns a collection of values (without keys).

Methods

void Add(object key, object value) : Adds an element (key/value pair).

void Remove(object key) : Removes an element based on specified key.

bool ContainsKey(object key) : Determines whether the specified key exists.

bool ContainsValue(object value) : Determines whether the specified value exists.

void Clear() : Removes all elements.

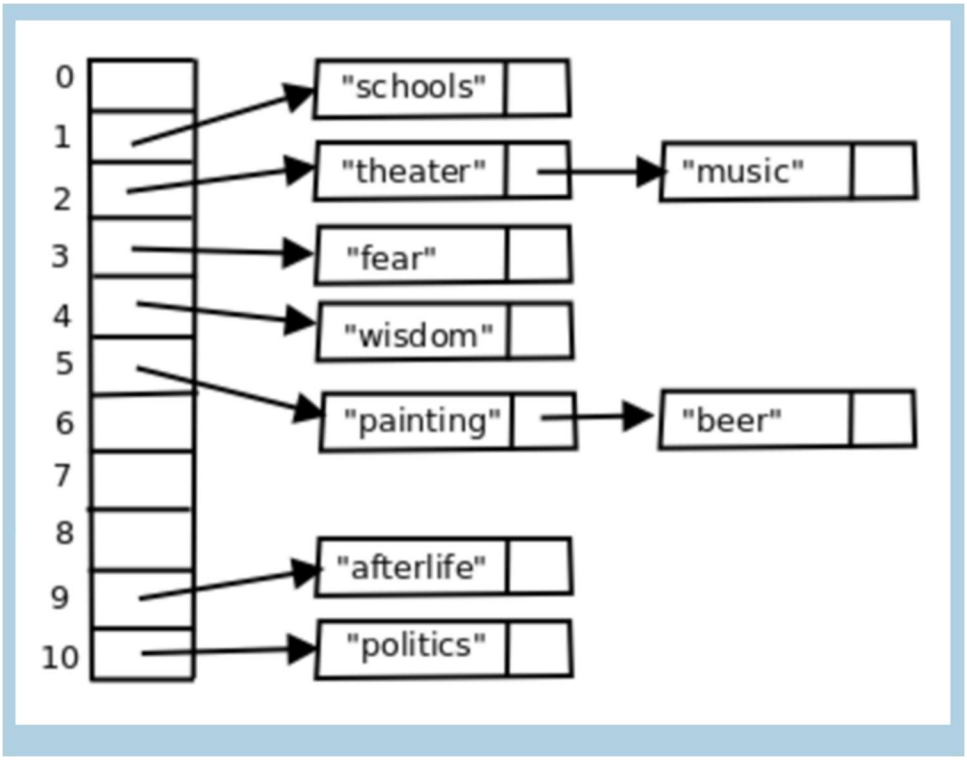
'HashSet' Collection

HashSet collection contains a group of elements of unique values stored at respective indexes.

Full Path: System.Collections.Generic.HashSet

Process of adding an element:

- Generate index based on the value. Ex: index = hash code % count
- Add the element (value) next to the linked list at the generated index.




```
HashSet<T> referenceVariable = new HashSet<T>( );
```

- The "HashSet" class is a generic class.
- You can't set / get the element based on the key / index.
- It searches elements based on the index generated based on the search value.
- HashSet allows only one null value; Hashtable allows only one null key; but allows multiple null values.
- You can't access elements based on key / index. You can use Contains method to search for an element.
- You can't sort elements in HashSet.

- Elements must be unique; duplicate elements are not allowed.

Properties

Count : Returns count of elements.

Methods

void Add(T value) : Adds an element (key/value pair).

void Remove(T value) : Removes an element based on specified key.

void RemoveWhere(Predicate) : Remove elements that matches with condition.

bool Contains (T value) : Determines whether the specified value exists.

void Clear() : Removes all elements.

void UnionWith(IEnumerable<T>) : Unions the hashset and specified collection.

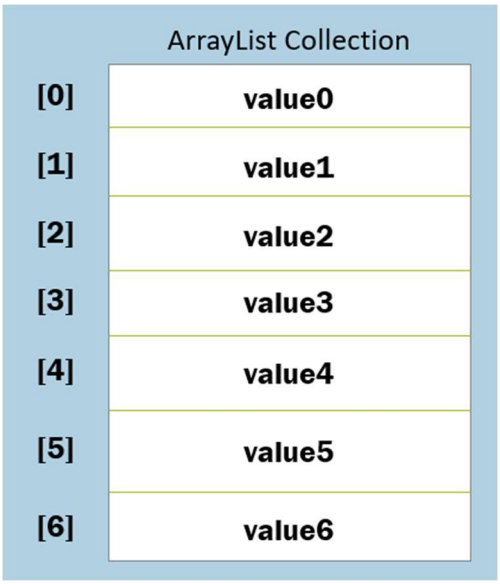
void IntersectWith(IEnumerable<T>) : Intersects the hashset and specified collection.

'ArrayList' Collection

ArrayList collection contains a group of elements of any type.

Full Path: System.Collections.ArrayList

The "ArrayList" class is a not a generic class; so you need not specify data type value while creating object.



```
ArrayList referenceVariable = new ArrayList( );
```

- It is dynamically sized. You can add, remove elements at any time.
- It is index-based. You need to access elements by using the zero-based index.
- It is not sorted by default. The elements are stored in the same order, how they are initialized.
- You don't specify data type of elements for ArrayList. So you can store any type of elements in ArrayList.
- Each element is treated as 'System.Object' type while adding, searching and retrieving elements.

Properties

Count

Capacity

Methods

Add(object) : Clear()

AddRange(ICollection) : IndexOf(object)

Insert(int, object) : BinarySearch(object)

InsertRange(int, ICollection) : Contains(object)

Remove(object) : Sort()

RemoveAt(int) : Reverse()

RemoveRange(int, int) : ToArray()

'Stack' Collection

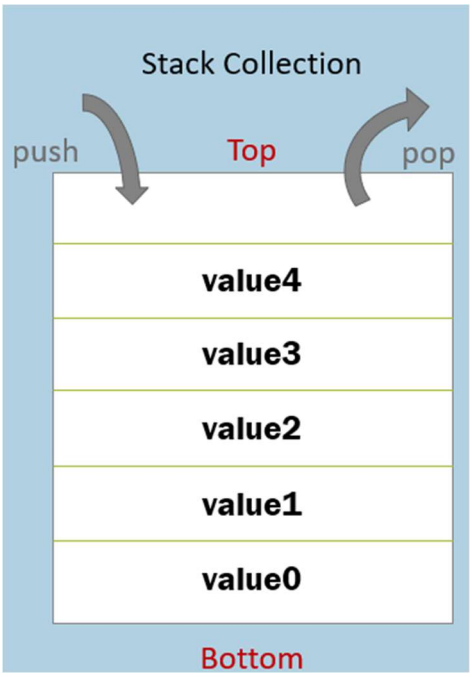
Stack collection contains a group of elements based on LIFO (Last-In-First-Out) based collection.

Full Path: System.Collections.Generic.Stack

It stores the elements in bottom-to-up approach.

Firstly added item at bottom.

Lastly added item at top.




```
Stack<T> referenceVariable = new Stack<T>( );
```

- It is based on LIFO (Last-In-First-Out).
- The "Stack" class is a generic class; so you need to specify data type of elements while creating object.
- You can't access elements based on index.
- You can add elements using 'Push' method; remove elements using 'Pop' method; access last element using 'Peek' method.
- It is not index-based. You need to access elements by either using pop, peek or foreach loop.

Properties

Count : Returns count of elements.

Methods

void Push(T) : dds an element at the top of the stack.

T Pop() : Removes and returns the element at top of stack.

T Peek() : Returns the element at the top of the stack.

bool Contains(T) : Determines whether the specified element exists.

T[] ToArray() : Converts the stack as an array.

void Clear() : Removes all elements.

'Queue' Collection

Queue collection contains a group of elements based on FIFO (First-In-First-Out) based collection.

Full Path: System.Collections.Generic.Queue

It stores the elements in front-to-rear approach.

Firstly added item at front.

Lastly added item at rear.


```
Queue<T> referenceVariable = new Queue<T>( );
```

- It is based on FIFO (First-In-First-Out).
- The "Queue" class is a generic class.
- You can't access elements based on index.
- You can add elements using 'Enqueue' method; remove elements using 'Dequeue' method; access last element using 'Peek' method.
- It is not index-based. You need to access elements by either using Dequeue, Peek or foreach loop.

Properties

Count : Returns count of elements.

Methods

void Enqueue(T) : Adds an element at the top of the queue.

T Dequeue() : Removes and returns the element at top of queue.

T Peek() : Returns the element at the top of the queue.

bool Contains(T) : Determines whether the specified element exists.

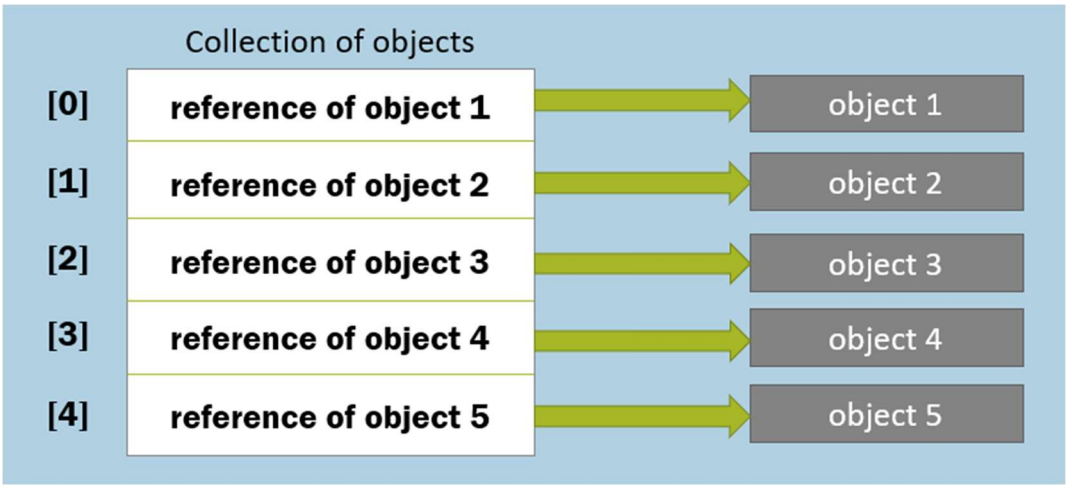
T[] ToArray() : Converts the queue as an array.

void Clear() : Removes all elements.

Collection of Objects

'Collection of objects' is an collection object, where each element stores a reference to some other object.

Used to store details of groups of people or things.

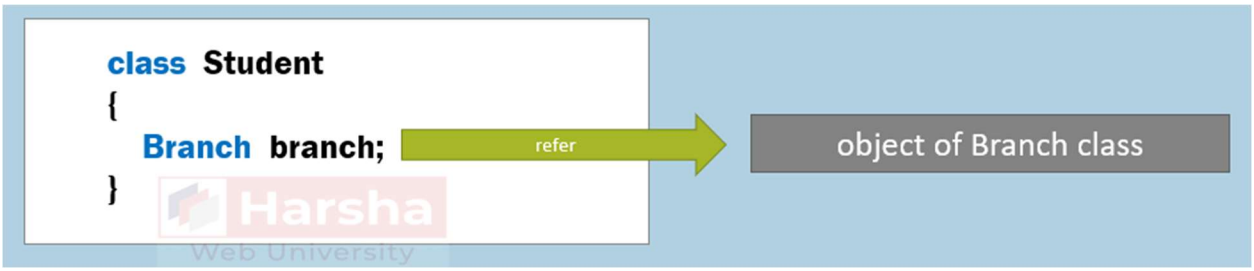



```
1. List<ClassName> referenceVariable = new List<ClassName>( );
2. referenceVariable.Add(object1);
3. referenceVariable.Add(object2);
4. referenceVariable.Add(object3);
5. …
```

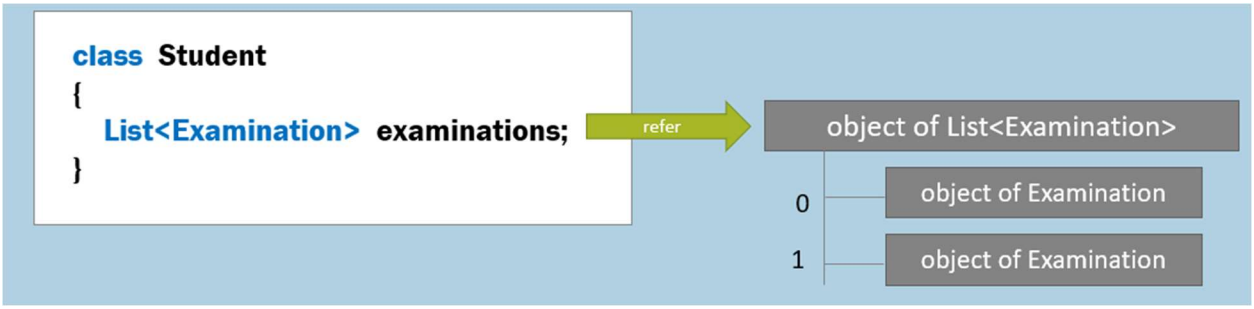
Object Relations

An object can contain a field that stores references to one or more objects.

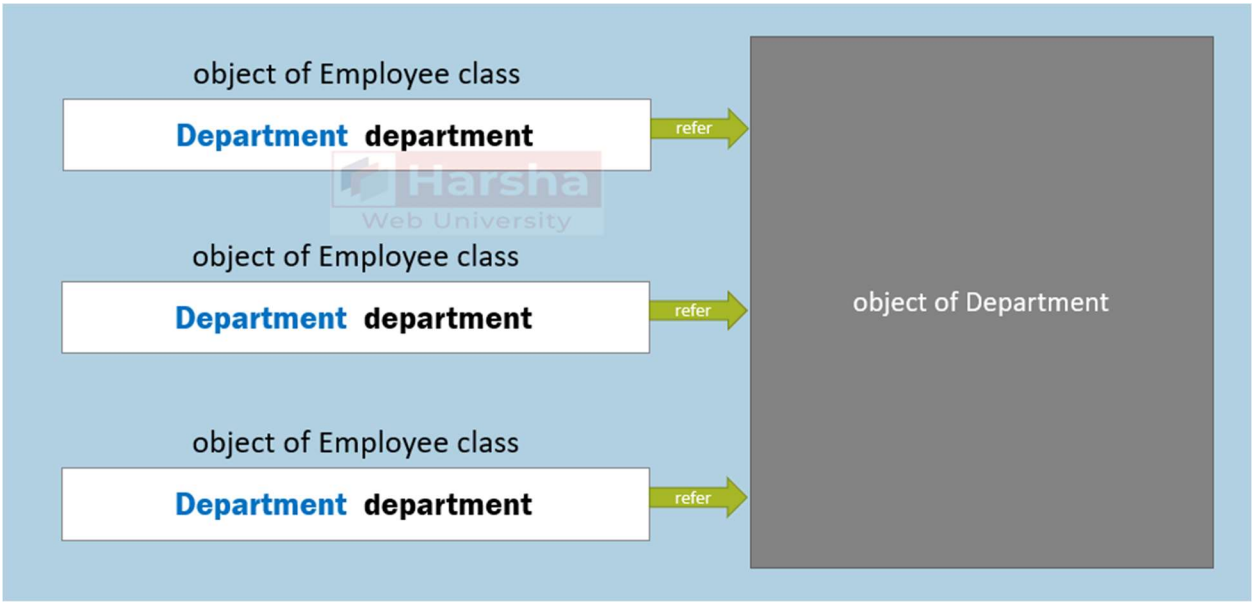
One-to-One Relation



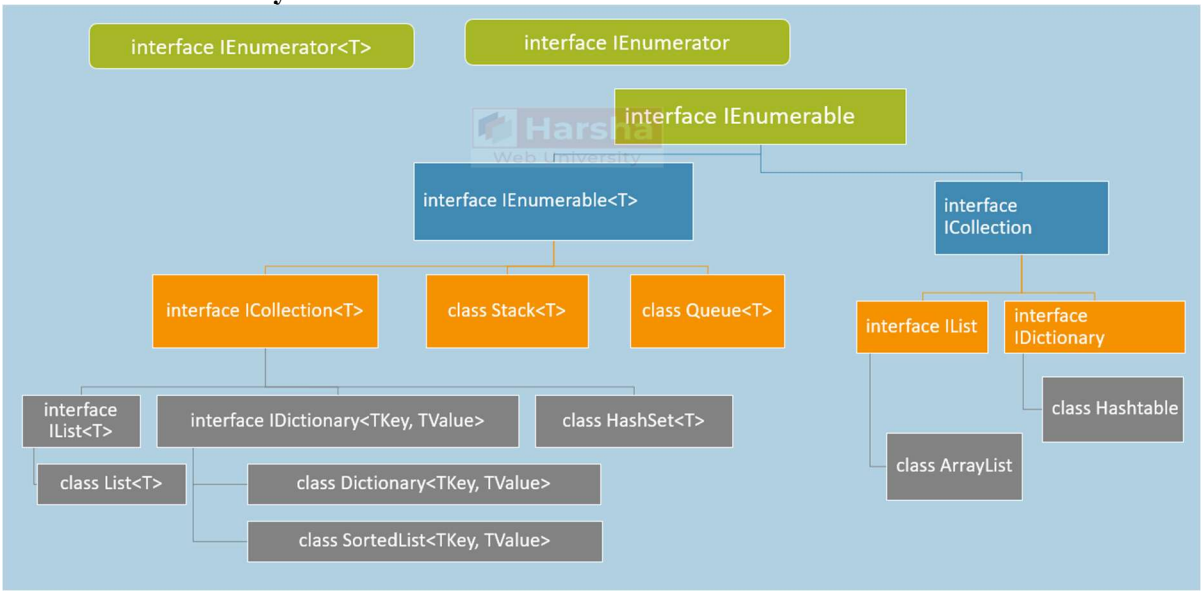
One-to-Many Relation



Many-to-One Relation



Collection Hierarchy:



IEnumerable

The IEnumerable interface represents a group of elements.

