**Section Cheat Sheet (PPT) 1**

#### 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

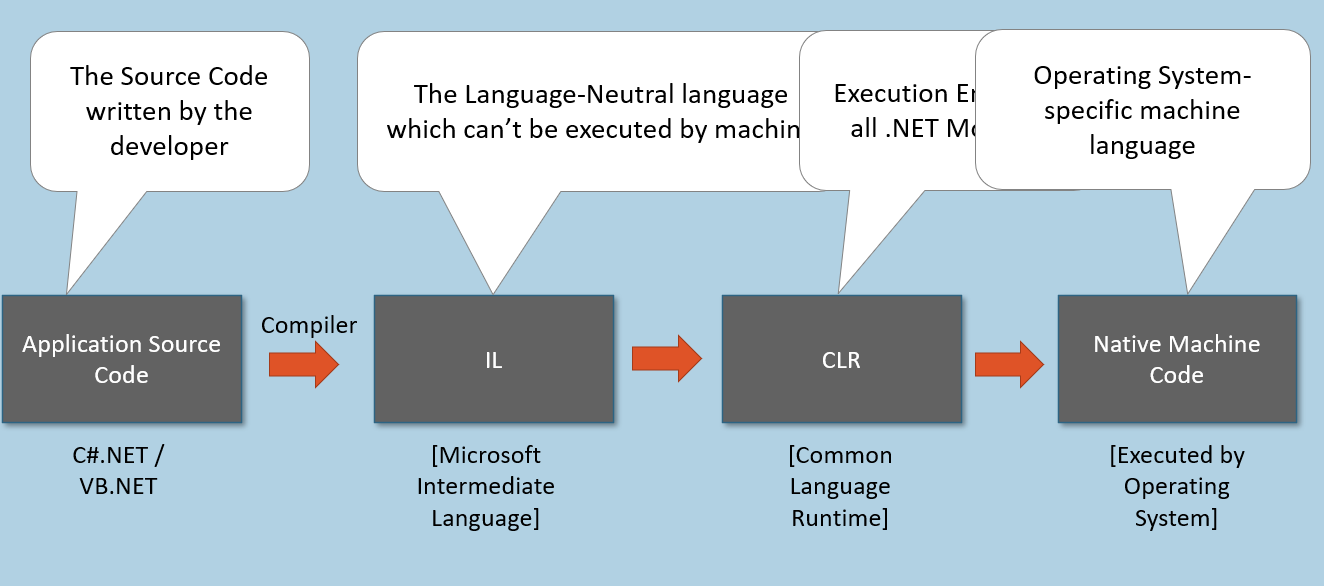
**C#.Net**

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.

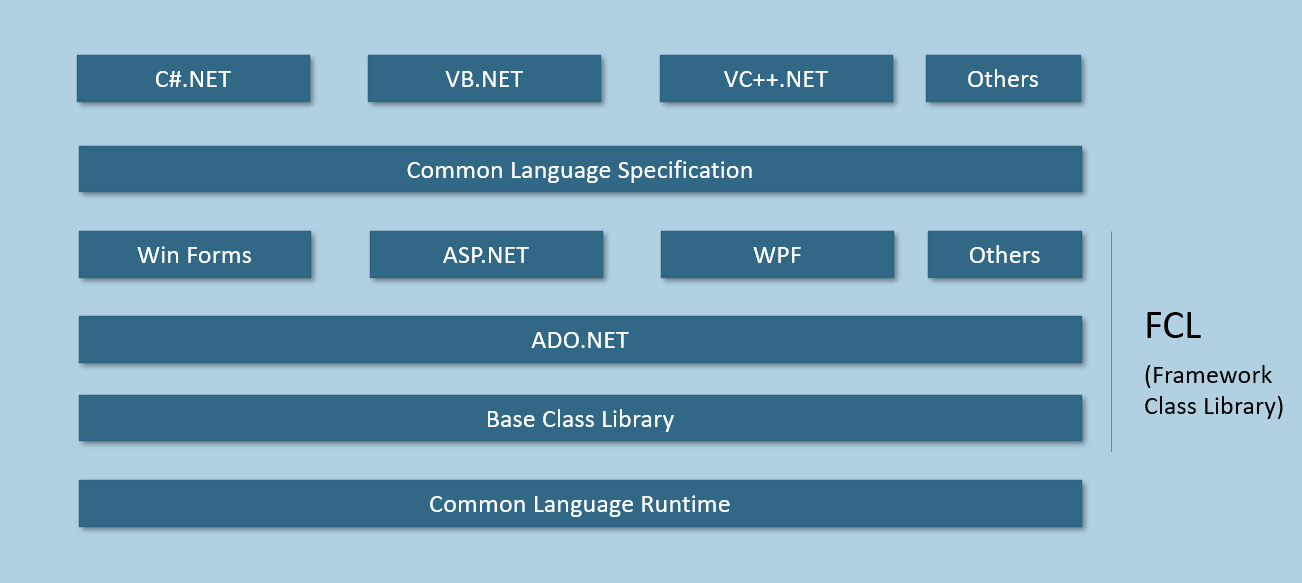
**Security Manager:**

* 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

#### IMG_257

**.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**

#### IMG_259

**.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'.

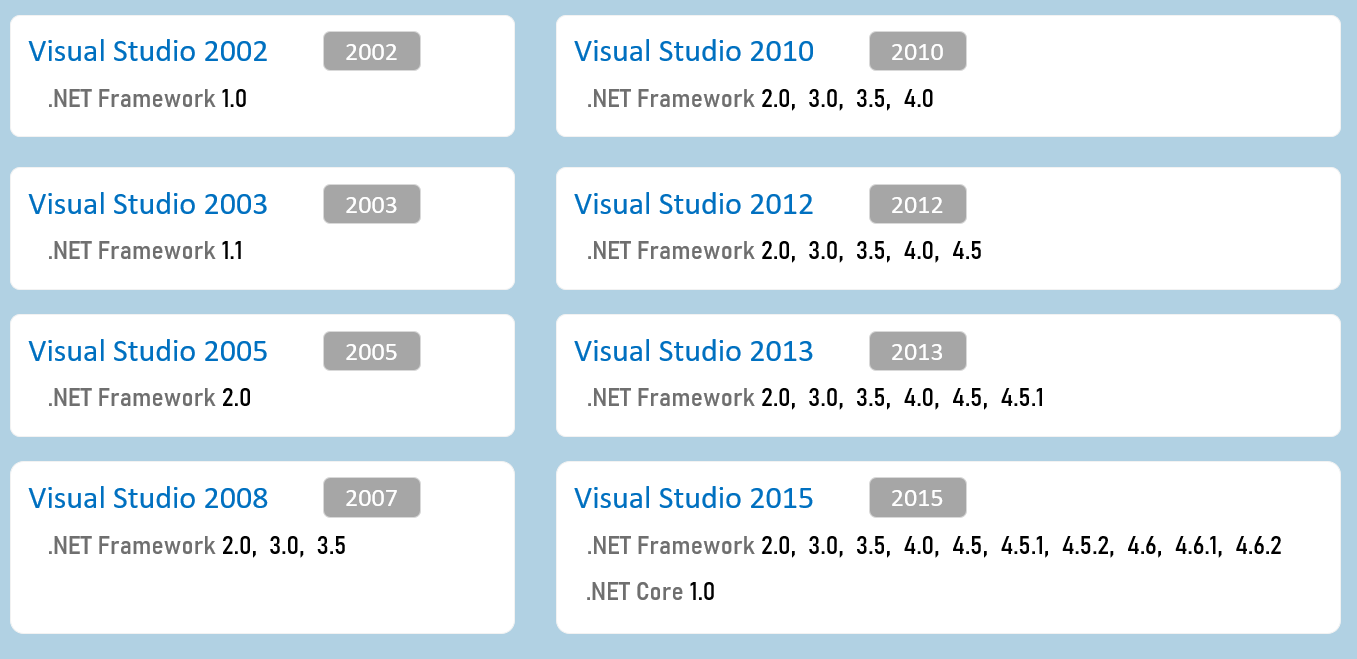
**Features:**

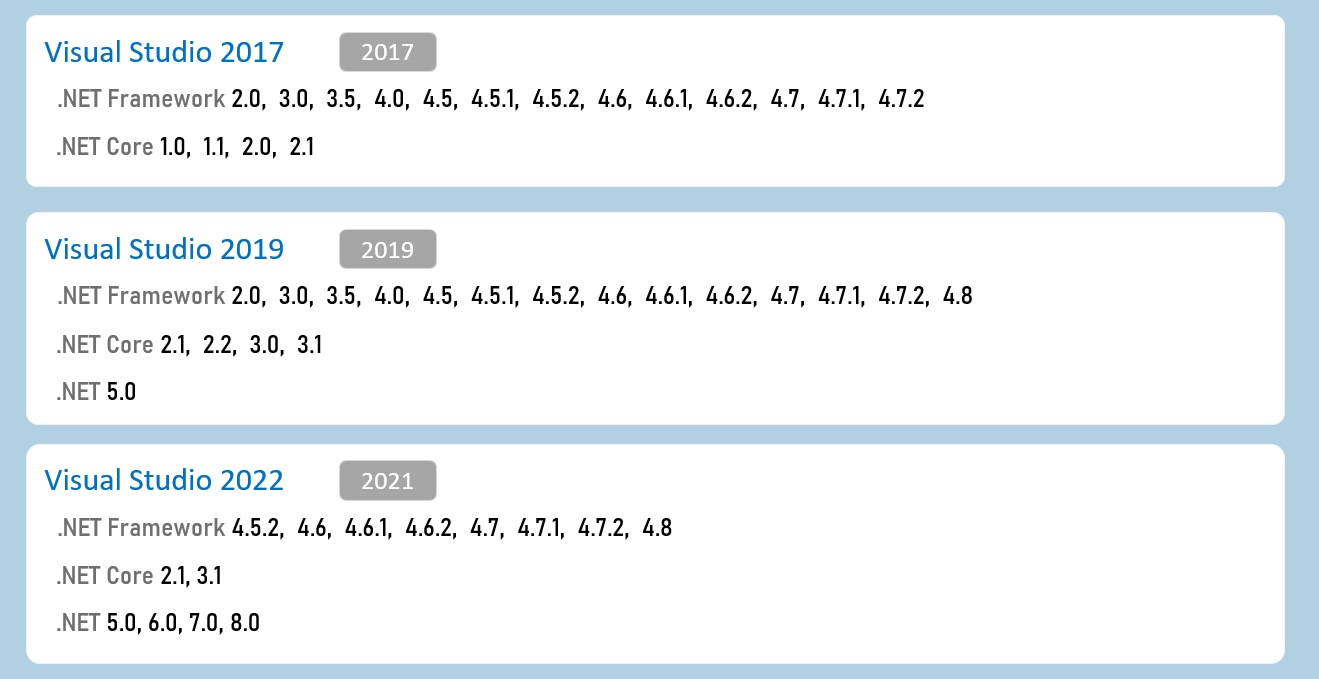
Intelli-sense (code completion)