It is the parent interface of all types of collections.

It is the parent of ICollection interface, which is implemented by other interfaces such as IList, IDictionary etc.

System.Collections.Generic.IEnumerable<T>

```
1. public interface IEnumerable<out T> : IEnumerable
2. {
3.   IEnumerator<T> GetEnumerator();
4. }
```

System.Collections.IEnumerable

```
1. public interface IEnumerable
2. {
3.   IEnumerator GetEnumerator();
4. }
```

System.Collections.Generic.IEnumerable<T>

```
IEnumerable<T> referenceVariable = new List<T>( );
```

System.Collections.IEnumerable

```
IEnumerable referenceVariable = new ArrayList( );
```

IEnumerator

The IEnumerator interface is meant for readonly and sequential navigation of group of elements.

IEnumerator is used by foreach loop internally.

IEnumerable interface has a method called GetEnumerator that returns an instance of IEnumerator.

IEnumerator by default starts with first element; MoveNext( ) method reads the next element; and the "Current" property returns the current element based on the current position.

System.Collections.Generic.IEnumerator<T>

```
1. public interface IEnumerator<out T> : IDisposable, IEnumerator
2. {
3.   T Current { get; }
4. }
```

System.Collections.IEnumerator

```
1. public interface IEnumerator
2. {
3.   object Current { get; }
4.   bool MoveNext();
5.   void Reset();
6. }
```

System.Collections.Generic.IEnumerator<T>

```
IEnumerator<T> referenceVariable = new List<T>( ).GetEnumerator( );
```

System.Collections.IEnumerator

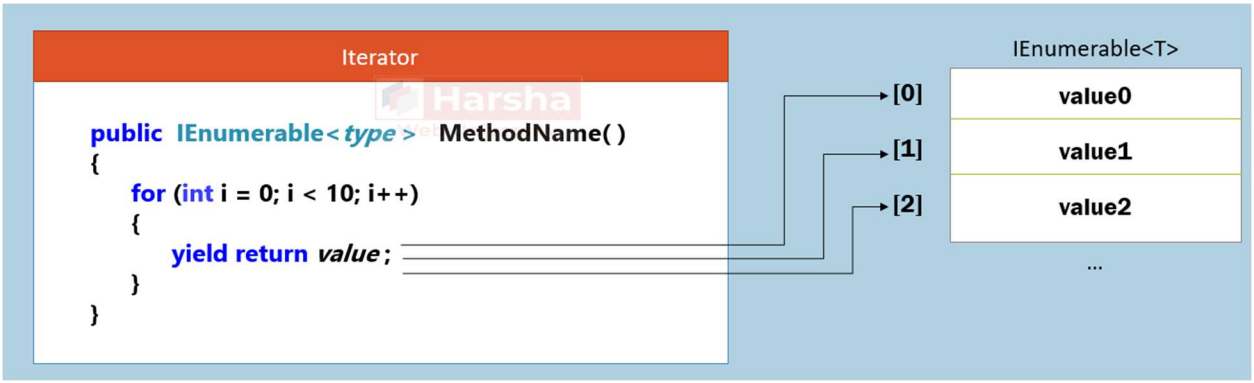
```
1. IEnumerator referenceVariable = new ArrayList( ).GetEnumerator( );
```

Iterators

Iterator is a method which yield returns elements one-by-one.

Iterator is used to iterate over a set of elements.

The *yield return* statement returns an element; but pauses execution of the method.



Iterator can be a method or get accessor of a property.

Iterator can't be constructor or destructor.

Iterator can't contain ref or out parameters.

Multiple *yield* statements can be used within the same iterator.

Iterators are generally implemented in custom collections.

While using the iterator, you must import the System.Collections.Generic namespace.

Custom Collections

Custom collection class makes it easy to store collection of objects of specific class.

It allows you to create additional properties and methods useful to manipulate the collection.

```
1. public class ClassName : IEnumerable
2. {
3. private List<yourClassName> list = new List<yourClassName>( );
4.
5. public IEnumerator GetEnumerator( )
6. {
7. for (int index = 0; index < list.Count; index++)
8. {
9. yield return list[index];
10. }
11. }
12. }
```

It implements IEnumerable interface; and stores a collection as a private field.

It can implement methods such as Add, AddRange, Find etc., and also indexer optionally.

You can also create custom collection by implementing IList<T>; but you need to implement all methods of IList<T> interface in this case.

You can also create custom collection by inheriting from List; it provides all methods of List to your collection class.

IEquatable

The System.IEquatable<T> interface has a method called "Equals", which determines whether the current object and parameter object are logically equal or not, by comparing data of fields.

It can be implemented in the class to make the objects comparable.

It is useful to invoke List.Contains method to check whether the object is present in the collection or not.

interface System.IEquatable<T>

```
1. public interface IEquatable<T>
2. {
3. bool Equals(T other);
4. }
```

Implementation of IEquatable interface

```
1. public class ClassName : IEquatable<ClassName>
2. {
3. public bool Equals(ClassName other)
4. {
5.  return this.field1 == other.field1 && this.field2 == other.field2;
6. }
7. }
```

IComparable

The System.IComparable interface has a method called "CompareTo", which determines order of two objects i.e. current object and parameter object.

It can be implemented in the class to make the objects sortable.

It is useful to invoke List.Sort method to sort the collection of objects.

interface System.IComparable

```
1. public interface IComparable
2. {
3.   int CompareTo(object obj);
4. }
```

Return value

0 : "this" object and parameter object occur in the same position (unchanged).

<0 : "this" object comes first; parameter object comes next.

>0 : parameter object comes first; "this" object comes next.

Implementation of IComparable interface

```
1. public class ClassName : IComparable
2. {
3.   public int CompareTo(object obj)
4.   {
5.    if (this.field1 == other.field1)
6.       return 0; //equal
7.    else if (this.field1 < other.field1)
```

```
8.        return -1; //"this" object comes first.
9.     else
10.       return 1; //parameter object comes first.
11.   }
12. }
```

IComparer

The System.Collections.Generic.IComparer interface has a method called "Compare", which determines order of two objects i.e. current object and parameter object.

It can be implemented by a separate class to make the objects sortable.

It is useful to invoke List.Sort method to sort the collection of objects.

It is an alternative to IComparable; useful for the classes that doesn't implement IComparable.

interface System.Collections.Generic.IComparer

```
1. public interface IComparer<T>
2. {
3.   int Compare(T x, T y);
4. }
```

Return value

0 : both x and y are equal; so will be kept in the same position.

<0 : x comes first; y comes next.

>0 : y comes first; x comes next.

Implementation of IComparer - Example 1:

```
1. public class ClassName : IComparer<T>
2. {
3.   public int Compare(T x, T y)
4.   {
5.     return value;
6.   }
7. }
```

Implementation of IComparer - Example 2:

```
1. public class ClassName : IComparer<T>
2. {
3.   public int Compare(ClassName x, ClassName y)
4.   {
5.      if (x.field1 == y.field1)
6.        return 0; //equal
```

```
7.      else if (x.field1 < y.field1)
8.        return -1; //x object comes next.
9.      else
10.       return 1; //y object comes next.
11.   }
12. }
```

Covariance

Covariance allows you to supply child type, where the parent type is expected.

Implemented with "out" keyword for the generate type parameter of an interface.

In this case, the generic type parameter can be used for return types of methods or properties created in the interface.

```
InterfaceName<ParentType> variable = new ClassName<ChildType>( );
```

Contravariance

Contravariance allows you to supply parent type, where the child type is expected.

Implemented with "in" keyword for the generate type parameter of an interface.

In this case, the generic type parameter can be used for parameter types of methods created in the interface.

```
InterfaceName<ChildType> variable = new ClassName<ParentType>( );
```

Anonymous Types

When you create an object with a set of properties along with values; automatically C# compiler creates a class (with a random name) with specified properties. It is called as 'anonymous type' or 'anonymous classes'.

Useful when you want to quickly create a class that contains a specific set of properties.

Creating Anonymous Object (based on anonymous type)

```
var referenceVariable = new { Property1 = value, Property2 =
value, … };
```

anonymous type [automatically generated]

```
1. class RandomClassName
2. {
3.   public type Property1 { get; set; }
```

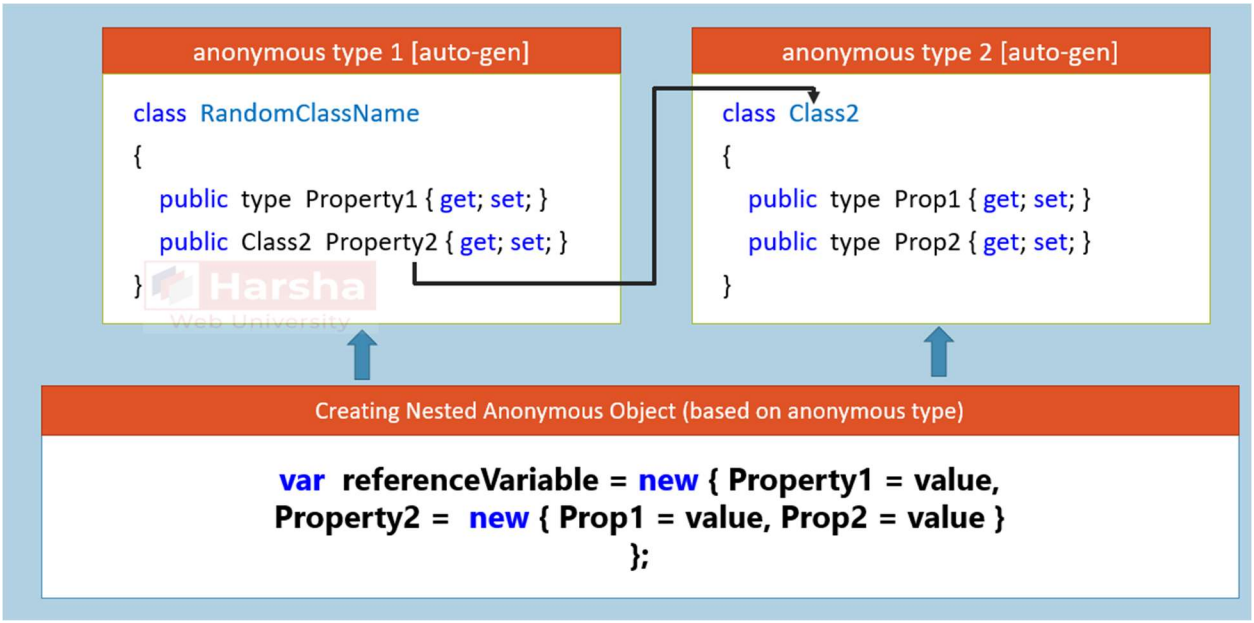
```
4.    public type Property2 { get; set; }
5. }
```

- Anonymous types are created by the C# compiler automatically at compilation time.
- The data types of properties of anonymous types will be automatically taken based on the value assigned into the property.
- Anonymous types are derived from System.Object class directly.
- Anonymous types are by default sealed class.
- Properties of anonymous types are by default readonly properties.
- Anonymous types can't contain other members such as normal properties, events, methods, fields etc.
- Properties of anonymous types will be always 'public' and 'readonly'.
- You can't add additional members of anonymous types once after compiler creates it automatically.
- 'null' can't be assigned into property of anonymous type.
- The data type of anonymous objects are always given as "var".
- Anonymous types can't casted to any other type, except into System.Object type.
- You can't create a field, property, event or return type of a method, parameter type as of anonymous type.
- It is recommended to use the anonymous objects within the same method, in which they are created. You can pass anonymous type object to method as parameter as 'System.Object' type; but it's not recommended.

Nested Anonymous Types

You can nest an anonymous object into another.

Then two anonymous types will be created.



Anonymous Arrays

You can create 'array of anonymous objects' or 'implicitly typed array' with group of anonymous objects.

All objects must contain same set of properties.

Creating anonymous array / Implicitly typed array:

```
var referenceVariable = new []
{
    new { Property1 = value, Property2 = value2, ... },
    new { Property1 = value, Property2 = value2, ... },
    ...
};
```

anonymous type [automatically generated]

```
class RandomClassName
{
  public type Property1 { get; set; }
  public type Property2 { get; set; }
}
```

Tuples

The System.Tuple class represents a set of values of any data type.

Introduced in C# 4.0.

Useful to return multiple values from a method (or) to pass multiple values to a method.

Represents a set of values quickly without creating a separate class.

Alternative to anonymous objects (to be used as parameter types / return types).

Step 1: Object of Tuple class

```
var referenceVariable = new Tuple<type1, type2, …>( ) { value1, value2, … };
```

Step 2: Accessing Elements

```
1. referenceVariable.Item1 //returns value1
2. referenceVariable.Item2 //returns value2
```

Tuple

```
1. Item1 = value1
2. Item2 = value2
```

Tuple stores only a set of values (of any data type); but doesn't store property names. So you should access them as Item1, Item2 etc.; which doesn't make sense some times.

Tuple supports up to 8 elements only by default. You can store more than 8 values by using nested tuples (tuple inside tuple).

Tuples are mainly used to pass multiple values to a method as parameter; and also return multiple values from a method.

Value Tuples

'Value Tuples' are advancement to 'Tuple' class with simplified syntax.

Introduced in C# 7.1.

Supports unlimited values.

You will access elements with real field names; instead of Item1, Item2 etc.

Can be used as method parameters / return value; much like Tuple class.

Step 1: Creating Value Tuple

```
(type fieldName1, type fieldName2, …) referenceVariable =
(value1, value2, … );
```

Step 2: Accessing Elements

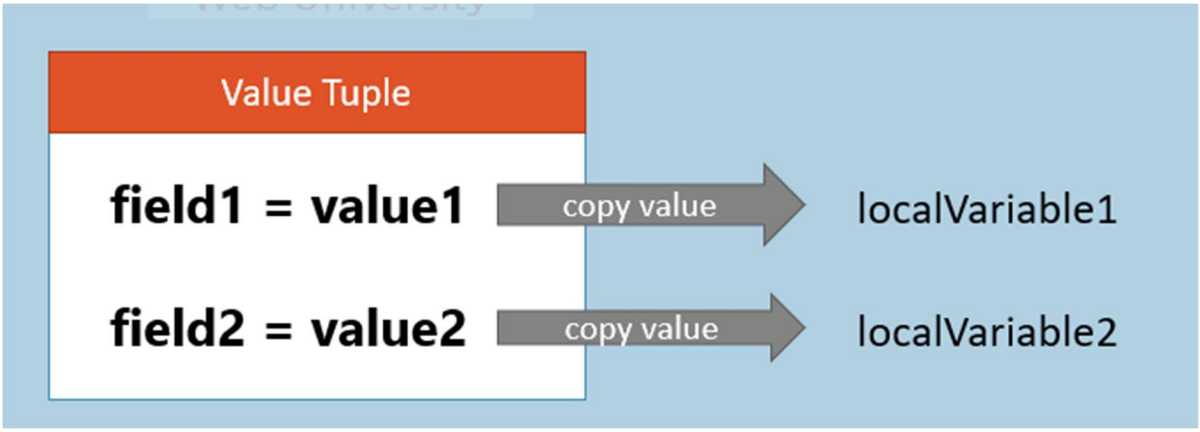
```
1. referenceVariable.fieldName1 //returns value1
2. referenceVariable.fieldName2 //returns value2
3. …
```

Value Tuple

```
1. field1 = value1
2. field2 = value2
```

Deconstruction

Deconstruction allows you to assign elements of value tuple into individual local variables.