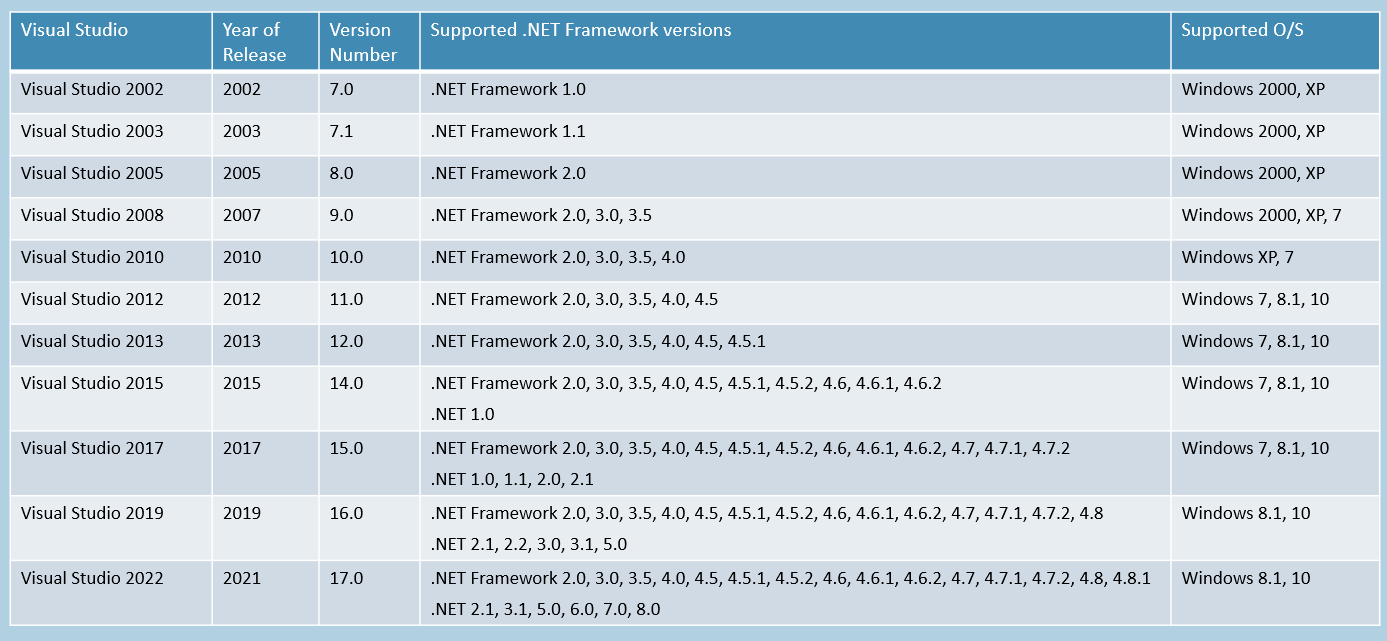
Syntax highlighting

Debugger

GUI Designer

**Versions of Visual Studio**





**Section Cheat Sheet (PPT)-2**

#### 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.

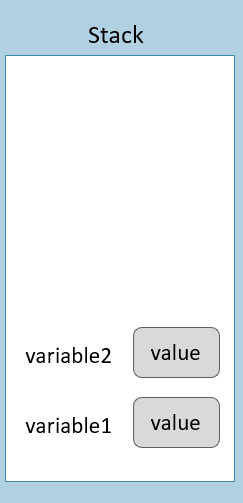
The variables must be declared before its usage.

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

• 179769313486232000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 (294 0's)

to 1797693134862320000000000000000000000000000000000000000000000000000000000000000000000000000000 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 (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 Operator
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**

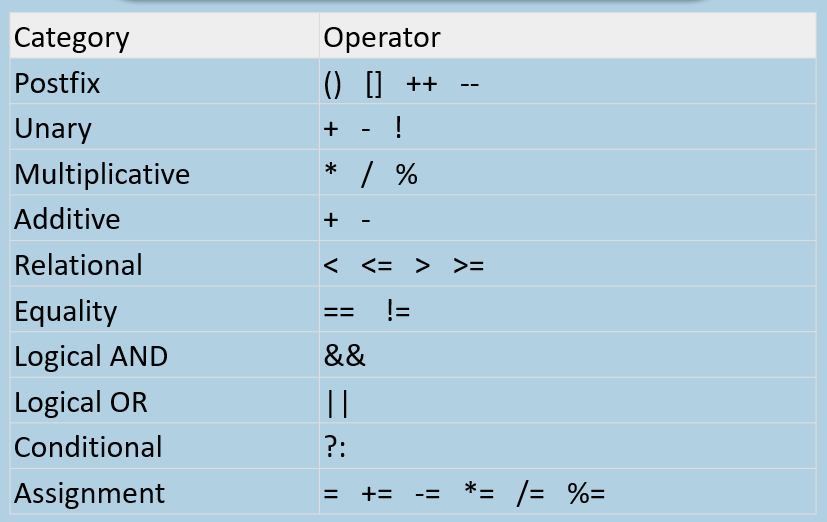
It evaluates the given Boolean value;

Returns first expression (consequent) if true;

Returns second expression (alternative) if false.

? :   (condition)? consequent : alternative

#### Operator Precedence



#### Control Statements

**Conditional Control Statements:**

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

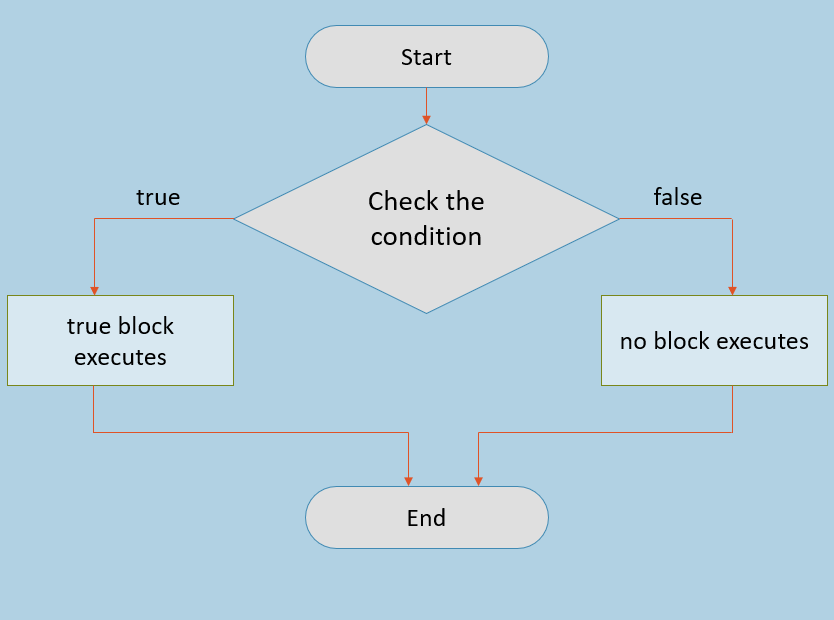
**Looping Control Statements:**

* while
* do-While
* For

**Jumping Control Statements:**

* goto
* break
* continue

#### simple-if



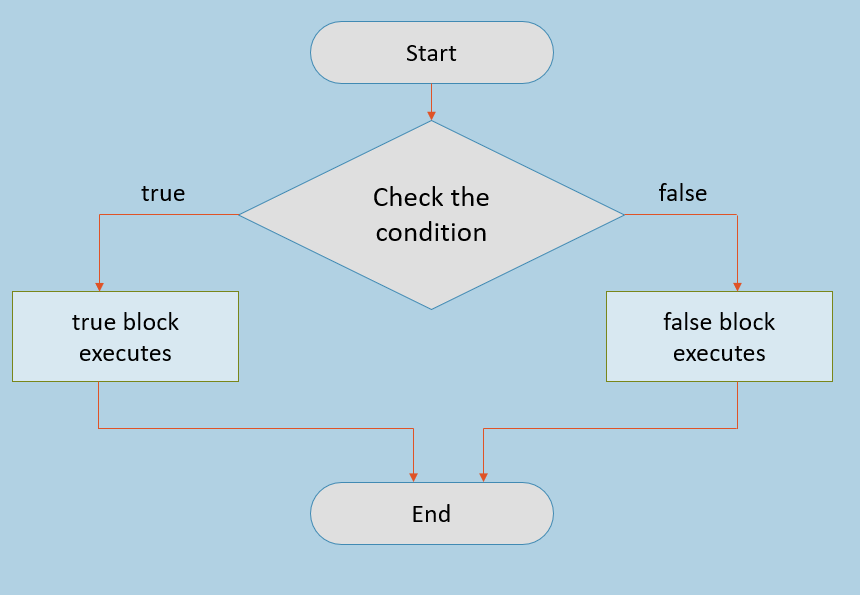
**Simple-if - Syntax**

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

**Simple-if - Example**

1. if (x < 10)
2. {
3. System.Console.WriteLine("x is smaller than 10");
4. }

#### if-else



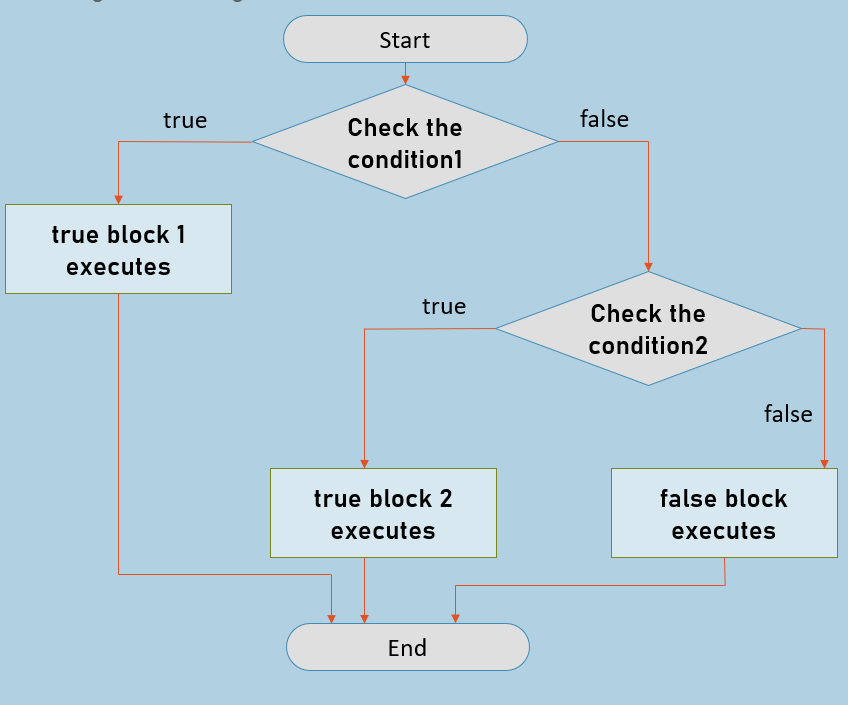
**if - else - Syntax**

1. if (condition)
2. {
3. true block here
4. }
5. else
6. {
7. false block here
8. }

**if-else - Example**

1. if (x > 10)
2. {
3. System.Console.WriteLine("x is larger than 10");
4. }
5. else
6. {
7. System.Console.WriteLine("x is smaller than or equal to 10");
8. }

#### else-if



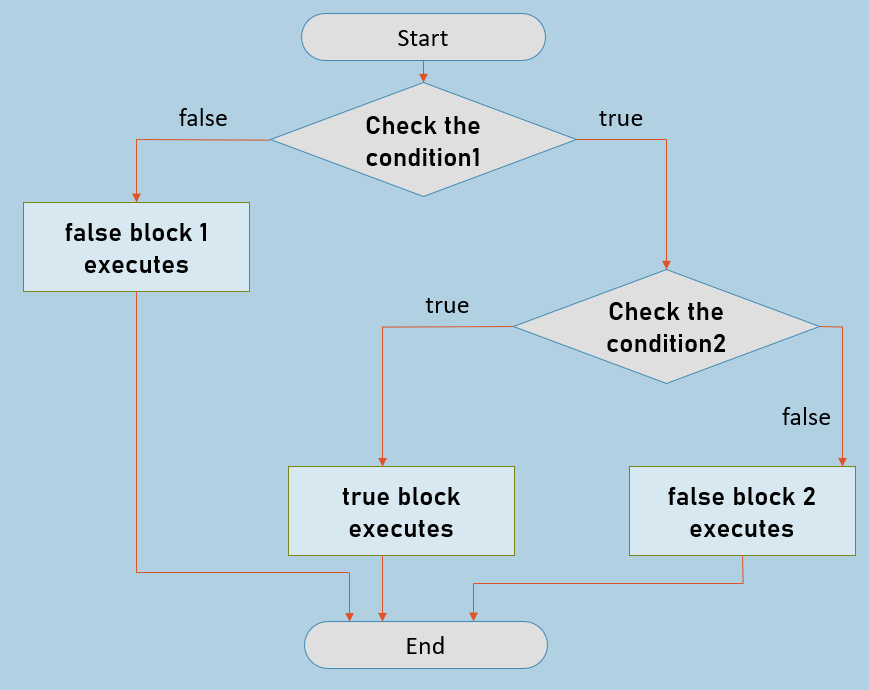
**else-if - Syntax**

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

**else If - Example**

1. if (a > 10)
2. {
3. System.Console.WriteLine("a is greater than 10");
4. }
5. else if (a < 10)
6. {
7. System.Console.WriteLine("a is less than 10");
8. }
9. else
10. {
11. System.Console.WriteLine("a is equal to 10");
12. }

#### nested-if



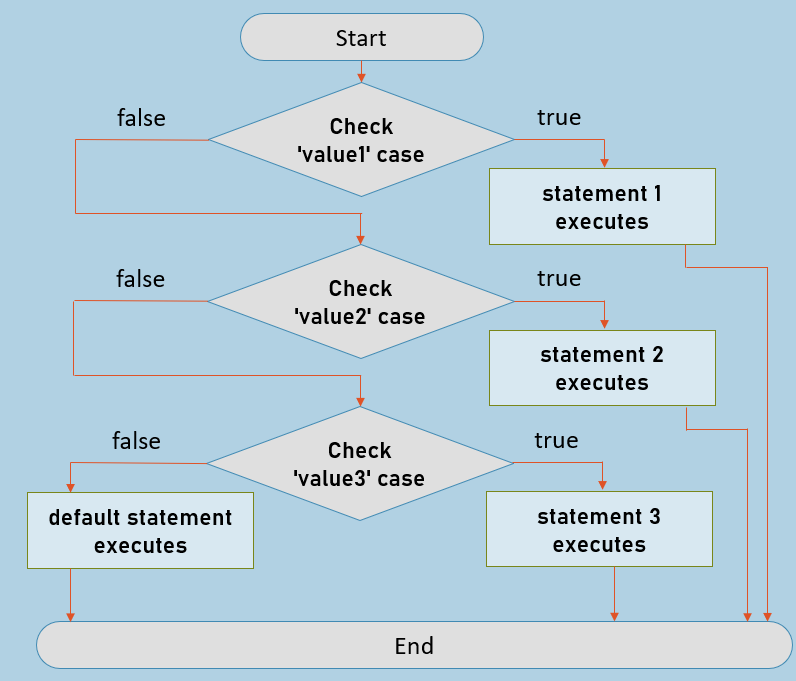
**nested-if - Syntax**

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**

1. if (a >= 10)
2. {
3. if (a > 10)
4. {
5. System.Console.WriteLine("a is greater than 10");
6. }
7. else
8. {
9. System.Console.WriteLine("a is equal to 10");
10. }
11. }
12. else
13. {
14. System.Console.WriteLine("a is less than 10");
15. }

#### switch-case



**switch-case - Syntax**

1. switch (variable)
2. {
3. case value1: statement1; break;
4. case value2: statement2; break;
5. case value3: statement3; break;
6. …
7. default: statement; break;
8. }

**switch-case - Example**

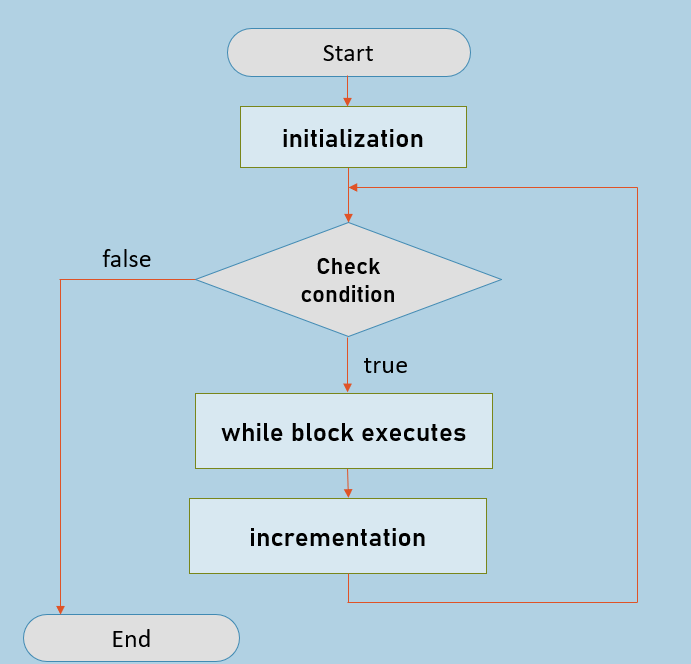
1. switch ( x )
2. {
3. case 1: System.Console.WriteLine("one"); break;
4. case 2: System.Console.WriteLine("two"); break;
5. case 3: System.Console.WriteLine("three"); break;
6. default: System.Console.WriteLine("none"); break;
7. }