Step 1: Create Value Tuple

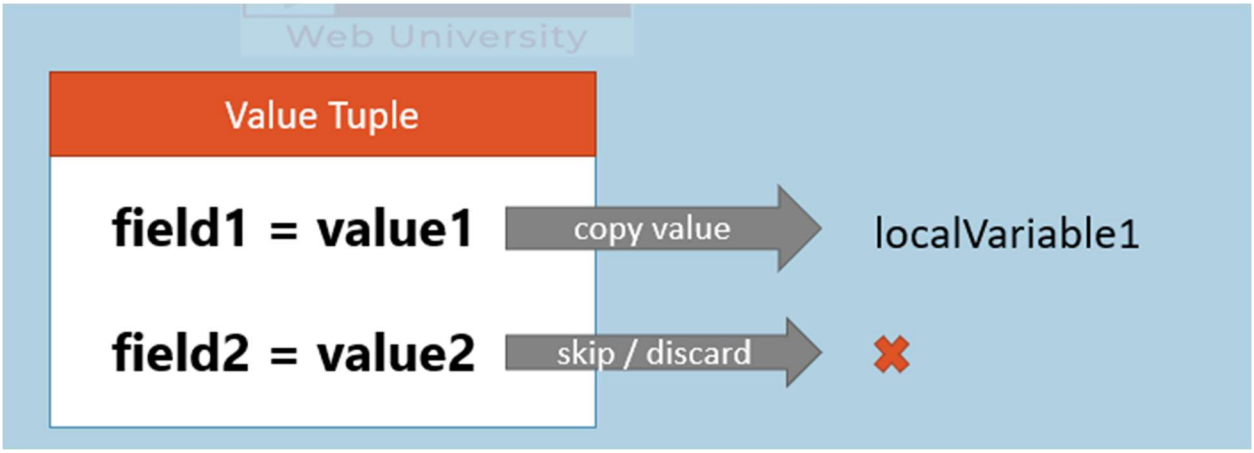
```
(type fieldName1, type fieldName2, …) referenceVariable =
(value1, value2, … );
```

Step 2: Deconstruction

```
(type variableName1, type variableName2, …) = referenceVariable;
```

Discards

Discard allows you to skip a value which you don't require, by using underscore (_).



Step 1: Create Value Tuple

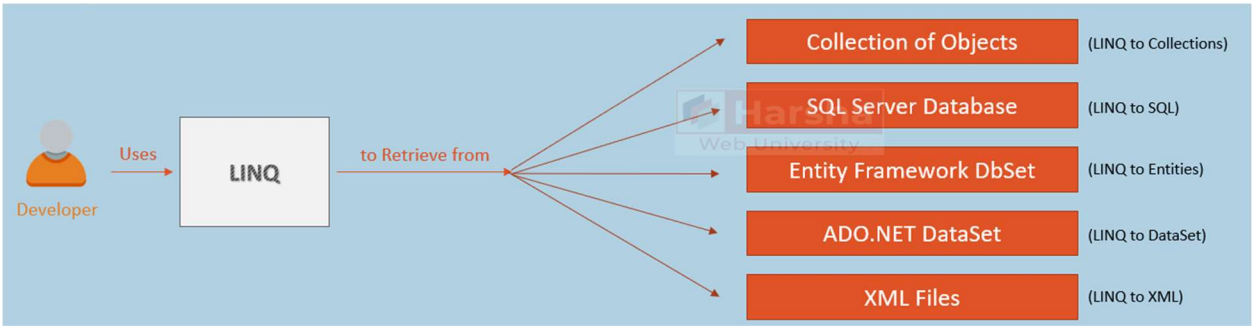
```
(type fieldName1, type fieldName2) referenceVariable = (value1, value2);
```

Step 2: Deconstruction with Discard

```
(type variableName1, _ ) = referenceVariable;
```

LINQ

LINQ is a 'uniform query syntax' that allows you to retrieve data from various data sources such as arrays, collections, databases, XML files.



LINQ Query - Example

```
1. var result = Customers.Where(temp => temp.Location == "New York").ToList( );
2. //returns a list of customers from New York location.
```

Advantages of LINQ

Single Syntax - To Query Multiple Data Sources

Developer uses the same LINQ syntax to retrieve information from various data sources such as collections, SQL Server database, Entity Framework DbSet's, ADO.NET DataSet etc.

Compile-Time Checking of Query Errors

Errors in the LINQ query will be identified while compilation time / while writing the code in Visual Studio.

IntelliSence Support

The list of properties of types are shown in VS IntelliSence while writing the LINQ queries.

LINQ Extension Methods

Filtering: Where, OfType

Sorting: OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse

Grouping: GroupBy

Join: Join

Project: Select, SelectMany

Aggregation: Average, Count, Max, Min, Sum

Quantifiers: All, Any, Contains

Elements: ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault

Set Operations: Distinct, Except, Intersect, Union

Partitioning: Skip, SkipWhile, Take, TakeWhile

Concatenation: Concat

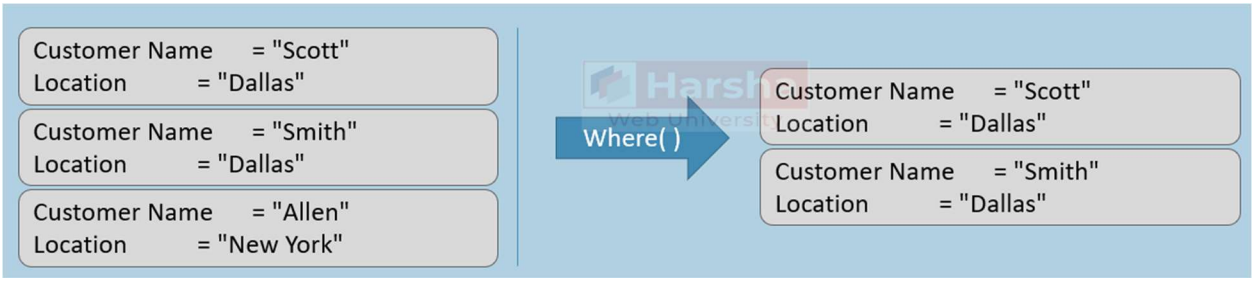
Equality: SequenceEqual

Generation: DefaultEmpty, Empty, Range, Repeat

Conversion: AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList

Where

Where() method filters collection based on given lambda expression and returns a new collection with matching element.



Where Extension Method - Declaration

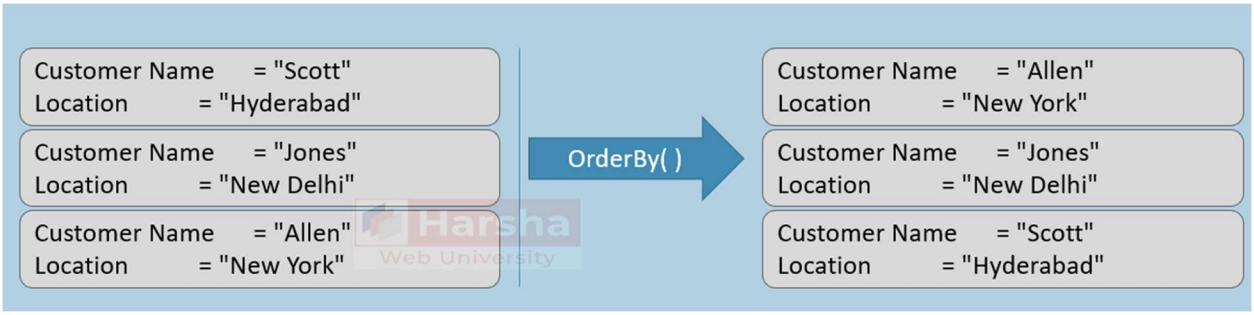
```
Where(Func<TSource, bool> predicate)
```

Where Extension Method - Usage

```
1. var result = Customers.Where(temp => temp.Location == "Dallas").ToList( );
2. //returns a list of customers from Hyderabad location.
```

OrderBy

OrderBy() method sorts collection based on given lambda expression (property) and returns a new collection with sorted elements.



OrderBy Extension Method - Declaration

```
1. OrderBy(Func<TSource, TKey> keySelector)
```

OrderBy Extension Method - Usage

```
1. var result = Customers.OrderBy(temp => temp.CustomerName).ToList( );
2. //returns a list of customers sorted based on customer name.
```

OrderByDescending Extension Method - Declaration

```
1. OrderByDescending(Func<TSource, TKey> keySelector)
```

OrderByDescending Extension Method - Usage

```
1. var result = Customers.OrderByDescending(temp => temp.CustomerName).ToList( );
2. //returns a list of customers sorted based on customer name in descending
   order.
```

ThenBy Extension Method - Declaration

```
1. ThenBy(Func<TSource, TKey> keySelector)
```

ThenBy Extension Method - Usage

```
1. var result = Customers.OrderBy(temp => temp.Location)
2. .ThenBy(temp => temp.CustomerName).ToList( );
3. //returns a list of customers sorted based on location and customer name.
```

ThenByDescending Extension Method - Declaration

```
ThenByDescending(Func<TSource, TKey> keySelector)
```

ThenByDescending Extension Method - Usage

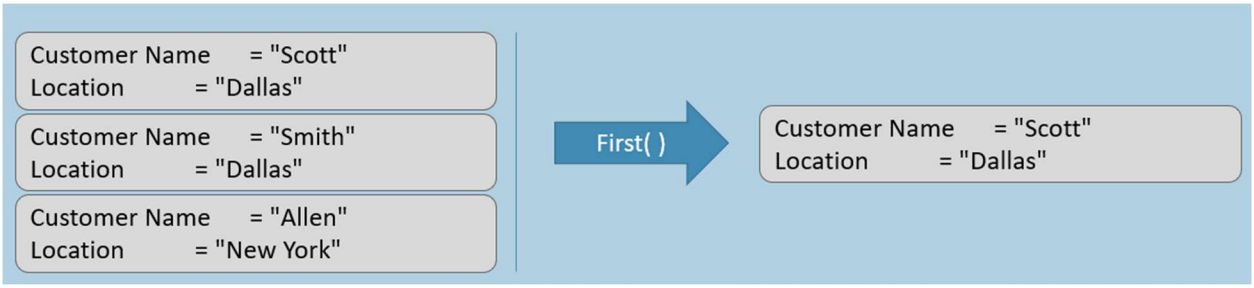
```
1. var result = Customers.OrderBy(temp => temp.Location)
```

```
2. .ThenByDescending(temp => temp.CustomerName).ToList( );
3. //returns a list of customers sorted based on location (ascending) and
   customer name (descending).
```

First

First() method returns first element in the collection that matches with the collection.

It throws exception if no element matches with the condition.



First Extension Method - Declaration

```
1. First(Func<TSource, bool> predicate)
```

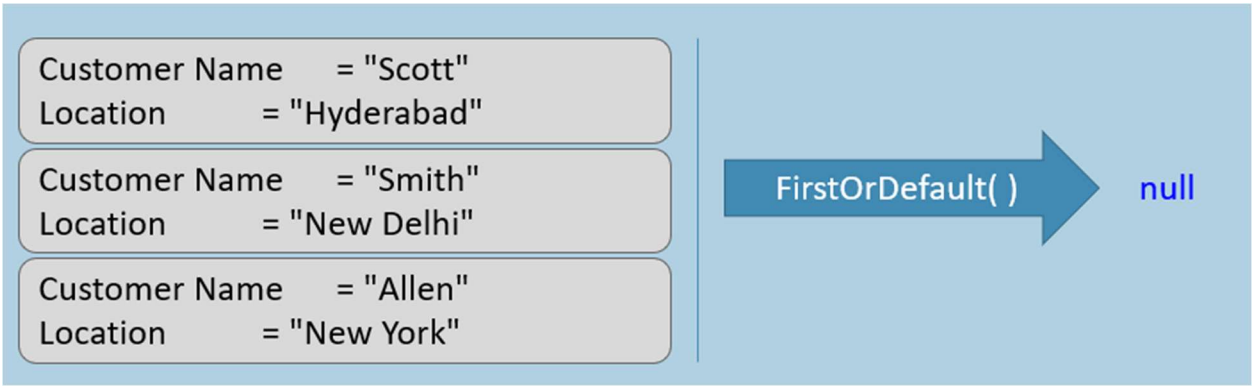
First Extension Method - Usage

```
1. var result = Customers.First(temp => temp.Location == "Dallas");
2. //returns the first customer from Dallas location.
```

FirstOrDefault

FirstOrDefault() method returns first element that matches with the condition.

It returns null if no element matches with the condition.



FirstOrDefault Extension Method - Declaration

```
FirstOrDefault(Func<TSource, bool> predicate)
```

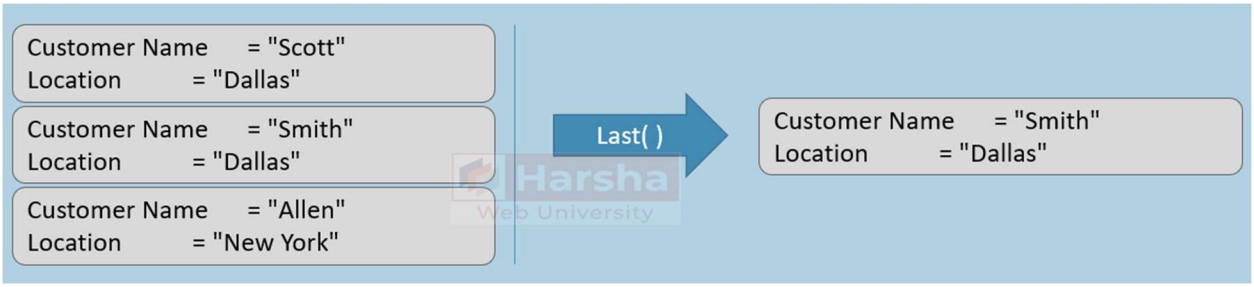
FirstOrDefault Extension Method - Usage

```
1. var result = Customers.FirstOrDefault(temp => temp.Location == "London");
2. //returns the first customer from London location (or) returns null if not
   exists.
```

Last

Last() method returns last element in the collection that matches with the collection.

It throws exception if no element matches with the condition.



Last Extension Method - Declaration

```
Last(Func<TSource, bool> predicate)
```

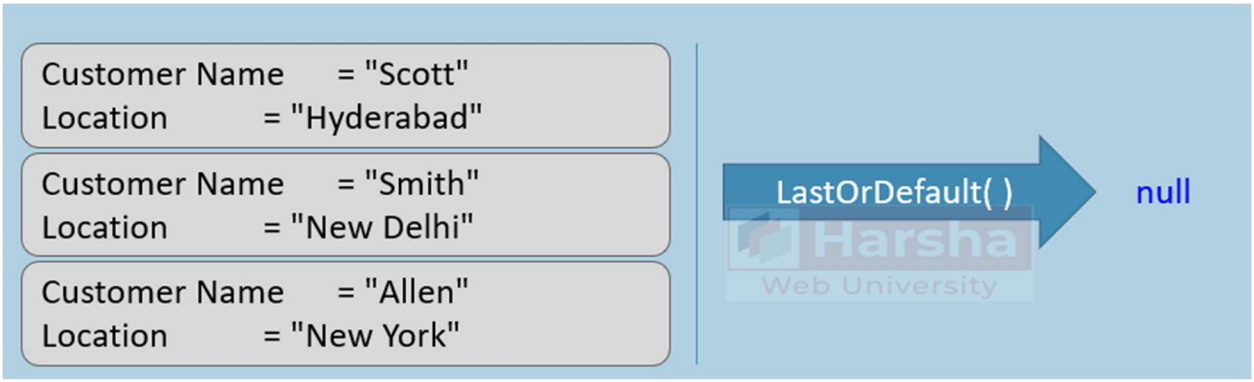
Last Extension Method - Usage

```
1. var result = Customers.Last(temp => temp.Location == "Dallas");
2. //returns the last customer from Dallas location.
```

LastOrDefault

LastOrDefault() method returns last element that matches with the condition.

It returns null if no element matches with the condition.



LastOrDefault Extension Method - Declaration

```
LastOrDefault(Func<TSource, bool> predicate)
```

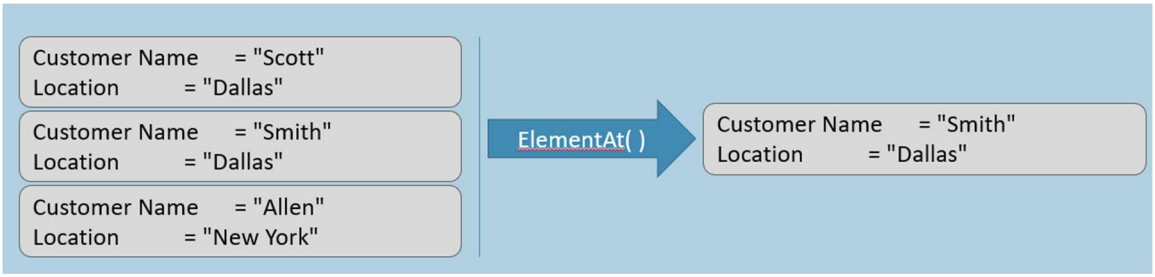
LastOrDefault Extension Method - Usage

```
1. var result = Customers.LastOrDefault(temp => temp.Location == "London");
2. //returns the last customer from London location (or) returns null if not
   exists.
```

ElementAt

Element() method returns an element in the collection at specified index.

It throws exception if no element exists at the specified index; to get 'null' instead, use ElementOrDefault().



ElementAt Extension Method - Declaration

```
ElementAt(int index)
```

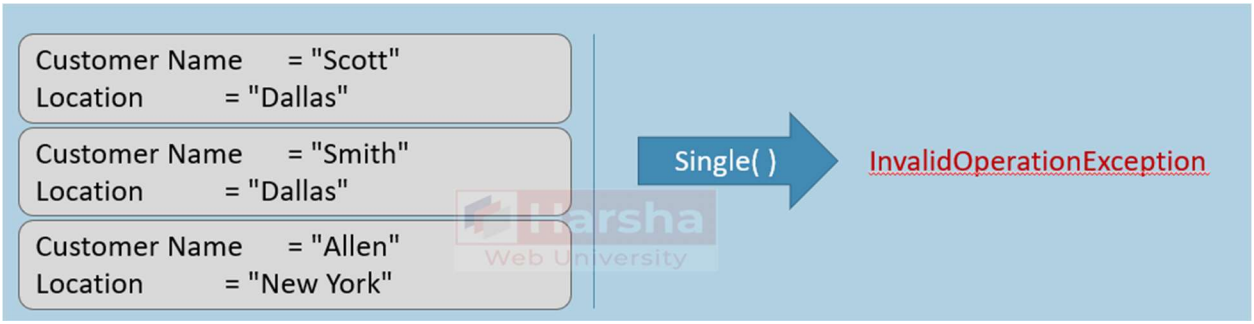
ElementAt Extension Method - Usage

```
var result = Customers.ElementAt(1); //returns the customer at index 1
```

Single

It returns first element (only one element) that matches with the collection.

It throws exception if no element or multiple elements match with the condition.



Single Extension Method - Declaration

```
Single(Func<TSource, bool> predicate)
```

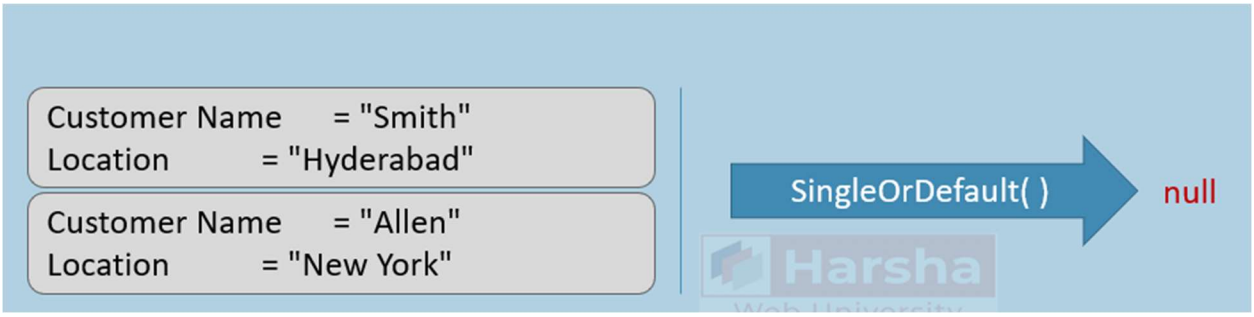
Single Extension Method - Usage

```
1. var result = Customers.Single(temp => temp.Location == "Dallas");
2. //returns the first (only one customer) from Dallas location.
3. but it throws exception if none / multiple elements matches with the
   condition.
```

SingleOrDefault

It returns first element (only one element) that matches with the collection.

It returns null if no element matches with the condition; but it throws exception if multiple elements match with the condition.



SingleOrDefault Extension Method - Declaration

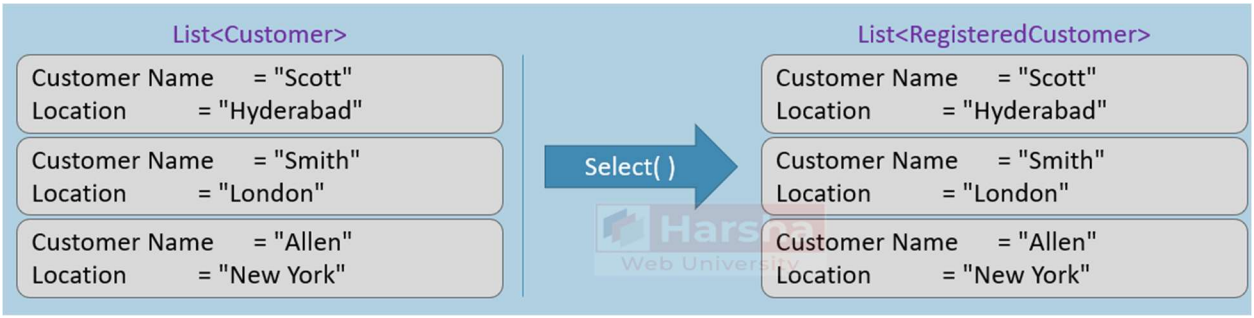
```
SingleOrDefault(Func<TSource, bool> predicate)
```

SingleOrDefault Extension Method - Usage

```
1. var result = Customers.SingleOrDefault(temp => temp.Location == "London");
2. //returns the first (only one customer) from London location.
3. it throws exception if multiple elements matches with the condition; but null
   in case of no match.
```

Select

It returns collection by converting each element into another type, based on the conversion expression.



Select Extension Method - Declaration

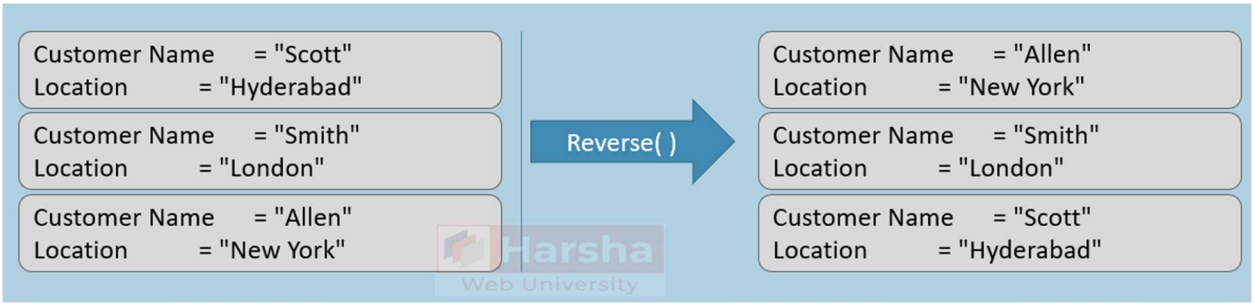
```
Select(Func<TSource, TResult> selector)
```

Select Extension Method - Usage

```
1. var result = Customers.Select(temp => new RegisteredCustomer()
2. { CustomerName = temp.CustomerName, Location = temp.Location } );
3. //converts all customers into a collection of RegisteredCustomer class.
```

Reverse

It reverses the collection.



Reverse Extension Method - Declaration

```
Reverse( )
```

Reverse Extension Method - Usage

```
var result = Customers.Reverse( ); //reverses the customers collection
```

Min, Max, Count, Sum, Average

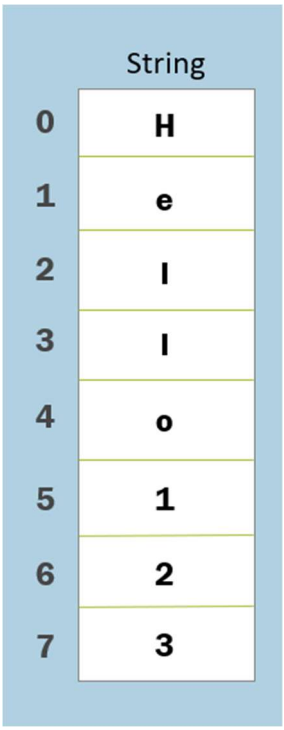
It performs aggregate operations such as finding minimum value of specific property of all elements of a collection.

Min, Max, Count, Sum, Average - Example

```
1. var result1 = Students.Min( temp => temp.Marks ); //minimum value of Marks property
2. var result2 = Students.Max( temp => temp.Marks ); //maximum value of Marks property
3. var result3 = Students.Count( ); //count of elements
4. var result4 = Students.Sum( temp => temp.Marks); //sum value of Marks property
5. var result5 = Students.Average( temp => temp.Marks); //average value of Marks property
```

String

- The System.String is a class that represents an array of Unicode characters.
- String stores a set of characters as char[].
- Max no. of characters in the string: 2 billion.
- String is immutable. That means, it can't be modified.




```
string referenceVariable = "Hello123";
```

Methods of 'String'

1. string ToUpper( )

Returns the same string in upper case.

2. string ToLower( )

Returns the same string lower case.

3. string Substring(int startIndex, int length)

Returns a string with a set of characters starting from the "startIndex" up to the "length" no. of characters.

The "length" parameter is optional.

4. string Replace(string oldString, string newString)

Returns a string after replacing all occurrences of oldString with newString.

5. string[] Split(char separator)

Returns a string[] after splitting the current string into an array of characters.

At each occurrence of separator character, a new string is formed-up.

At last, all the pieces of strings are converted as a string[].

6. string Trim()

Returns the same string after removing extra spaces at beginning and ending of the current string.

It returns the same string if no spaces found at L.H.S. and R.H.S. of the current string.

7. string ToCharArray()

Returns the same string as an array of characters (char[]).

8. static string Join(string separator, IEnumerable<T> values)

Returns the a string with joined values with separator in between each value.

9. bool Equals(string otherString)

Returns a Boolean value that indicates whether the current string and the specified otherString are equal or not (each character will be compared)

10. bool StartsWith(string otherString)

Returns a Boolean value that indicates whether the current string beings with the specified otherString.

11. bool EndsWith(string otherString)

Returns a Boolean value that indicates whether the current string ends with the specified otherString.

12. bool Contains(string otherString)

Returns a Boolean value that indicates whether the current string contains the specified otherString anywhere.

13. int IndexOf(string otherString, int startIndex)

Returns index of the first character of the specified otherString, where the otherString exists in the current string. Searching process starts from the specified startIndex. The startIndex is optional.

In case if the specified otherString doesn't exist in the current string, the method returns -1.

14. int LastIndexOf(string otherString, int startIndex)

It is same as IndexOf(); but searching process takes place from Right-To-Left direction, staring from the startIndex.

15. static bool IsNullOrEmpty(string value)

It determines whether the given string value is null or empty ("").

16. static bool IsNullOrWhiteSpace(string value)

It determines whether the given string value is null or white space (" ").

17. static string Format(string format, object arg0, object arg1, …)

Returns the string after substituting the specified argument values at specified placeholders.

A place holder is represented as {indexOfArgument}.

Eg: string.Format("{0} Oriented {1}", "Object", "Programming") //Object Oriented Programming

18.string Insert(int startIndex, string value)

Returns a new string object which inserts the new string value at the specified index in the existing string object.

19.string Remove(int startIndex, int count)

It returns a new string object after removing specified count of characters at the specified start index, from the current string object.

Constructors of 'String'

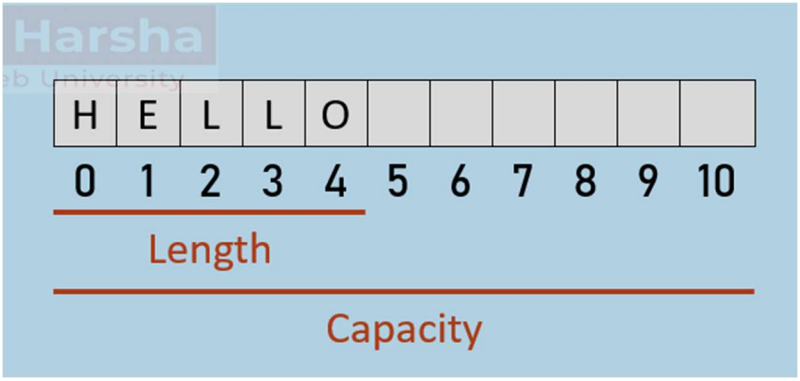
1. String(char[])

It initializes the string with specified char[].

StringBuilder

The System.Text.StringBuilder is a class that represents an appendable string.

The string value stored in StringBuilder can be modified, without recreating StringBuilder object.




```
1. StringBuilder sb = new StringBuilder("initial string", capacity);
2. sb.Append("additional string")
```

The 'Capacity' property of String Builder represents the number of characters that can be stored in the string builder, as per current memory allocation.

When the programmer tries to store much more number of characters than the Capacity, String Builder automatically increases the 'Capacity' property to its double.

The default value of 'Capacity' is 16; the programmer can assign its value via constructor or set accessor of the 'Capacity' property.

Constructors of 'StringBuilder'

1. StringBuilder( )

It initializes the StringBuilder type of object with the default capacity (16).

2. StringBuilder(int capacity)

It initializes the StringBuilder type of object with the specified capacity.

3. StringBuilder(string value)

It initializes the StringBuilder type of object with the specified value.

4. StringBuilder(string value, int capacity)

It initializes the StringBuilder type of object with the specified value and capacity.

Properties of 'StringBuilder'

1. int Length { get; set; }

This property gets / sets the length (number of characters) in the string builder.

2. char [int index] { get; set; }

This indexer gets / sets the single character at the specified character index.

3. int Capacity { get; set; }

This property returns the number of characters that can be stored in the current string builder (currently memory allocated). Default is 16.

4. int MaxCapacity { get; }

This property represents maximum number of characters up to which, the Capacity can be extended.

Default is int.MaxValue.

Methods of 'StringBuilder'

1. string Append(string value)

Adds the given string value at the end of current value of string builder.

2. string Insert(int startIndex, string value)

Returns a new string object which inserts the new string value at the specified index in the existing string object.

3. string Remove(int startIndex, int count)

It returns a new string object after removing specified count of characters at the specified start index, from the current string object.

4. string Replace(string oldString, string newString)

Returns a string after replacing all occurrences of oldString with newString.

5. string ToString()

Returns the current value of string builder as a string object.

When to use 'String'

- When you would like to make less number of changes to the string.
- When you perform limited number of concatenation operations.
- When you want to perform extensive search operations using methods such as IndexOf, Contains, StartsWith etc.

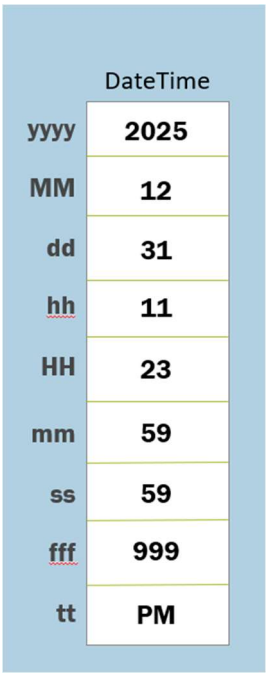
When to use 'String Builder'

- When you would like to unknown number of / extensive number of changes to a string (Eg: making changes / concatenations to string through a loop).
- When you perform less number of search operations on strings.