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**

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

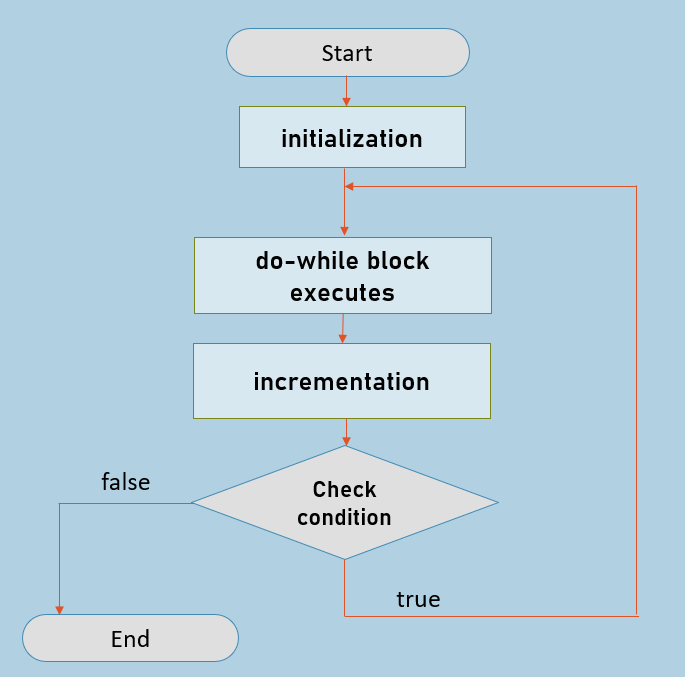
**while - Example**

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 );

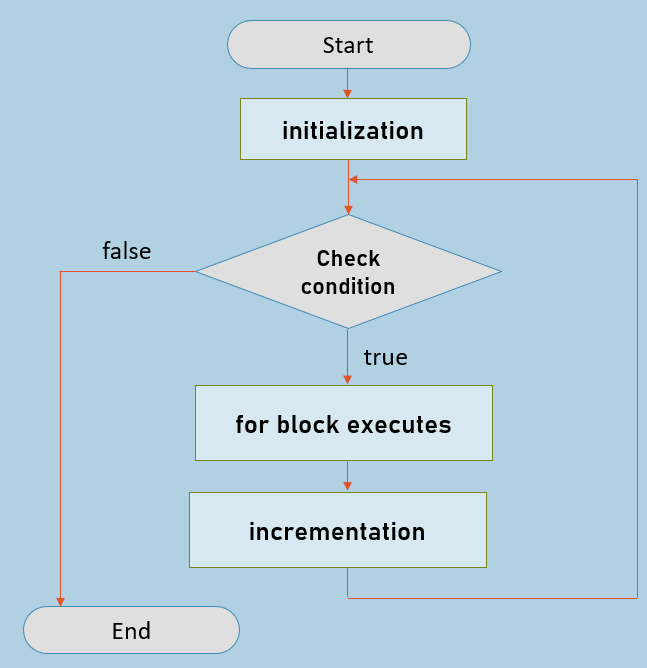
Used to execute a set of statements, as long as the condition is TRUE.›Once the condition is false, it will exit from the while loop.

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. }

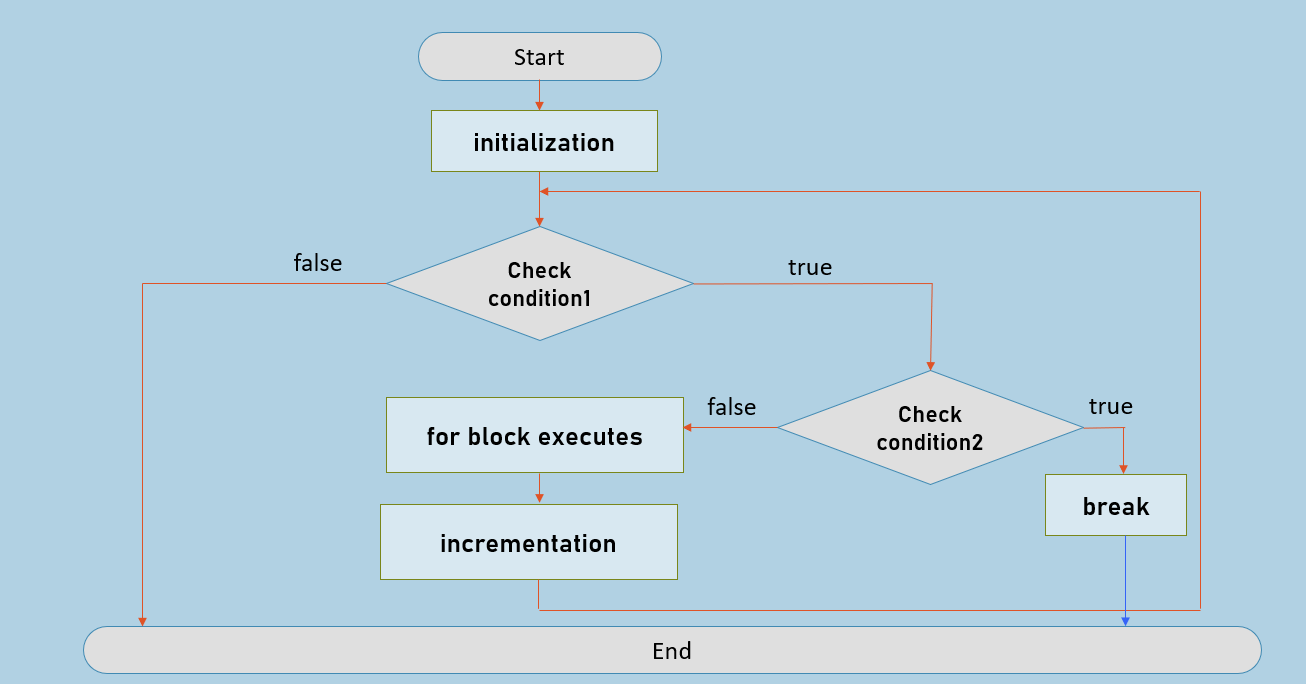
Used to execute a set of statements, as long as the condition is TRUE.

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



**break - Syntax**

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

**break - Example**

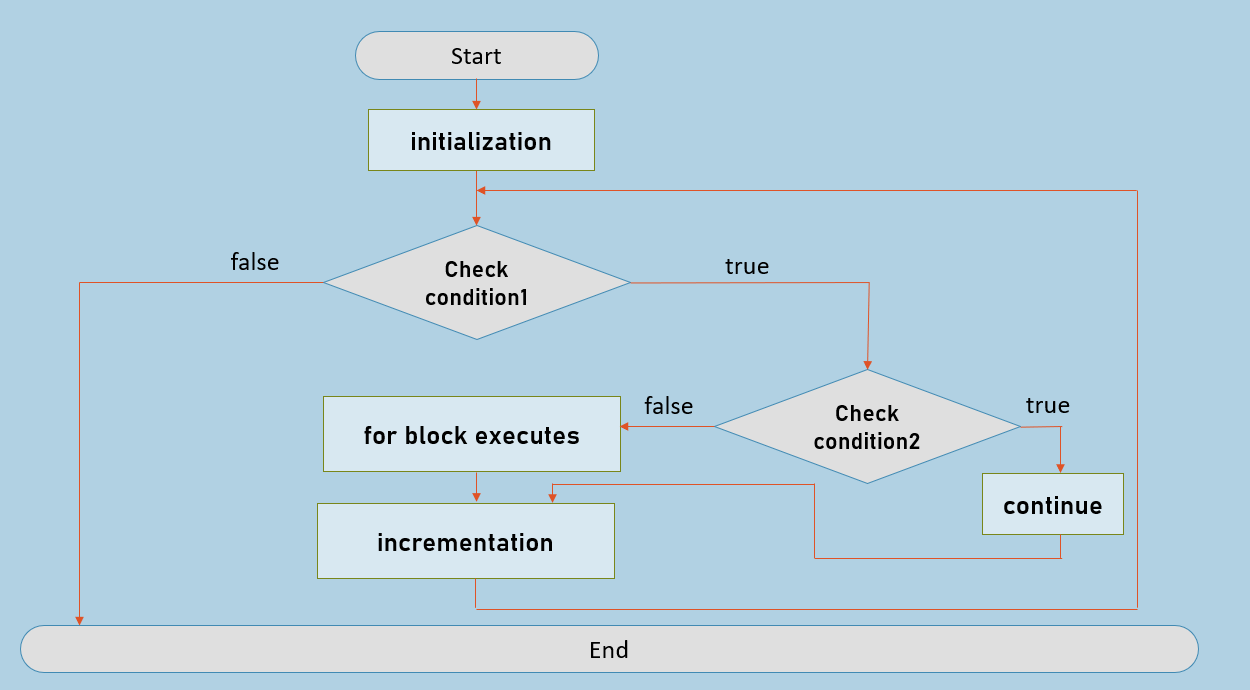
1. for (int i = 0; i <= 10; i++)
2. {
3. if (i == 6)
4. {
5. break;
6. }
7. System.Console.WriteLine(i);
8. }
9. //Output: 0, 1, 2, 3, 4, 5

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



**continue - Syntax**

1. for (initialization; condition1; incrementation)
2. {
3. if (condition2)
4. {
5. continue;
6. }
7. for block code here
8. }

**continue - Example**

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

1. for (initialization; condition1; incrementation)
2. {
3. for (initialization; condition2; incrementation)
4. {
5. inner-loop code here
6. }
7. outer-loop code here
8. }