DateTime

The System.DateTime is a structure that represents date and time value.

Default format: `yyyy-MM-dd hh:mm:ss.fff tt`



	DateTime
yyyy	2025
MM	12
dd	31
hh	11
HH	23
mm	59
ss	59
fff	999
tt	PM

Creating DateTime using 'Parse' method

```
DateTime.Parse("2025-12-31 11:59:59.999 PM")
```

Creating DateTime using 'ToDateTime' method

```
Convert.ToDateTime("2025-12-31 11:59:59.999 PM")
```

'DateTime' Constructors

1. DateTime(int year, int month, int day, int hour, int minute, int second, int millisecond)

It creates a new instance (structure instance) with the specified date and time values.

"hour" should specified 24-hour format (0 to 23).

DateTime Properties

1. int Day { get; }

It returns the day (1 to 31) of current date & time value.

2. int Month { get; }

It returns the month (1 to 12) of current date & time value

2. int Year { get; }

It returns the year (1 to 9999) of current date & time value.

3. int Hour { get; }

It returns the hour (0 to 23) of current date & time value.

4. int Minute { get; }

It returns the minute (0 to 59) of current date & time value.

5. int Second { get; }

It returns the second (0 to 59) of current date & time value.

6. int Millisecond { get; }

It returns the millisecond (0 to 999) of current date & time value.

7. int DayOfYear { get; }

It returns the day of year (1 to 366) based on current date & time value.

8. DayOfWeek DayOfWeek { get; }

It returns the day of week - Sunday to Saturday (0 to 6) based on current date & time value.

9. static DateTime Now { get; }

It returns an instance of DateTime structure that represents the current system date & time.

DateTime Methods

1. string ToString()

It returns the date & time value in the default date format, based on current Windows settings.

2. string ToString(string format)

It returns the date & time value in the specified format.

3. string ToShortDateString()

It returns the date value in the default short date format, based on current Windows settings.

Eg: MM/dd/yyyy | 12/31/2030

4. string ToLongDateString()

It returns the date value in the default long date format, based on current Windows settings.

Eg: dd MMMM yyyy | 31 December 2030

5. string ToShortTimeString()

It returns the date value in the default short time format, based on current Windows settings.

Eg: hh:mm tt | 11:59 pm

6. string ToLongTimeString()

It returns the date value in the default long time format, based on current Windows settings.

Eg: hh:mm:ss tt | 11:59:59 pm

7. static int DaysInMonth(int year, int month)

It returns number of days in the specified month in the specified year.

8. static DateTime Parse(string value)

It creates and returns a new instance of DateTime structure, based on the given date string.

It uses the system default date format or "yyyy-MM-dd hh:mm:ss.fff tt" format.

Eg: "2030-12-31 11:59:59.999 pm"

9. static DateTime ParseExact(string value, string format, IFormatProvider provider, DateTimeStyle style)

It creates and returns a new instance of DateTime structure, by converting (parsing) the given string value into DateTime instance, which is in the specified format.

10. int CompareTo(DateTime value)

It returns:

-1: This instance value is earlier than value.

0: This instance value is equal to value.

1: This instance value is later than value.

11. TimeSpan Subtract(DateTime value)

It returns an instance of TimeSpan structure, that represents date difference between the current instance and given date value.

12. DateTime AddDays(double value)

It creates and returns a new instance of DateTime structure, after adding specified days (+ve or -ve) to the current date & time value.

13. DateTime AddMonths(double value)

It creates and returns a new instance of DateTime structure, after adding specified months (+ve or -ve) to the current date & time value.

14. DateTime AddYears(double value)

It creates and returns a new instance of DateTime structure, after adding specified years (+ve or -ve) to the current date & time value.

15. DateTime AddHours(double value)

It creates and returns a new instance of DateTime structure, after adding specified hours (+ve or -ve) to the current date & time value.

16. DateTime AddMinutes(double value)

It creates and returns a new instance of DateTime structure, after adding specified minutes (+ve or -ve) to the current date & time value.

17. DateTime AddSeconds(double value)

It creates and returns a new instance of DateTime structure, after adding specified seconds (+ve or -ve) to the current date & time value.

18. DateTime AddMilliseconds(double value)

It creates and returns a new instance of DateTime structure, after adding specified milliseconds to the current date & time value.

Math

The System.Math is a static class that provides built-in methods for perform common mathematical operations with numbers.

Example:

```
Math.Pow(2, 3) //returns the value of 2 power 3
```

Properties of 'Math' class

1. const double PI = 3.1415926535897931

It returns the value of mathematical π value as constant, which represents ratio of the circumference of a circle to its diameter.

Methods of 'Math' class

1. static double Pow(double x, double y)

It returns the value of x power y.

2. static double Min(double val1, double val2)

It returns the minimum (smaller) among given two numbers.

3. static double Max(double val1, double val2)

It returns the maximum (bigger) among given two numbers.

4. static double Floor(double value)

It rounds-down the given number. Eg: 10.29 becomes 10.

5. static double Ceiling(double value)

It rounds-up the given number. Eg: 10.29 becomes 11.

6. static double Round(double value)

It rounds up / down the given number, to the nearest integer value.

Eg: 10.29 becomes 10 and 10.59 becomes 11.

7. static double Round(double value, int decimals)

It rounds up / down the given number, to the specified fractional digits.

Eg: Math.Round(value, 2)

10.272 becomes 10.27 and 10.275 becomes 10.38.

8. static int Sign(double value)

It returns:

-1: if the value is a negative value

0: if the value is equal to 0

1: if the value is a positive value

9. static double Abs(double value)

It returns the positive value of given number.

10. static int DivRem(int val1, int val2, out int result)

It returns the quotient value of val1/val2; and provides the remainder value as 'result' out parameter

11. static double Sqrt(double value)

It returns square root value of given number.

Regular Expressions

Regular Expression is a pattern that contains set of conditions of a string value.

Example: ^[a-zA-Z]*$ - for alphabets and spaces on

The 'Regex' class represents regular expression to check whether the string value matches with specified pattern or not.

Useful for validations.

Regex

```
1. Regex referenceVariable = new Regex( “Your Pattern Here”);
2.
3. referenceVariable.IsMatch( value ); //returns true or false
```

Number Systems

In mathematics, a 'Number system' provides a set of digits for counting.

Decimal: Base 10

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Octal: Base 8

0, 1, 2, 3, 4, 5, 6, 7

Binary: Base 2

0, 1

Hexadecimal: Base 16

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

ASCII Table

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

dia.org

Encoding

It is a concept that tells you represent characters into numbers (or any other specific format).

ASCII (American Standard Code for Information Interchange)

- Each character occupies 7 bits (generally considered as 1 byte)
- 128 characters (0 to 127)
- Includes all keyboard characters with alphabets, digits, special characters etc.

UTF-16 / Unicode (Universal Code)

- Each character occupies 2 or 4 bytes (generally considered as 2 bytes)
- About 144697 characters (approx)
- Includes all natural language characters along with ASCII.

'System.IO' namespace

This namespace contains classes to perform File I/O operations.

System.IO:

1. class File
2. class DriveInfo
3. class FileStream
4. class Directory
5. class FileNotFoundException
6. class FileInfo
7. class StreamWriter
8. class BinaryWriter
9. class DirectoryInfo
10. class StreamReader
11. class BinaryReader

class File

It is a static class that manipulates file.

class Directory

It is a static class that manipulates directory (folder).

class FileInfo

It is a class that represents a file on the disk and performs manipulations on files.

class DirectoryInfo

It is a class that represents a directory (folder) on the disk and performs manipulations on directories.

class DriveInfo

It is a class that represents a drive and performs manipulations on drives.

class FileStream

It is a class that performs file I/O operations.

class StreamWriter

It is a class that writes text data into the file.

class StreamReader

It is a class that reads text data from the file.

class BinaryWriter

It is a class that writes binary data into the file.

class BinaryReader

Class that reads binary data from the file.

File

Static class that manipulates file.

All methods of this class are static methods.

Eg:

```
System.IO.File.Method( );
```

Methods of 'File'

1. static FileStream Create(string path)

Create / overwrites a file at the specified path.

2. static bool Exists(string path)

Determines whether the file exists in the disk or not.

3. static void Copy(string sourceFile, string destFile)

Copies the source file to the destination location.

4. static void Move(string sourceFile, string destFile)

Moves the source file to the destination location.

5. static void Delete (string path)

Deletes the specified file permanently.

6. static void WriteAllLines(string path, IEnumerable<string> contents)

Creates / overwrites a file; writes specified lines of content to the file; and close the file.

7. static string[ ] ReadAllLines(string path)

Reads file content as text and returns all lines.

8. static string ReadAllText(string path)

Reads file content as text and returns the same.

9. static void WriteAllText(string path, string contents)

Creates / overwrites a file; writes specified content to the file; and close the file.

10. static FileStream Open(string path, FileMode mode, FileAccess access)

Opens the file in specified file mode with specified file access permission and returns a new object of FileStream type.

11. static FileStream OpenRead(string path)

Opens the file in 'Open' mode and returns a new object of FileStream type.

12. static FileStream OpenWrite(string path)

Opens the file in 'OpenOrCreate' mode and returns an object of FileStream type.

FileInfo

Represents a file and provides methods to manipulate the file.

Eg:

```
FileInfo referenceVariable = new FileInfo( “Your File Path Here”);
```

Constructors of 'FileInfo'

1. FileInfo(string filePath)

It initializes the file path which needs to be manipulated.

Methods of 'FileInfo'

1. FileInfo CopyTo(string destFilePath, bool overwrite)

Copies the file into the new destination path.

It returns an object of FileInfo class, that represents the newly created file.

2. void MoveTo(string destFilePath)

Moves the file into the new destination path (should be in the same drive).

3. void Delete( )

Deletes the file permanently.

4. FileStream Create( )

Creates the file at specified path & creates and returns an object of FileStream class with 'Create' mode, which can write data to the same file.

5. FileStream Open(FileMode mode, FileAccess access)

Opens the file at specified file mode and specified file access permission & creates and returns an object of FileStream class that can write / read the file data.

6. FileStream OpenRead( )

Opens the file in 'Read' mode & creates and returns an object of FileStream class that can read the file data.

7. FileStream OpenWrite( )

Opens the file in 'OpenOrCreate' mode & creates and returns an object of FileStream class that can read the file data.

8. StreamWriter CreateText( )

Opens the file in 'Create' mode & creates and returns an object of StreamWriter class that can write text to the file.

9. StreamWriter AppendText( )

Opens the file in 'Append' mode & creates and returns an object of StreamWriter class that can write the appended text to the file.

10. StreamReader OpenText( )

Opens the file in 'Open' mode & creates and returns an object of StreamReader class that can read text from the file.

Properties of 'FileInfo'

bool Exists

Determines whether the file exists in the disk or not.

string FullName

Represents full path (including file name and extension) of the file.

string Name

Represents only name of the file (without path).

string DirectoryName

Represents only path of the file (without file name).

string Extension

Represents only file extension (without file name).

DateTime CreationTime

Represents date and time of file creation.

DateTime LastWriteTime

Represents date and time of last modification of the file.

DateTime LastAccessTime

Represents date and time of last access of the file.

long Length

Represents file size (in the form of no. of bytes).

Directory

Static class that manipulates directory (folder).

All methods of this class are static methods.

Eg: `System.IO.Directory.Method( );`

Methods of 'Directory'

1. static DirectoryInfo CreateDirectory(string path)

Create a directory at the specified path.

2. static void Delete (string path, bool recursive)

true: Deletes the directory including sub directories and all files.

false: Deletes the directory only if it is empty.

3. static bool Exists(string path)

Determines whether the directory exists in the disk or not.

4. static string[ ] GetDirectories(string path)

Returns string[ ] that contains paths of sub directories of specified directory.

5. DirectoryInfo[ ] GetDirectories (string searchPattern)

Returns DirectoryInfo[ ] that represents sub directories that matches with specified search pattern.

6. FileInfo[ ] GetFiles( )

Returns FileInfo[ ] that represents files of current directory.

7. FileInfo[ ] GetFiles(string searchPattern)

Returns FileInfo[ ] that represents files that matches with specified search pattern.

8. void MoveTo(string destDirName)

Moves the current directory to the specified location in the same

DirectoryInfo

Class that represents a directory (folder) on the disk and performs manipulations on directories.

Eg:

```
DirectoryInfo referenceVariable = new DirectoryInfo( “Directory Path Here”);
```

Constructors of 'DirectoryInfo'

1. DirectoryInfo(string path)

It initializes the directory path which needs to be manipulated.

Methods of 'DirectoryInfo'

1. void Create( )

Create the directory at the current path.

2. DirectoryInfo CreateSubDirectory (string path)

Creates a sub directory at the current path with specified name.

3. void Delete(bool recursive)

true: Deletes the directory including sub directories.

false: Deletes the directory only if it is empty.

4. DirectoryInfo[ ] GetDirectories()

Returns DirectoryInfo[ ] that represents sub directories of current directory.

5. DirectoryInfo[ ] GetDirectories (string searchPattern)

Returns DirectoryInfo[ ] that represents sub directories that matches with specified search pattern.

6. FileInfo[ ] GetFiles( )

Returns FileInfo[ ] that represents files of current directory.

7. FileInfo[ ] GetFiles(string searchPattern)

Returns FileInfo[ ] that represents files that matches with specified search pattern.

8. void MoveTo(string destDirName)

Moves the current directory to the specified location in the same drive

Properties of 'DirectoryInfo'

1. bool Exists

Determines whether the directory exists in the disk or not.

2. string FullName

Represents full path of the directory

3. string Nme

Represents only name of the directory (without path).

4. DirectoryInfo Parent

Represents parent directory of the current directory.

5. DirectoryInfo Root

Represents root (drive) of the directory.

6. DateTime CreationTime

Represents date and time of directory creation.

7. DateTime LastWriteTime

Represents date and time of last modification of the directory.

8. DateTime LastAccessTime

Represents date and time of last access of the directory.

DriveInfo

Class that represents a drive and performs manipulations on drives.

You can read both fixed / removable drives.

Eg:

```
1. DriveInfo referenceVariable = new DriveInfo( “Your Drive Name Here”);
```

Constructor of 'DriveInfo'

1. DriveInfo(string path)

It initializes the drive name which needs to be manipulated.

Properties of 'DriveInfo'

1. string Name

Represents name of the drive.

2. string DriveType

Represents type of drive either 'Fixed' or 'Removable'

3. string VolumeLabel

Represents label of the drive (set by user).

4. DirectoryInfo RootDirectory

Represents root directory of the drive.

5. long TotalSize

Represents total size (bytes) of the drive.

6. long AvailableFreeSpace

Represents total free space (bytes) of the drive.

FileStream

Class that performs file I/O operations.

Writes / reads data in byte[] format.

Eg:

```
1. FileStream referenceVariable = new FileStream( "File Path", FileMode.Create,
   FileAccess.Write );
```

Constructors of 'FileStream'

1. FileStream(string filePath, FileMode mode, FileAccess access)

It initializes opens the file at specified mode and specified access permission.

FileMode: CreateNew, Create, Open, OpenOrCreate, Append

FileAccess: Read, Write, ReadWrite

'FileMode' in FileStream

1. CreateNew

It specifies that the o/s should create a new file.

If the file already exists an IOException will be thrown.

2. Create

It specifies that the o/s should create a new file.

If the file already exists, it will be overwritten.

3. Open

It specifies that the o/s should open an existing file.

If the file doesn't exist, a FileNotFoundException will be thrown.

4. OpenOrCreate

It specifies that the o/s should open an existing file.

If the file doesn't exist, a new file will be created.

It is useful for reading content of the file.

5. Append

It specifies that the o/s should open an existing file; and seek to end of the file, in order to write content at the end.

It is useful with FileAccess.Write

'FileAccess' in FileStream

1. Read

It is used to read content from an existing file.

2. Write

It is used to write content to a file.

3. ReadWrite

It can be used for both read / write operations in a file.

Methods of 'FileStream'

1. void Write(byte[] array, int offset, int count)

Writes the specified no. of bytes specified by count, based on the byte[] specified by 'array' into the file, after the no. of bytes specified by 'offset'.

2. int Read(byte[] array, int offset, int count)

Read the specified no. of bytes specified by count, into the byte[] specified by 'array' from the file, after the no. of bytes specified by 'offset'.

3. void Close( )

Closes the file.