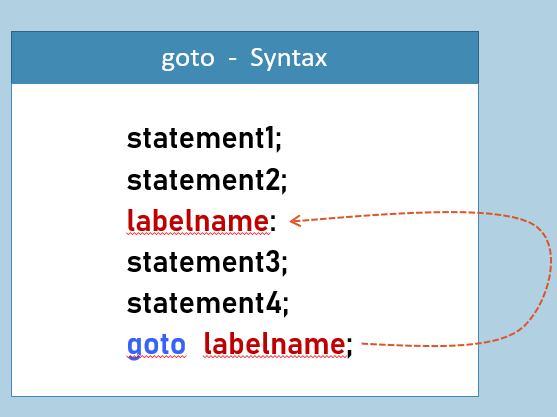
**nested for - Example**

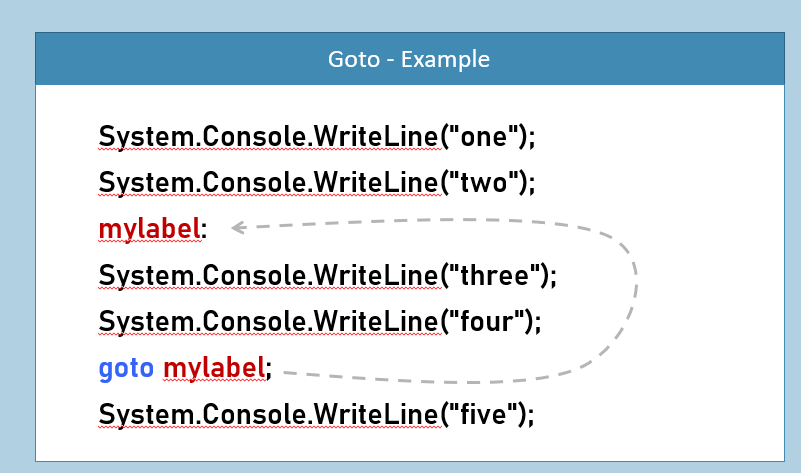
1. for (int i = 0; i < 5; i++)
2. {
3. for (int j = 0; j < 5; j++)
4. {
5. System.Console.WriteLine( j );
6. }
7. }
9. //Output: 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4,

#### ]

#### 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.

**Section Cheat Sheet (PPT) --4**

#### 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

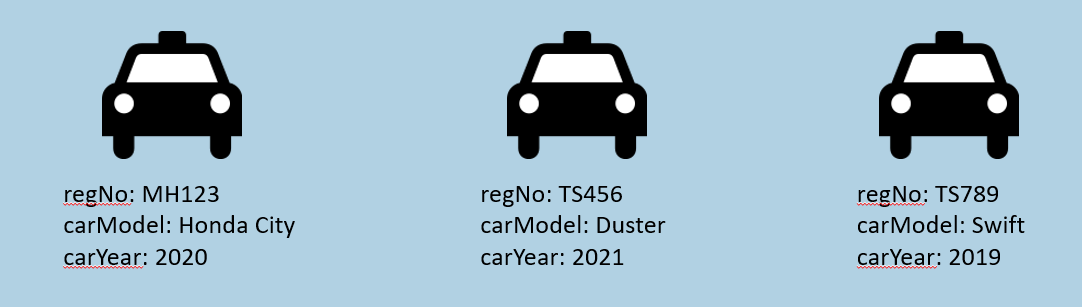
Object is a small unit (entity) in the program that represents a real-world person or thing.

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:

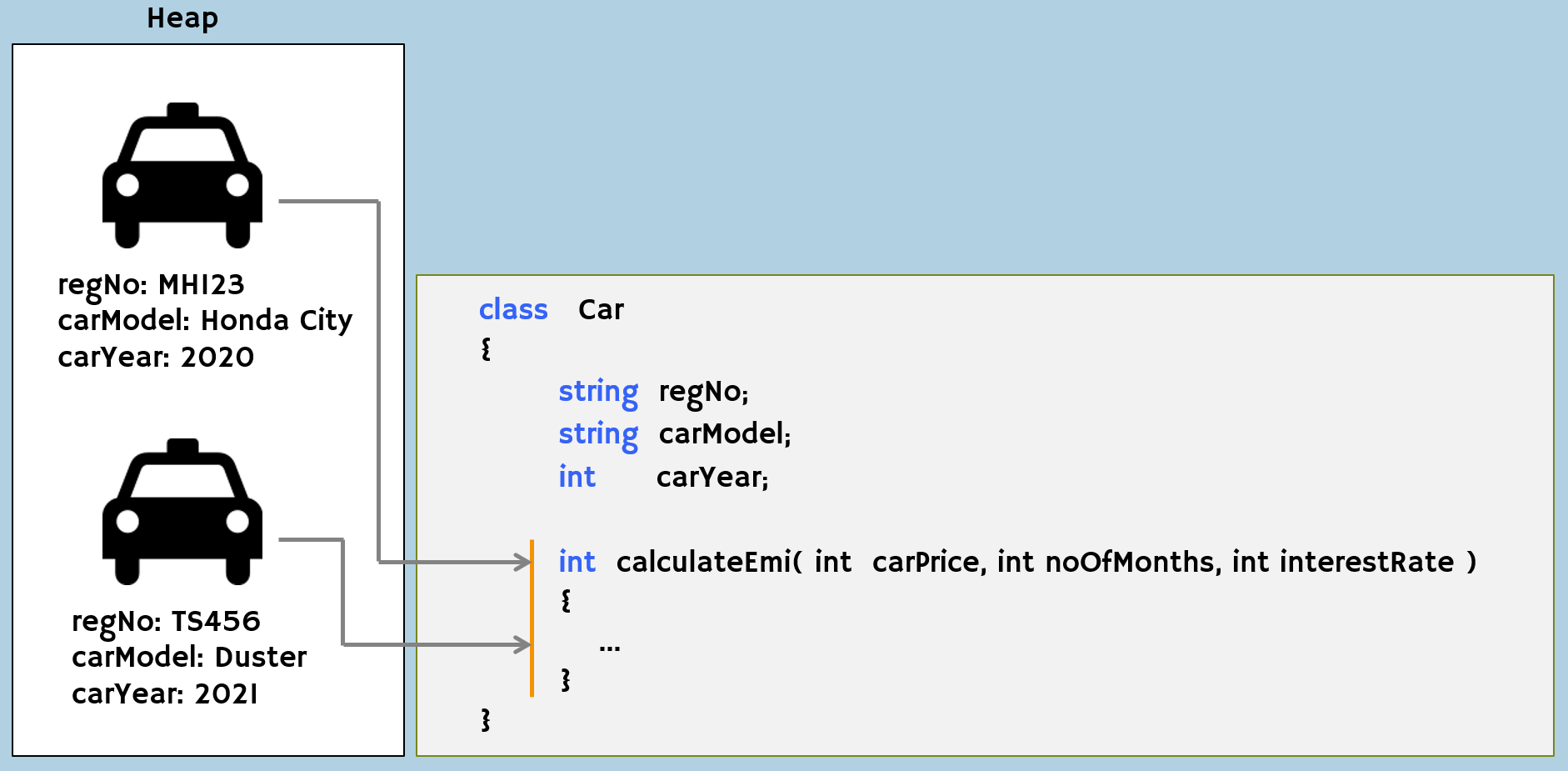
1. class Car
2. {
3. int calculateEmi( int carPrice, int noOfMonths, int interestRate )
4. {
5. //do calculation here
6. return (emi);
7. }
8. }

#### Object & Class Association

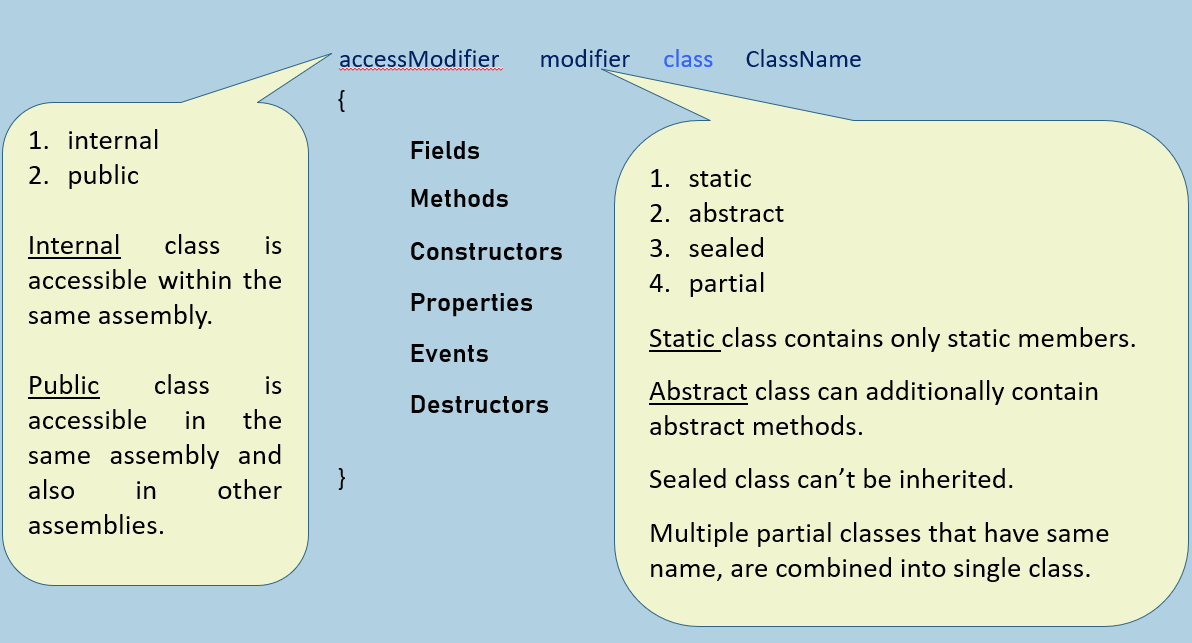
Object stores fields.