StreamWriter

Class that writes text data into the file.

Internally uses FileStream.

Eg:

```
StreamWriter referenceVariable = new StreamWriter( "File Path" );
```

Constructors of 'StreamWriter'

1. StreamWriter(FileStream stream)

It initializes the StreamWriter based on the specified FileStream.

2. StreamWriter(string path)

It initializes the StreamWriter for the specified file path.

Methods of 'StreamWriter'

1. void Write(string content)

Writes the specified string content to the file.

2. void Close( )

Close the file.

StreamReader

Class that reads text data from the file.

Internally uses FileStream.

Eg:

```
1. StreamReader referenceVariable = new StreamReader( "File Path" );
```

Constructors of 'StreamReader'

1. StreamReader(string path)

It initializes the StreamReader for the specified file path.

2. StreamReader(FileStream stream)

It initializes the StreamReader based on the specified FileStream

Methods of 'StreamReader'

1. int Read(char[] buffer, int startIndex, int count)

Reads the specified number of characters (using 'count') from the file into the specified buffer (char[]) and inserts the characters at specified index of buffer.

2. string ReadToEnd( )

Reads complete content of the file in text format and returns the same as a string.

3. string ReadLine( )

Reads a line from the file in text format and returns the same as a string.

4. void Close( )

Close the file.

BinaryWriter

Class that writes binary data into the file.

Internally uses FileStream.

For example, 65 is written as 100 0001.

Eg:

```
BinaryWriter referenceVariable = new BinaryWriter( fileStream );
```

Constructors of 'BinaryWriter'

1. BinaryWriter(FileStream stream)

It initializes the BinaryWriter based on the specified FileStream.

Methods of 'BinaryWriter'

1. void Write(…)

Writes any type of value (only primitive types / string) into the file.

2. void Close( )

Close the file.

BinaryReader

Class that reads binary data from the file.

Internally uses FileStream.

Eg:

```
BinaryReader referenceVariable = new BinaryReader( fileStream );
```

Methods of 'BinaryReader'

1. byte ReadByte( )

Reads and returns a byte value from the file.

2. ReadSByte( ), ReadInt16( ), ReadUInt16( ), ReadInt32( ), ReadUInt32( ), ReadInt64( ), ReadUInt64( ), ReadSingle( ), ReadDouble( ), ReadDecimal( ), ReadChar( ), ReadString( ), ReadBoolean( )

Reads and returns the specific type of value from the file.

3. string Close( )

Closes the file.

BinaryFormatter

Class that serializes (converts) an object-state into binary format into and stores in binary file.

Serialization is a process of converting an object from one format to another format.

It can also read existing object-state from the binary file.

Full Path: `System.Runtime.Serialization.Formatters.Binary.BinaryFormatter`

Eg:

```
1. BinaryFormatter referenceVariable = new BinaryFormatter( );
```

Methods of 'BinaryFormatter'

1. void Serialize(FileStream FileStream, object data)

Converts the object-state into the binary file, using the specified FileStream (with Write access).

2. object Deserialize(FileStream FileStream)

Reads the existing object-state from the binary file, using the specified FileStream (with Read access).

JavaScriptSerializer

Class that serializes (converts) an object-state into JSON format.

It can also convert JSON data into object of any class.

Full Path: `System.Web.Script.Serialization.JavaScriptSerializer`

Eg:

```
1. //serialization:
2. javaScriptSerializer.Serialize( YourObject );
3.
4.
5. //deserialization:
6. javaScriptSerializer.Deserialize( string jsonData, typeof(ClassName) );
```

XmlSerializer

Class that serializes (converts) an object-state into XML format and stores in XML file.

It can also read existing object-state from the XML file.

Full Path: `System.Xml.Serialization.XmlSerializer`

XmlSerializer

```
XmlSerializer referenceVariable = new XmlSerializer(
typeof(ClassName) );
```

Methods of 'XmlSerializer'

1. void Serialize(FileStream FileStream, object data)

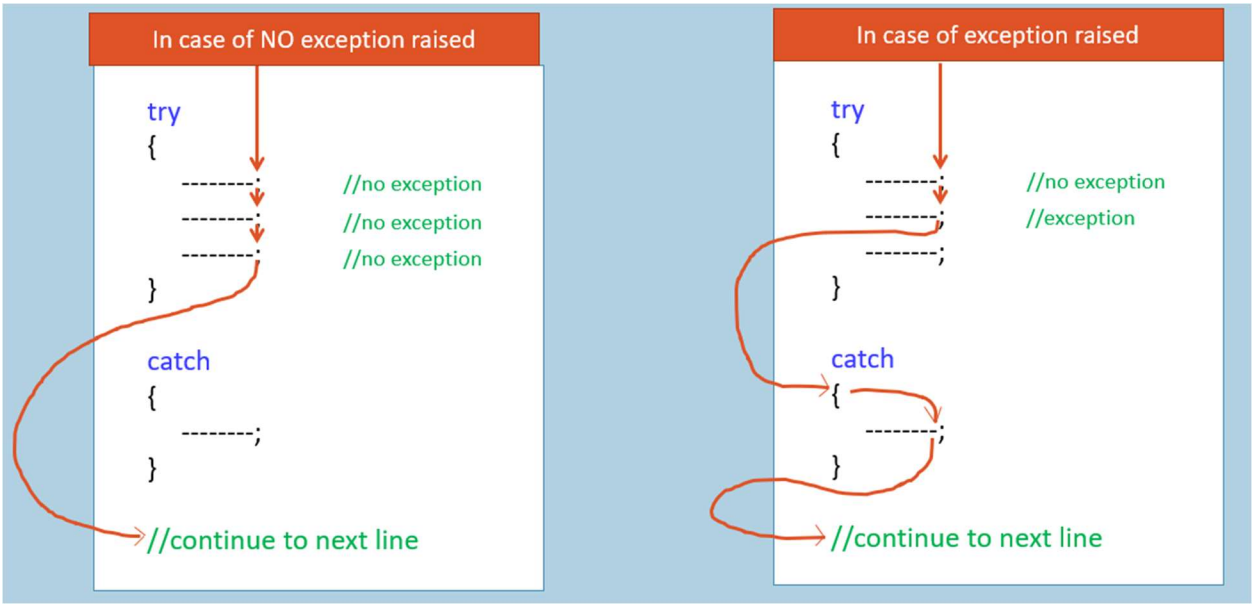
Converts the object-state into the xml file, using the specified FileStream (with Write access).

2. object Deserialize(FileStream fileStream)

Reads the existing object-state from the xml file, using the specified FileStream (with Read access).

Exception Handling

- Exception is a run time error occurs while executing the application.
- When exception occurs, the current application terminates abruptly.
- Exception Handling avoids abrupt termination of the application, in case of exception.




```
try
{
  statement1;
  statement2;
  statement3;
  …
}
catch (ExceptionClassName variable)
{
  //do something with exception variable
  //or show error message
}
finally
{
  //do some cleaning process
}
```

- When CLR is unable to execute a statement, it is treated as exception.
- ‘try’ and ‘catch’ blocks are mandatory.
- ‘finally’ block and multiple ‘catch’ blocks are optional.
- "try" block contains all the actual code, where exceptions may occurs.
- Multiple "try" blocks for one catch block is not allowed. Nested "try" blocks is allowed
- "catch" block contains error handling code; it executes only when a particular type of exception is raised during the execution of "try" block. Multiple "catch" blocks is allowed.
- "finally" block executes after successful completion of "try" block; or after any catch block. It is optional.
- "throw" keyword is used to throw built-in or custom exceptions, in case of invalid values found.

FormatException

FormatException represents an error when it is unable to convert a string to a number, as the string contains characters other than digits (such as alphabets, spaces etc.).

Occurs when calling methods of 'Convert' class or 'Parse' method of numeric types with incorrect string value.

It's a bad practice to throw FormatException implicitly / explicitly.

```
throw new FormatException(…)
```

Constructors of 'FormatException'

1. FormatException()

It initializes nothing.

2. FormatException(string message)

It initializes 'Message' property.

3. FormatException(string message, Exception innerException)

It initializes 'Message' and 'InnerException' properties.

IndexOutOfRangeException

IndexOutOfRangeException represents an error when an index was supplied outside the available range to an array.

Occurs when accessing an array with a wrong index value, which is less than 0 or greater than or equal to size of the array.

It's a bad practice to throw IndexOutOfRangeException implicitly / explicitly.

```
1. throw new IndexOutOfRangeException(…)
```

Constructor of 'IndexOutOfRangeException'

1. IndexOutOfRangeException()

It initializes nothing.

2. IndexOutOfRangeException(string message)

It initializes 'Message' property.

3. IndexOutOfRangeException(string message, Exception innerException)

It initializes 'Message' and 'InnerException' properties.

NullReferenceException

NullReferenceExeption represents an error when you try to access a property, indexer or method through a reference variable when its value is null.

It's a bad practice to throw NullReferenceException implicitly / explicitly.

```
throw new NullReferenceException(…)
```

Constructors of 'NullReferenceException'

1. NullReferenceException()

It initializes nothing.

2. NullReferenceException(string message)

It initializes 'Message' property.

3. NullReferenceException(string message, Exception innerException)

It initializes 'Message' and 'InnerException' properties.

ArgumentNullException

ArgumentNullException represents an error when a null value is passed as argument to a method; and that method doesn't accept null's into that parameter.

It's a good practice to throw ArgumentNullException implicitly / explicitly.

```
throw new ArgumentNullException(…)
```

Constructors of 'ArgumentNullException'

1. ArgumentNullException()

It initializes nothing.

2. ArgumentNullException(string paramName)

It initializes 'ParamName' property.

3. ArgumentNullException(string message, Exception innerException)

It initializes 'Message' and 'InnerException' properties.

4. ArgumentNullException(string paramName, string message)

It initializes 'ParamName' and 'Message' properties.

ArgumentOutOfRangeException

ArgumentOutOfRangeException represents an error when a numeric value is passed as argument to a method; and it outside the acceptable range of values accepted by that method.

It's a good practice to throw ArgumentOutOfRangeException implicitly / explicitly.

```
throw new ArgumentOutOfRangeException(…)
```

Constructors of 'ArgumentOutOfRangeException'

1. ArgumentOutOfRangeException()

It initializes nothing.

2. ArgumentOutOfRangeException(string paramName)

It initializes 'ParamName' property.

3. ArgumentOutOfRangeException(string message, Exception innerException)

It initializes 'Message' and 'InnerException' properties.

4. ArgumentOutOfRangeException(string paramName, string message)

It initializes 'ParamName' and 'Message' properties.

5. ArgumentOutOfRangeException(string paramName, object actualValue, string message)

It initializes 'ParamName', 'ActualValue' and 'Message' properties.

ArgumentException

ArgumentException represents an error when the argument value of a parameter is invalid as per any validation rules / requirements.

It's a good practice to throw ArgumentException implicitly / explicitly.

```
throw new ArgumentException(…)
```

Constructors of 'ArgumentException'

1. ArgumentException()

It initializes nothing.

2. ArgumentException(string message)

It initializes 'Message' property.

4. ArgumentException(string message, Exception innerException)

It initializes 'Message' and 'InnerException' properties.

5. ArgumentException(string message, string paramName)

It initializes 'Message' and 'ParamName' properties.

6. ArgumentException(string message, string paramName, string innerException)

It initializes 'Message', 'ParamName' and 'InnerException' properties.

InvalidOperationException

InvalidOperationException represents an error when you call a method; and it is invalid to call it a per current state of the object.

It's a good practice to throw InvalidOperationException implicitly / explicitly.

```
throw new InvalidOperationException(…)
```

Constructors of 'InvalidOperationException'

1. InvalidOperationException()

It initializes nothing.

2. InvalidOperationException(string message)

It initializes 'Message' property.

3. InvalidOperationException(string message, Exception innerException)

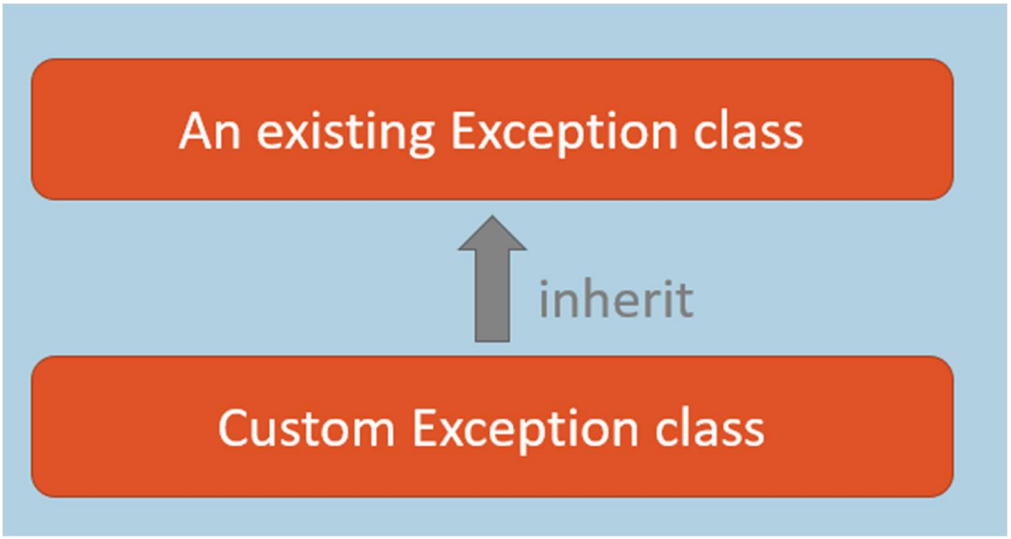
It initializes 'Message' and 'InnerException' properties.

Custom Exception class

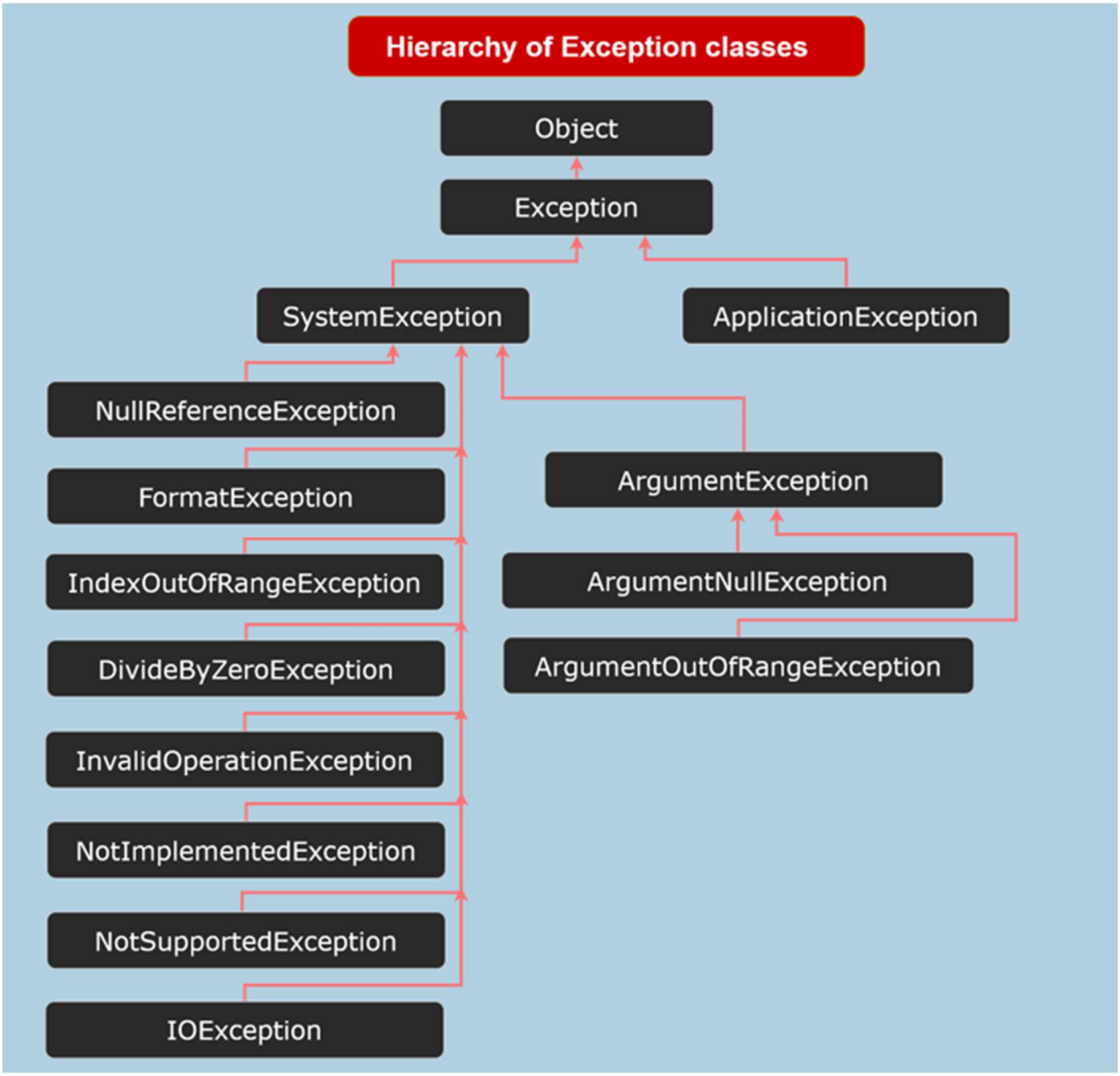
Custom Exception class is a class that is derived from any one of the exception classes (such as System.Exception or any other exception class).