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

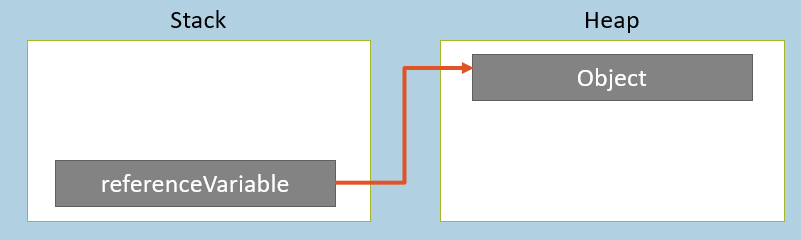
Class declares list of fields; defines list of methods.



**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**

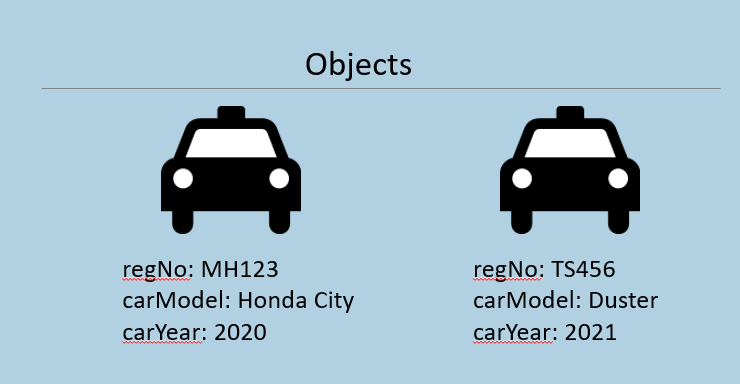
1. Object is a programmatic representation of a person or thing.
2. All objects are created based on classes; stored in 'heap'.
3. For each application execution, a new heap will be created (and only one).
4. All reference variables (local variables of methods) are stored in stack. For each method call,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

**Section Cheat Sheet (PPT)--5**

#### Fields

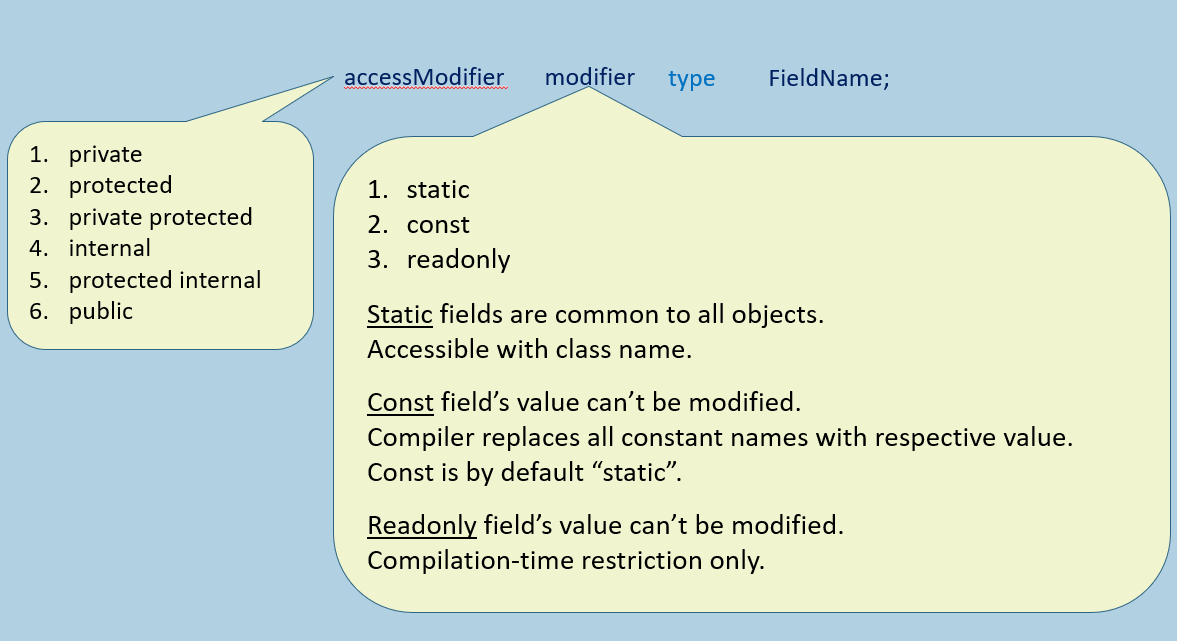
Variables that are declared in the class; stored in the objects.

Isolated for each object.



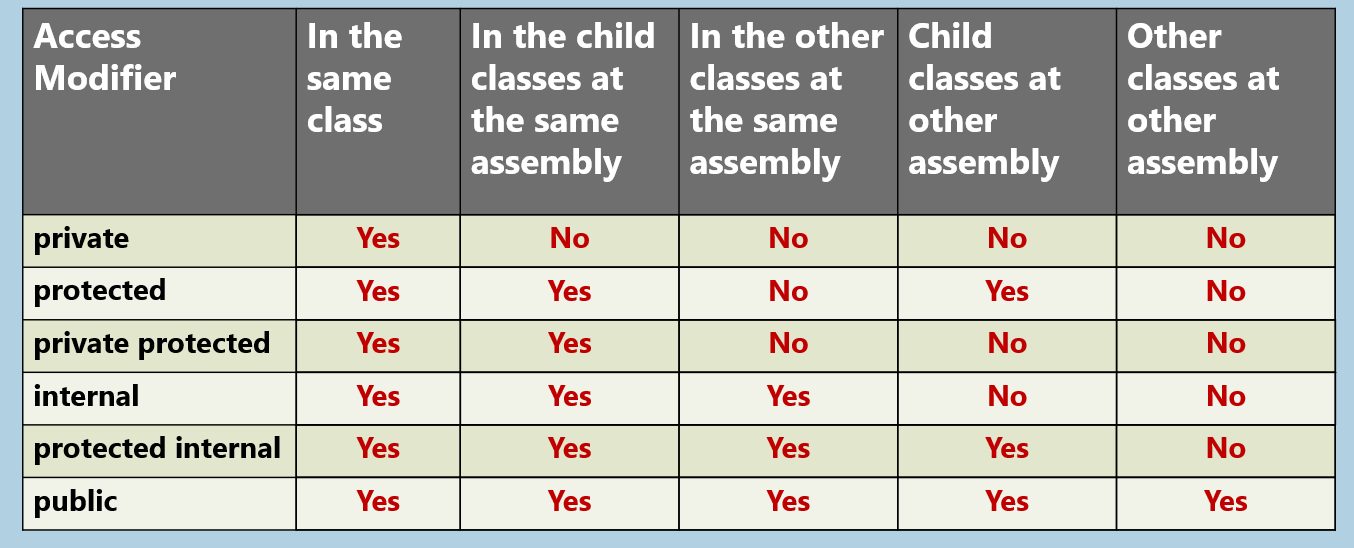
1. class Car
2. {
3. string regNo;
4. string carModel;
5. int carYear;
6. }

**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.



**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.

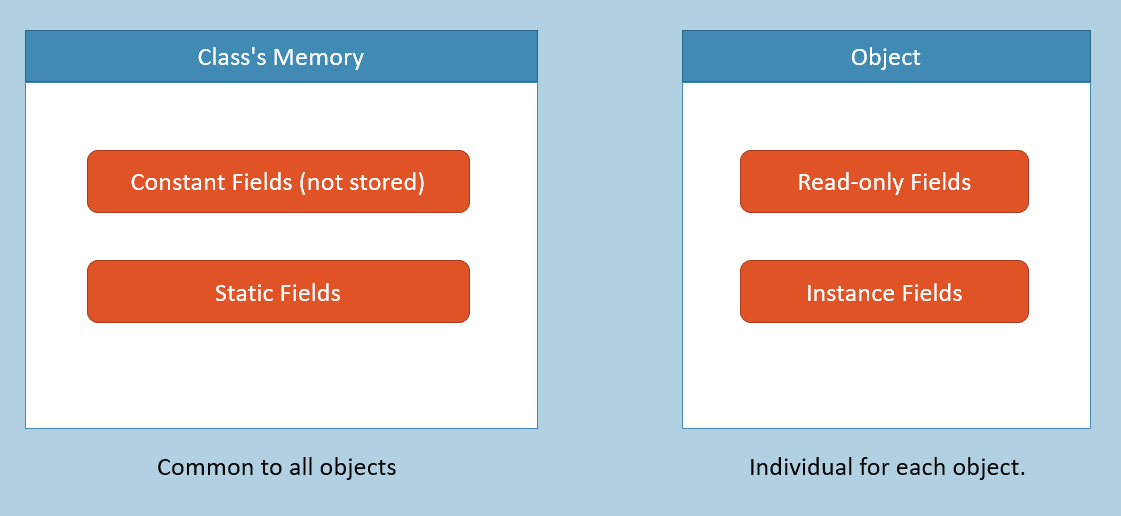
**Static Fields:**Allocated only once for the entire program; i.e. when the class is used for the first time while executing the program.

#### Constant Fields

1. Constant Fields are like static fields, that are common to all objects of the class.
2. We can't change the value of constant field.
3. Constant Fields are accessible with class name [not with object].
4. Constant Fields are not stored in the object; will not be stored anywhere.
5. Constant Fields will be replaced with its value, while compilation; so it will not be stored anywhere in memory.
6. Constant Fields must be initialized, in line with declaration (with a literal value only).
7. Constants can also be declared as 'local constants' (in a method).
8. 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'.