If none of the pre-defined exception class meets your requirement, you will create an user-defined exception class (custom exception class).

```
throw new CustomExceptionClassName(…)
```



Hierarchy of Exception Classes



All exception class in .net are derived from System.SystemException class.

System.SystemException is derived from System.Exception class.

Custom Exception classes should NOT be inherited from ApplicationException.

Catch-When

New feature introduced in C# 7.1.

The "catch" block catches the exception, only when the given condition "true".

"Catch-when" is also known as "Exception Filters".

Eg:

```
1. try
2. {
3.   //statements
4. }
5. catch (ExceptionType referenceVariable) when (condition)
6. {
7.   //error handling
8. }
```

'nameof' operator

Introduced in C# 6.0.

Returns actual name of the specified field / property.

Useful when you are writing same code for multiple properties.

Eg:

```
nameof ( FieldOrPropertyName )
```

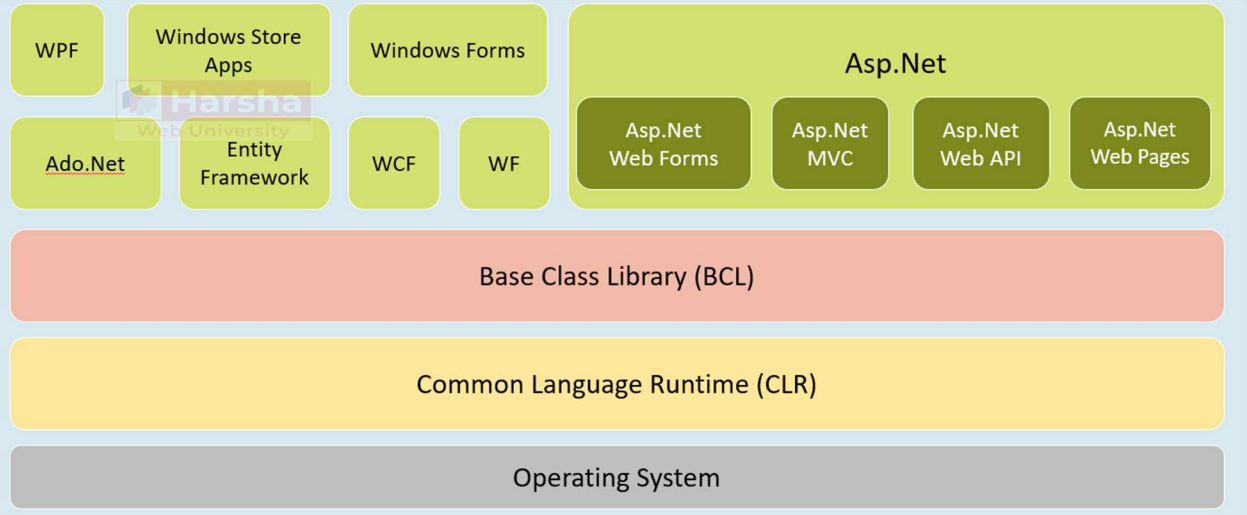
.Net Framework [vs] .Net Core [vs] .Net

.Net Framework: Since 2002

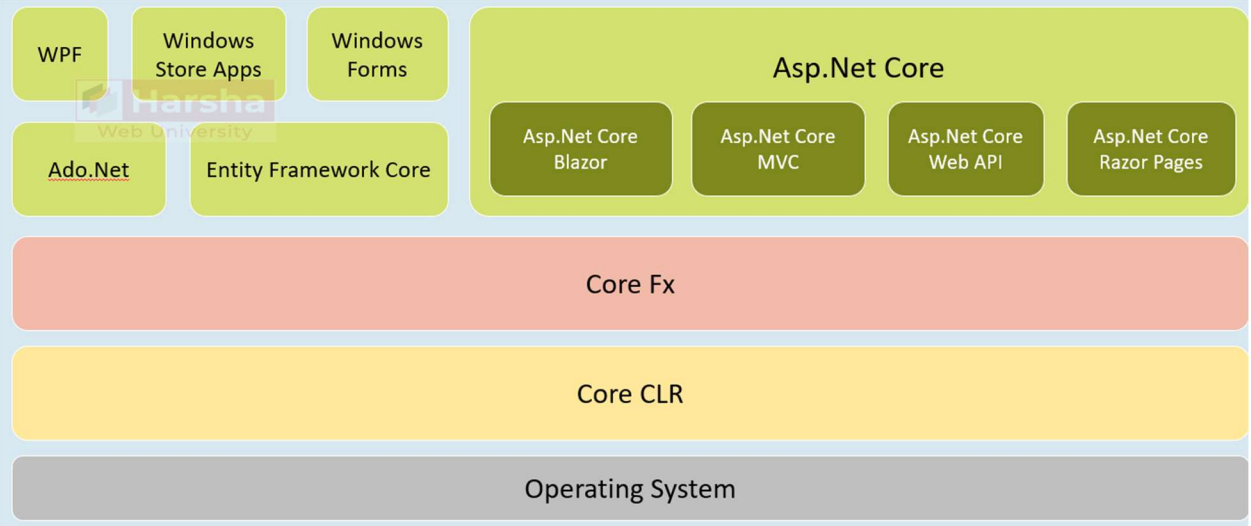
.Net Core: Since 2016

.Net: Since 2020

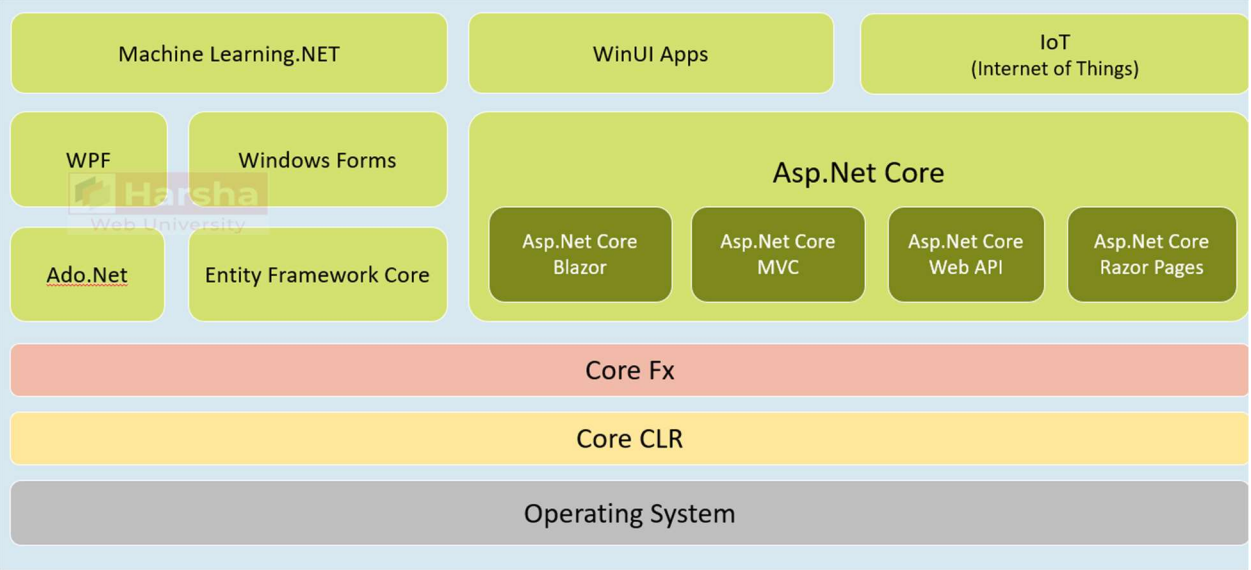
.Net Framework Closed-source, Monolithic, Thick, Average-performance



.Net Core Open-source, Modular, Cross-platform, Minimalistic, Faster-performance



.Net Unified, Open-source, Modular, Cross-platform, Minimalistic, Faster-performance



Top Level Statements

Allows a sequence of statements to occur right before the namespaces / type definitions in a single file in the C# project.

File1.cs

```
statements…
namespaces / types…
```

-- would compile as:

```
static class Program
{
   static async Task Main(string[ ] args)
   {
     //statements…
   }
}
```

- **Advantage:** Make C# learning curve easy for C# learners (newbies).
- The compiler-generated class and Main method are NOT accessible through code of any other areas of the project.
- The compiled Main method would be 'async', by default. So it allows 'await' statements in top-level statements.
- Only one compilation unit (C# file) can have top level statements in a C# project.

- The local variables / local functions declared in the top-level statements are NOT accessible elsewhere (in other types / files).
- Top level statements can access command-line arguments using 'args'. A "string[ ] args" parameter would be generated by the compiler automatically.

File Scoped Namespaces

Allows you to declare a namespace at the top of the file (before/after the 'using' statements) and all types of the same file would be a part of that namespace.

File1.cs

```
using statements…
namespace namespace_name;
using statements…
types…
```

-- would compile as:

```
using statements…

namespace namespace_name
{
  //types
}
```

- **Advantage:** Allows developers to quickly create one-or-few types in a namespace without nesting them in the 'namespace declaration'.
- Only one 'file-scoped namespace' statement is allowed for one source file (C# file).
- The 'file-scoped namespace' statement CAN be written before / after the 'using' statements.
- A source file can't contain both 'file-scoped namespace' and 'normal namespace declarations'.

Global 'using' directives

Allows you import a namespace for the entire project, by adding 'global' keyword to the 'using' statement.

File1.cs

```
global using namespace_name;
namespaces / types…
```

-- would compile as:

```
using namespace_name; (and in other files also)

namespace_names / types…
```

Advantage: Allows developers to reduce attention on lengthy 'using' statements at the top of every file; but concentrate on actual code (types in the file).

It is recommended to write all 'global using' statements in a separate file (one-for each project)

The following namespaces are implicitly imported in every C# project implicitly:

1. System
2. System.Collections.Generic
3. System.IO
4. System.Linq
5. System.Net.Http
6. System.Threading
7. System.Threading.Tasks

Module Initializers

Allows you to run some code with 'global initialization logic' at application startup, when the application loads into memory.

File1.cs

```
1. using System.Runtime.CompilerServices;
2.
3. class class_name
4. {
5.    [ModuleInitializer]
6.    internal static void method_name( )
7.    {
8.    }
9. }
```

It would execute at application startup (before the Main method).

Advantage over static constructors: The static constructors execute ONLY if the class is used at least once; otherwise will NOT execute.

One project CAN have more than module initializer methods (if so, they are called based on alphabetical order of file names).

The initializer method must be:

1. Either "internal", "protected internal" or "public" only.
2. Static method
3. Parameterless method
4. Return type is 'void'

5. Not be a generic method
6. Can't be a local function

Use Cases:

- Loading environment variables
- Initializing connection strings / database server names
- Initializing URL's of API servers
- Loading Azure connection strings
- Initializing file paths

etc.

The initializer class must be:

- Either "internal" or "public" only.
- Can be static class [optionally]
- Not be a generic class

Nullable Reference Types

Introduces 'nullable reference types' and 'non-nullable reference types' to allow the compiler to perform

'static flow analysis' for purpose of null-safety.

```
1. class_name variable_name; //'class_name' is non-nullable reference type
2. class_name? variable_name; //'class_name?' is nullable reference type
```

Advantage: The compiler can perform a static analysis to identify where there is a possibility of 'null' values and can show warnings; so we can avoid NullReference Exceptions at coding-time itself.

By default, all classes and interfaces are 'non-nullable reference types'. To convert them as 'nullable reference type', suffix a question mark (?). Eg: class?

- **Null forgiving operator (!)**
- Meaning: "I'm sure, it's not null".
- Suffix your expression (variable or property) with "!" operator to make that expression as "not null", at compilation time.
- It has no effect at run time.
- It means, the developer says to the C# compiler - that, a variable or property is "not null". But at run time, if it is actually null, it leads to "NullReference Exception" as normal.
- So use this operator only when you are sure that your expression (variable of property) is NOT null.

Target-typed 'new' expressions

Allows the developer "not-to-mention" the class name; but allows to create an object in the 'new' expression.

```
class_name variable_name = new( ); //equivalent to "new class_name( )"
```

Benefit: We can create object of a class in shortcut way.

It can't be used in:

```
1. //using block:
2. using (var variable = new( ) )
3. {
4. }
5.
6. //foreach:
7. foreach (var variable in new( ) )
8. {
9. }
```

Pattern Matching - Overview

Enables developers to easily check the data type of a variable and also check its value with some conditions.

"is" expression:

```
1. if (variable_name is class_name another_variable)
2. {
3.   if (another_variable.property == value)
4.   {
5.     statements…
6.   }
7. }
```

"switch-case" expression:

```
1. switch (variable_name)
2. {
3.   case class_name another_variable
4.       when another_variable.property == value:
5.          statements…
6.   break;
7. }
```

"switch" expression:

```
variable_name switch {
  class_name another_variable when another_variable.property == value  => result_expression
}
```

Pattern Matching - Type Pattern - with "if"

Regular Code:

```
//Check whether the variable is of specified 'class_name' type.
if (variable.GetType( ) == typeof(class_name) || variable.GetType().IsSubClassOf(typeof(class_name))
{
   statements…
}
```

"is" expression:

```
//Check whether the variable is of specified 'class_name' type
if (variable is class_name)
{
   statements…
}
```

Pattern Matching - Type Pattern - with "if" - with Variable

Regular Code:

```
//Check whether the variable is of specified 'class_name' type.
if (variable.GetType( ) == typeof(class_name) || variable.GetType( ).IsSubClassOf(typeof(class_name))
{
   //typecast the value into the specified class
   class_name another_variable = (class_name)variable_name;

   statements…
}
```

"is" expression:

```
//Check whether the variable is of specified 'class_name' type & also typecast the value into specified class.
if (variable is class_name another_variable)
{
   statements…
}
```

Pattern Matching - Type Pattern - with "switch-case"

Regular Code:

```
1. //Check whether the variable is of specified 'class_name' type.
2. switch (variable.GetType().Name)
3. {
4.    case "class_name":
5.       statements; break;
6. }
```

"switch-case" expression:

```
1. //Check whether the variable is of specified 'class_name' type
2. switch (variable)
3. {
4.    case class_name:
5.       statements…; break;
6. }
```

Pattern Matching - Type Pattern - with "switch-case" - with variable

Regular Code:

```
1. //Check whether the variable is of specified 'class_name' type.
2. switch (variable.GetType( ).Name)
3. {
4.    case "class_name":
5.        //typecast the value into the specified class
6.        class_name another_variable = (class_name)variable_name;
7.
8.        statements; break;
9. }
```

"switch-case" expression:

```
1. //Check whether the variable is of specified 'class_name' type
2. switch (variable)
3. {
4.    case class_name another_variable:
5.       statements…; break;
6. }
```

Pattern Matching - Type Pattern - with 'when' - with "switch-case"

Regular Code:

```
1. //Check whether the variable is of specified 'class_name' type.
2. switch (variable.GetType( ).Name)
```

```
3. {
4.     case "class_name":
5.           //typecast the value into the specified class
6.           class_name another_variable = (class_name)variable_name;
7.
8.           if (another_variable.property == value)
9.           {
10.                 statements;
11.          }
12.          break;
13. }
```

"switch-case" expression:

```
1. //Check whether the variable is of specified 'class_name' type
2. switch (variable)
3. {
4.      case class_name another_variable
5.           when another_variable.property == value:
6.              statements…; break;
7. }
```

Pattern Matching - Type Pattern - with 'when' - with "switch expression"

Regular Code:

```
1. //Check whether the variable is of specified 'class_name' type.
2. switch (variable.GetType( ).Name)
3. {
4.    case "class_name":
5.       //typecast the value into the specified class
6.       class_name another_variable = (class_name)variable_name;
7.
8.       if (another_variable.property == value)
9.       {
10.            statements;
11.      }
12.      break;
13. }
```

"switch" expression:

```
1. //Check whether the variable is of specified 'class_name' type
2. variable switch
3. {
4.     class_name another_variable
5.           when another_variable.property == value
6.           => statements…
7. }
```

Pattern Matching - Relational Pattern

```
1. //Check whether the variable is of specified 'class_name' type
2. variable switch
3. {
4.    class_name another_variable when
5.       another_variable.property is value //another_variable.property == value
6.       another_variable.property is < value //another_variable.property < value
7.       another_variable.property is > value //another_variable.property > value
8.       another_variable.property is <= value //another_variable.property <=
   value
9.       another_variable.property is >= value //another_variable.property >=
   value
10.      => result_expression…
11. }
```

Pattern Matching - Logical Pattern

```
1. //Check whether the variable is of specified 'class_name' type
2. variable switch
3. {
4.    class_name another_variable when
5.       another_variable.property is expression1 and expression2 //conjunctive
   pattern (and)  // another_variable.property == expression1 &&
   another_variable.property == expression2
6.
7.       another_variable.property is expression1 or expression2 //disjunctive
   pattern (or)  //another_variable.property == expression1 ||
   another_variable.property == expression2
8.
9.       another_variable.property is not expression //negated pattern (not)  //
   another_variable.property != expression
10.
11.      => result_expression…
12. }
```

Pattern Matching - Property Pattern

```
1. variable switch
2. {
3.    { property: value } //variable.property == expression
4.    { property: < value } //variable.property < value
5.    { property: > value } //variable.property > value
6.    { property: <= value } //variable.property <= value
7.    { property: >= value } //variable.property >= value
8.    => result_expression…
9. }
```

Pattern Matching - Tuple Pattern

```
1. (variable.property1, variable.property2) switch
2. {
3.    ( expression1, expression2 ) //variable.property1 == expression1 &&
   variable.property2 == expression2
4.    => result_expression…
5.
6.    ( expression1, expression2 ) //variable.property1 == expression1 &&
   variable.property2 == expression2
7.    => result_expression…
8. }
```

Pattern Matching - Positional Pattern

```
1. variable switch {
2.   ( expression1, expression2 ) //variable.property1 == expression1 &&
   variable.property2 == expression2
3. => result_expression…
4.
5.   ( expression1, expression2 ) //variable.property1 == expression1 &&
   variable.property2 == expression2
6. => result_expression…
7. }
```

Deconstruct method:

```
1. public void Deconstruct(out type1 variable1, out type2 variable2)
2. {
3.   variable1 = this.property1;
4.   variable2 = this.property2;
5. }
```

Pattern Matching - Nested Property Pattern

```
1. variable switch
2. {
3.    { outer_property: { nested_property: value } }
   //variable.outer_property.nested_property == expression
4.    { outer_property: { nested_property: < value } }
   //variable.outer_property.nested_property < value
5.    { outer_property: { nested_property: > value } }
   //variable.outer_property.nested_property > value
6.    { outer_property: { nested_property: <= value } }
   //variable.outer_property.nested_property <= value
7.    { outer_property: { nested_property: >= value } }
   //variable.outer_property.nested_property >= value
8.    => result_expression…
```

```
}
```

Pattern Matching - Extended Property Pattern

```
variable switch
{
    { outer_property.nested_property: value }
//variable.outer_property.nested_property == expression
    { outer_property.nested_property: < value }
//variable.outer_property.nested_property < value
    { outer_property.nested_property: > value }
//variable.outer_property.nested_property > value
    { outer_property.nested_property: <= value }
//variable.outer_property.nested_property <= value
    { outer_property.nested_property: >= value }
//variable.outer_property.nested_property >= value
    => result_expression…
}
```

Need of Immutability

Goal: The values of fields and properties should be readonly (immutable). No other classes can change them, after they get initialized.

Immutable class:

```
class class_name
{
    data_type readonly field_name; //readonly field

    data_type property_name { get => field_name } //readonly property
}
```

Benefits:

- Avoid unexpected value changes in response data retrieved from API servers.
- Avoid unexpected value changes in the data retrieved from database servers.
- Use objects of immutable classes as 'key' in Dictionary and in Hashtable.
- Avoid unexpected value changes in objects while multiple threads access the same objects simultaneously.

Immutable Classes

A class with readonly fields and readonly properties.

Immutable class:

```
class class_name
```

```
{
    data_type readonly field_name; //readonly field

    data_type property_name { get => field_name } //readonly property

    public class_name( ) //constructor
    {
       field_name = value; //initialize the field
    }
}
```

'init' only properties

'init' only properties can be initialized either inline with declaration, in the constructor or in the object initializer.

Init-only property

```
data_type property_name { get; init; } //'init' instead of 'set'
```

Immutable class with 'init' only properties

```
class class_name
{
    data_type readonly field_name; //readonly field

    data_type property_name
    {
       get => field_name //get accessor
       init => field_name = value; //init accessor instead of 'set' accessor
    }

    public class_name( ) //constructor
    {
       field_name = value; //initialize the field
    }
}
```

Object of immutable class:

```
class variable_name = new class_name( ) { property_name = value; }
//initialize value of 'init' only property in object initializer
```

Readonly structs

Enforces you to write only 'readonly fields' and 'readonly properties' to achieve immutability in your struct.

Readonly struct

```
readonly struct struct_name
```

```
{
    //readonly fields
    //readonly properties
}
```

Readonly struct

```
readonly struct struct_name
{
    data_type readonly field_name; //readonly field

    data_type property_name
    {
        get => field_name //get accessor
        init => field_name = value; //init accessor instead of 'set' accessor
    }

    public class_name( ) //constructor
    {
        field_name = value; //initialize the field
    }
}
```

Parameterless Struct Constructors

Struct with parameter-less constructor

```
readonly struct struct_name
{
    data_type readonly field_name; //readonly field

    data_type property_name
    {
        get => field_name //get accessor
        init => field_name = value; //init accessor instead of 'set' accessor
    }

    public class_name( ) //constructor
    {
        field_name = value; //you must initialize all the fields
    }
}
```

Records

Goal: Concise syntax to create a reference-type with immutable properties.

Record

```
record record_name(data_type Property1, data_type Property2, …);
```

-- would be compiled as:

Compiled code of Record

```
class record_name
{
    public data_type Property1 { get; init; }
    public data_type Property2 { get; init; }

    public record_name(data_type Parameter1, data_type Parameter2)
    {
       this.Property1 = Parameter1;
       this.Property2 = Parameter2;
    }
}
```

Features:

- Records are 'immutable' by default.
- All the record members become as 'init-only' properties.
- Records can also be partially / fully mutable - by adding mutable properties.
- Supports value-based equality.
- Supports inheritance.
- Supports non-destructive mutation using 'with' expression.

Records with mutable properties:

```
record record_name(data_type Property_name, …)
{
    data_type Property_name { get; set; }
}
```

Records - Equality

Supports value-based equality.

Records provide a compiler-generated Equals() method and overloads == and != operators that compares two instances of records that compare the values of fields (but doesn't compare references) .

Record - 'Equals' method

```
record_name variable1 = new record_name(value1, value2);
record_name variable2 = new record_name(value1, value2);
variable1 == variable2; //true
```

```
4. variable1.Equals(variable2); //true
```

Records - "with" expression

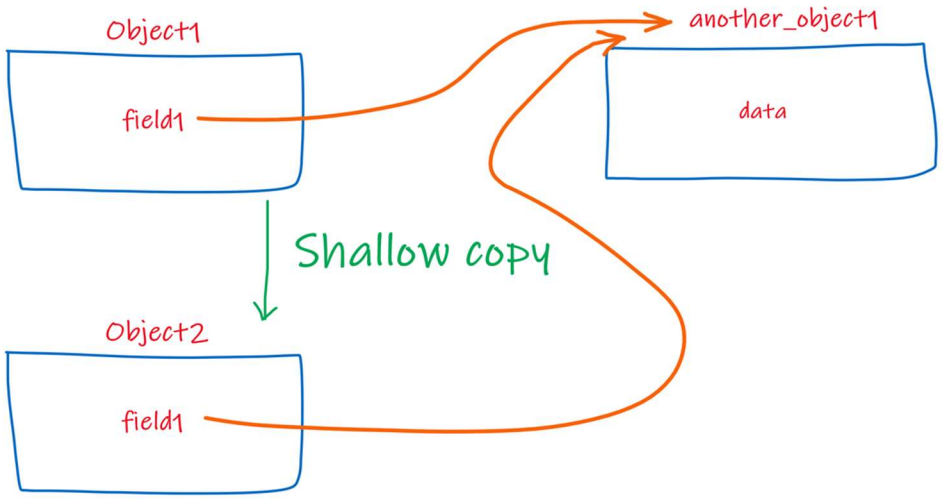
'with' expression acts as object initializer for 'records'.

It creates a shallow copy of an existing record object and also overwrites the values of specified properties.

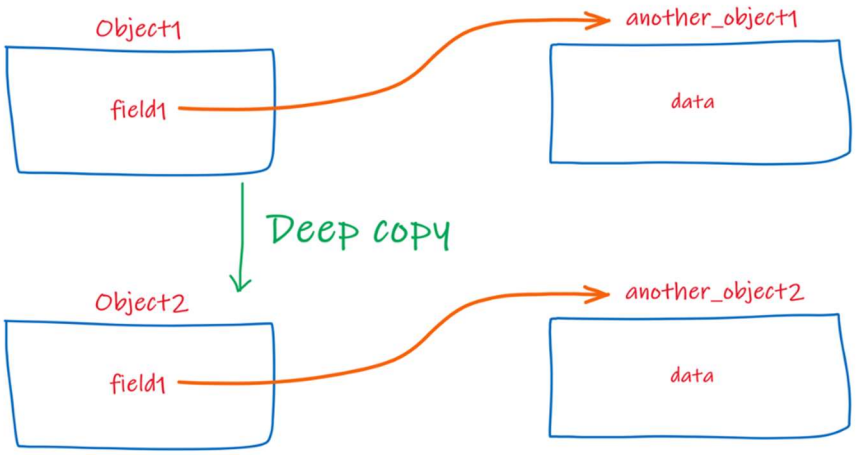
Record - 'with' expression

```
1. record_name variable1 = new record_name(value1, value2);
2. record_name variable2 = variable1 with { Property = value, … } //with
   expression
```

Shallow copy:



Deep copy:



Records - "Deconstruct"

A compiler-generated 'Deconstruct' method is provided for all records that returns all property values as a tuple.

It is useful while reading few specific set of properties from a record object.

Record - deconstruct'

```
1. record_name reference_variable = new record_name(value1, value2);
2. var (variable1, variable2, …) = reference_variable;
```

Records - ToString()

A compiler-generated 'ToString()' is provided for all records that returns a string with all properties and values.

```
1. public record record_name(Properties_list)
2. {
3.    public override string ToString( ) //compiler-generated
4.    {
5.       //returns a string: Record_Name { Property1 = value1, Property2 =
   value2, … }
6.    }
7. }
```

You can override that compiler-generated 'ToString()' with 'override' keyword.

Record - ToString() - User defined

```
1. public record record_name(Properties_list)
2. {
3.    public override string ToString( ) //user-defined
4.    {
5.       //return any string
6.    }
7. }
```

Records - Constructor

A compiler-generated 'parameterized constructor' is provided for all records that initializes all property values.

You must invoke the compiler-generated constructor of the record with 'this' keyword, in case if you create your own constructor.

Record - User-Defined Constructor

```
1. public record_name(parameters): this(parameters) //invokes compiler-generated
   constructor
2. {
3.    Property = value;
4. }
```

Records - Inheritance

A record can inherit from another record.

```
1. public record Parent_record_name(Properties_list);
2. public record Child_record_name(Properties_list) : Parent_record_name;
```

- A record CAN inherit from another record.
- A record CAN'T inherit from another class.
- A class CAN'T inherit from another record.
- A record CAN implement (inherit) one or more interfaces.
- A record CAN be 'abstract' and 'sealed'.

Records - Sealed ToString()

The user-defined 'override ToString()' can be 'sealed', in order to prevent further overriding.

```
1. public record record_name(Properties_list)
2. {
3.    public override sealed string ToString( ) //user-defined
4.    {
5.       //return any string
6.    }
7. }
```

Record Structs

Record [or] Record class

```
record record_name(Properties_list);
```

- A record is a class internally (after compilation).
- All positional parameters of a record are init-only properties by default.

Readonly record struct

```
readonly record struct record_name(Properties_list);
```

- A readonly record struct is a 'struct' internally (after compilation).
- All positional parameters of a readonly record struct are init-only properties by default.

Record struct

```
record struct record_name(Properties_list);
```

- A record struct is a 'struct' internally (after compilation).
- All positional parameters of a record struct are read-write properties by default.

Command Line Arguments

Goal: Supply inputs from the command line / terminal to an application.

```
app.exe value1 value2
```

-- will be converted as array:

Code that receives arguments from command line / terminal

```
1. class class_name
2. {
3.    static void Main(string[ ] args)
4.    {
5.       //Use args
6.    }
7. }
```

Features:

- CLR converts all the command line arguments (space-separated values) as string array (string[ ]) only (in the same order).
- The Main method can't have additional arguments - other than
- string[ ] args.
- The parameter name 'args' isn't fixed. You can give it any other name.
- Top level statements have an implicit parameter called 'args' of string[ ] type, which contains the command line argument s received.

Improvements in Partial Methods in C# 9

- A partial method CAN have any return type (not-only 'void') in C# 9.
- A partial method CAN have any access modifier in C# 9.
- A partial method CAN have 'out' parameters in C# 9.
- Partial methods must have a definition in any one of the parts of the same partial class.

Partial Method in C# 9

```
public partial return_type Method_name(Parameters_list);
```

Static Anonymous Functions

- A static anonymous function is an anonymous method or lambda expression, prefixed with 'static' keyword.
- It CAN'T access the state (local variables, parameters, 'this' keyword and 'base' keyword) of the enclosing method; and also CAN'T access instance members of enclosing type.
- It CAN access static members and constants of enclosing type.

Static anonymous function (anonymous method)

```
1. static delegate (Parameters_list)
2. {
3.    //can't access locals, parameters, instance members
4.    //can access static members and constants
5. }
```

Static anonymous function (lambda expression)

```
1. static (Parameters_list) =>
2. {
3.    //can't access locals, parameters, instance members
4.    //can access static members and constants
5. }
```

Return Type of Lambda Functions

A lambda expression (or lambda function) can have a return type before the list of parenthesized parameters.

Useful when you return a value of any one of two or more types.

Lambda Function Return Type in C# 10

```
return_type (Parameters_list) => return_value;
```

Constant Interpolated Strings

Constant strings may be initialized using 'string interpolation' i.e. with ${ }, if all the placeholders are constant strings.

Eg: Useful when you are creating global API URLs.

Constant Interpolated Strings

```
const data_type variable_name = $"{constant_string}";
```

Interface Default Methods

Default methods are methods in interfaces with concrete implementation.

These methods are accessible through a reference variable of the interface type.

Interface Default Methods

```
1. interface interface_name
2. {
3.    access_modifier return_type method_name(parameters)
4.    {
5.       //method body
6.    }
7. }
```

Access Modifiers on Interface Methods

Interface methods can have any access modifiers including private, protected, internal, protected internal, private protected and public

Non-public interface methods can be either implemented as public or explicitly (to preserve the same access modifier).

Interface methods with access modifier

```
1. interface interface_name
2. {
3.    access_modifier return_type method_name(parameters);
4. }
```

Interface Private Methods

Private interface methods must have method body.

Private interface method

```
1. interface interface_name
2. {
3.    private return_type method_name(parameters)
4.    {
5.       //method body
6.    }
7. }
```

Interface Static Methods

Static methods are allowed with concrete implement in interface.

Interface static methods can be called through the interface name.

Interface Static Methods

```
1. interface interface_name
2. {
3.    access_modifier static return_type method_name(parameters)
4.    {
5.       //method body
6.    }
7. }
8.
9. interface_name.method_name(arguments); //calling the interface static method
```

**********--THE END--**********