**Section Cheat Sheet (PPT)--6**

#### Methods

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

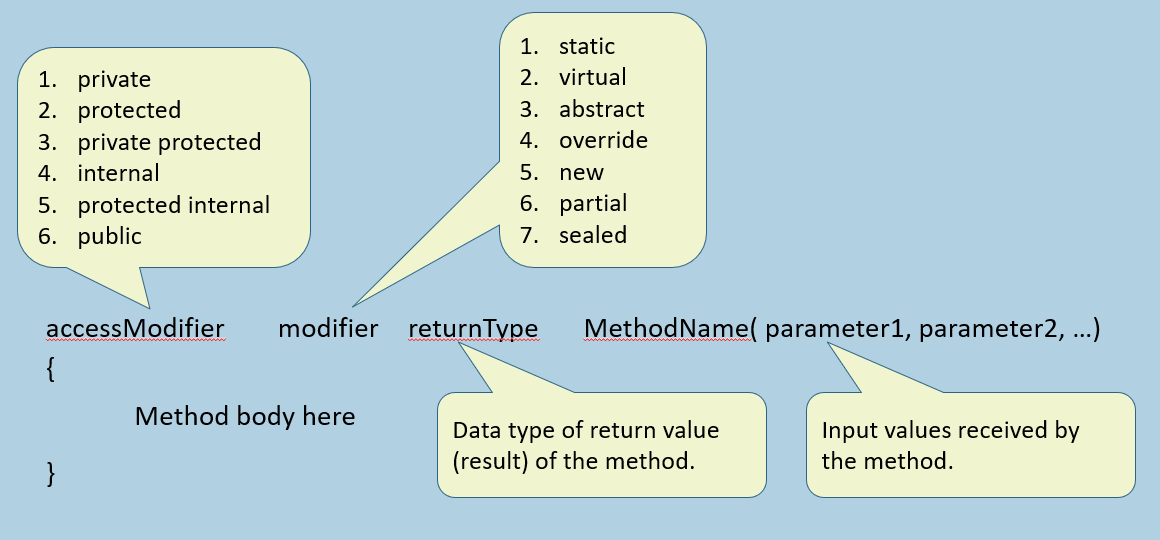
Methods are parts of the class.

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

Eg:

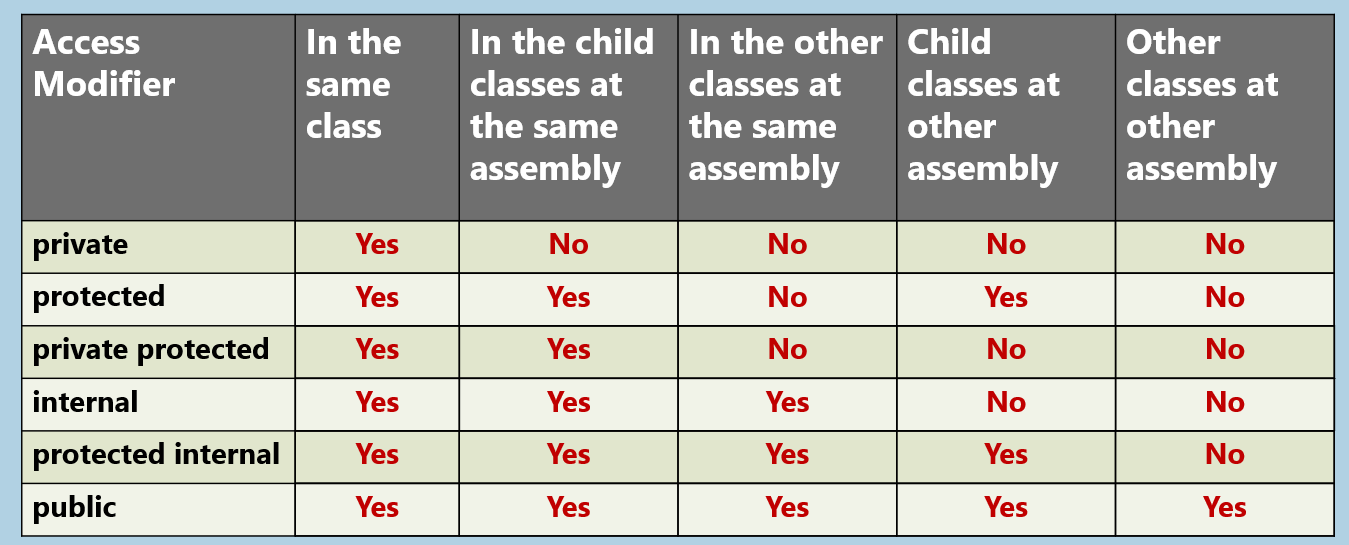
1. class Car
2. {
3. int calculateEmi( int carPrice, int noOfMonths, int interestRate )
4. {
5. //do calculation here
6. return (emi);
7. }
8. }

**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.



#### Encapsulation

Encapsulation is a concept of:

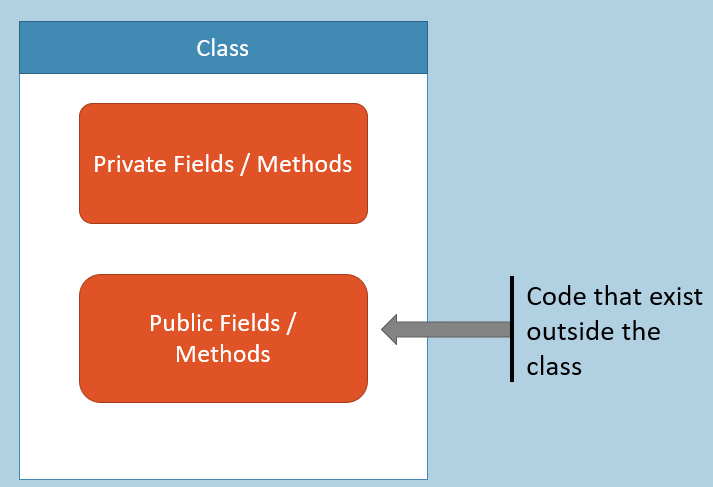
* Bundling the data (fields) and operations (methods) that manipulate the data together.
* ides internal implementation details of an object and provide a essential members to interacting with them.

Benefits:

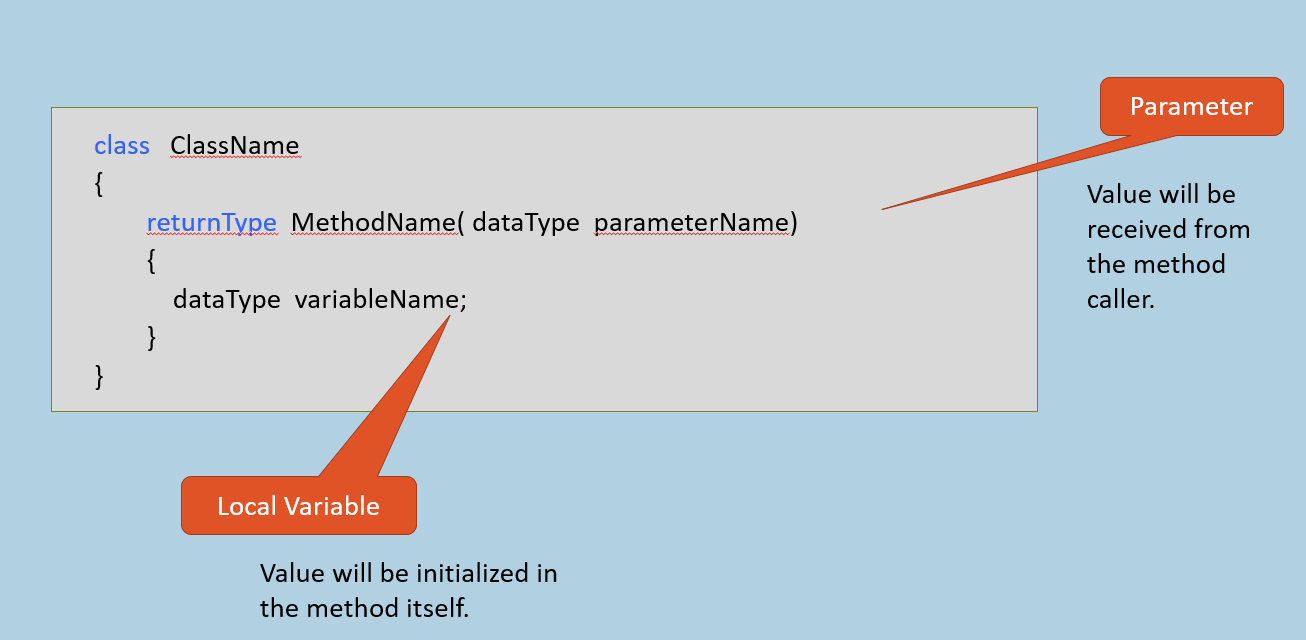
1. Modularity
2. Hiding implementation details
3. Data Integrity

Implemented using:

1. Private fields &
2. Public properties or public 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.

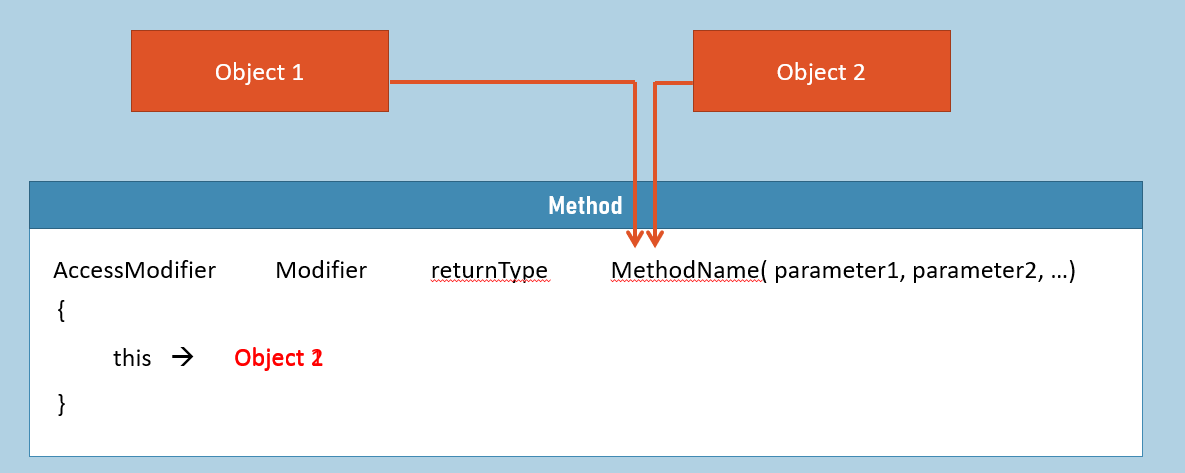
Local variables are stored in the same stack, just like parameters.

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.

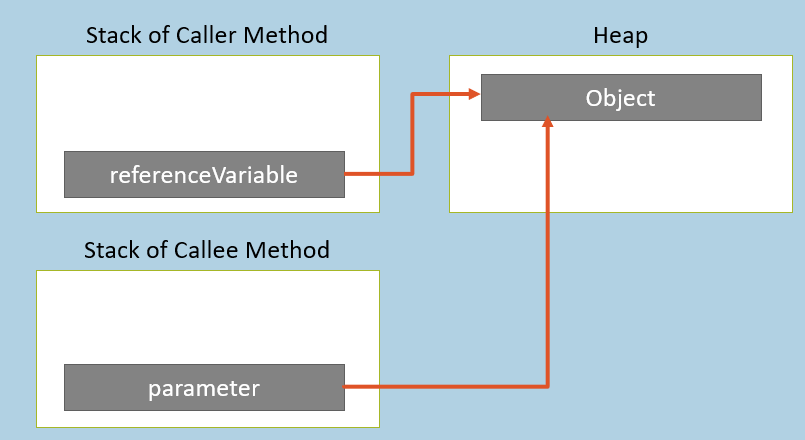
**Static Methods:**Can't access instance methods; but can access static methods only.

**"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.

1. accessModifier   modifier    returnType  MethodName( parameter1 )
2. {
3. Method body here
4. }

#### 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.

1. MethodName( )
2. MethodName( int )
3. MethodName( string )
4. MethodName( int, string )
5. MethodName( string , int)
6. MethodName( string , string, int)
7. etc.

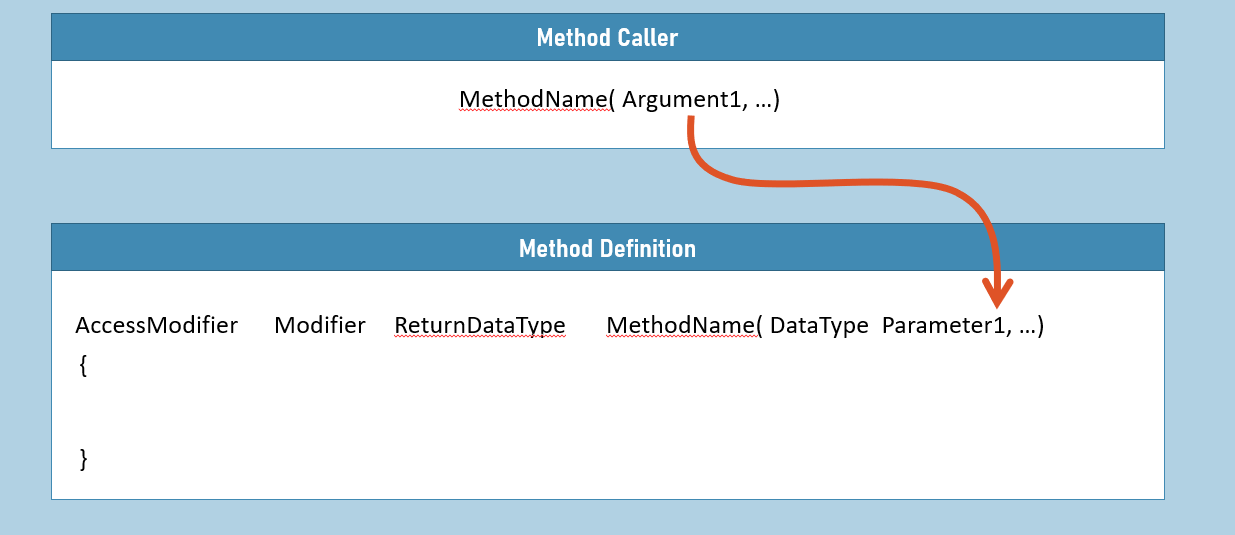
#### Parameter Modifiers

Specifies how the parameter receives a value.

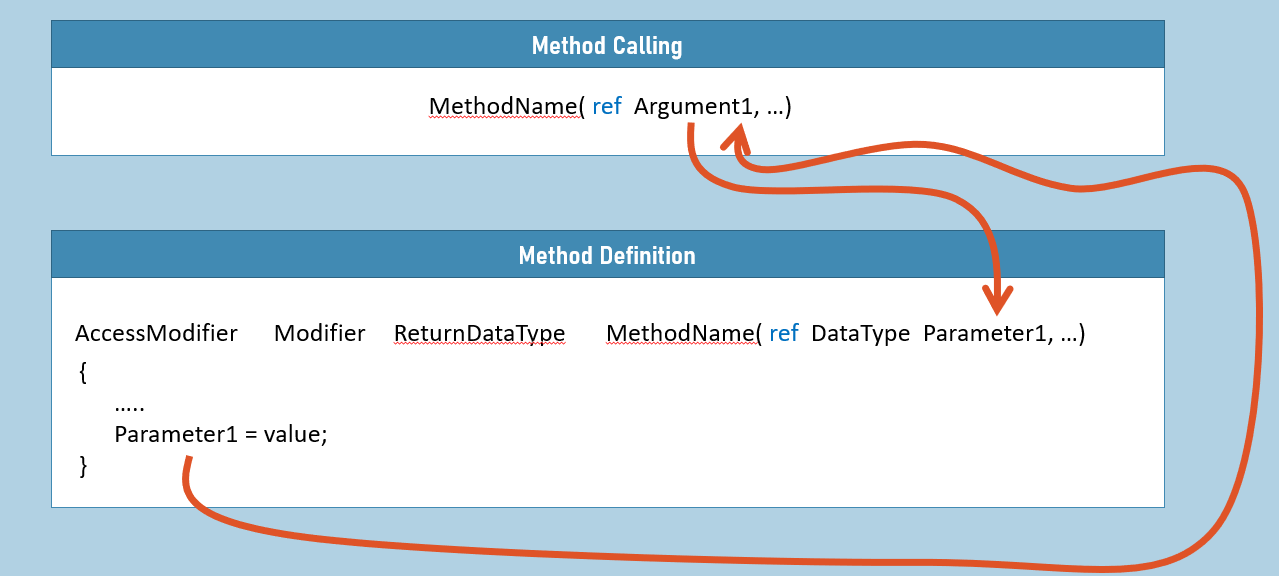
* Default [No keyword]
* ref
* out
* in
* params

#### Parameter Modifiers (default)

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



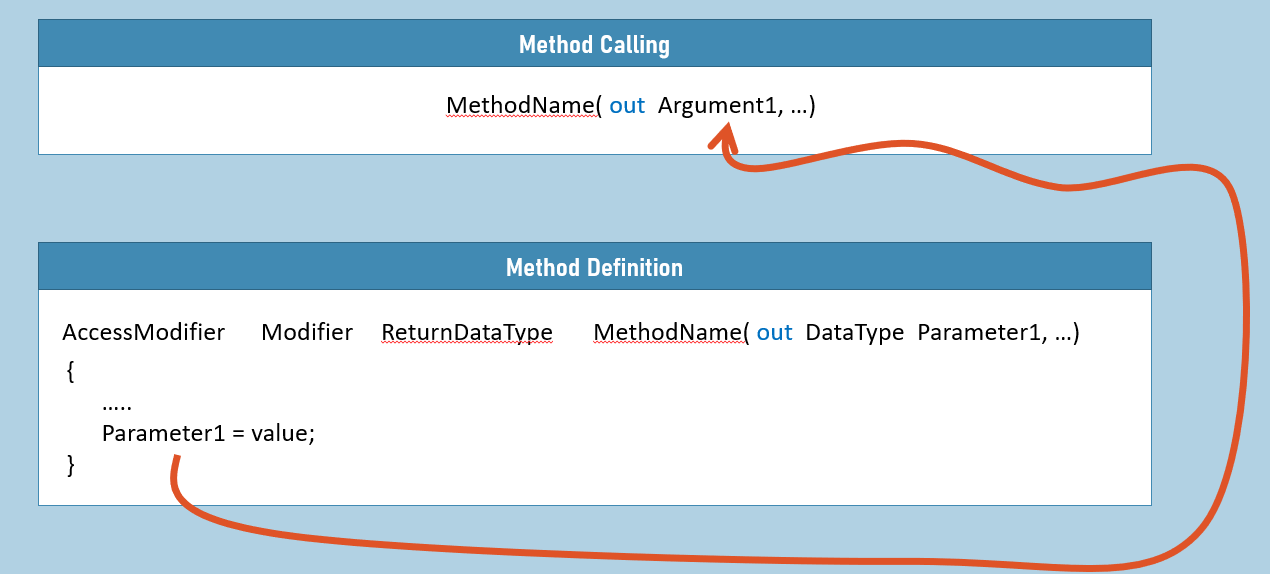
#### Parameter Modifiers (ref)

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.

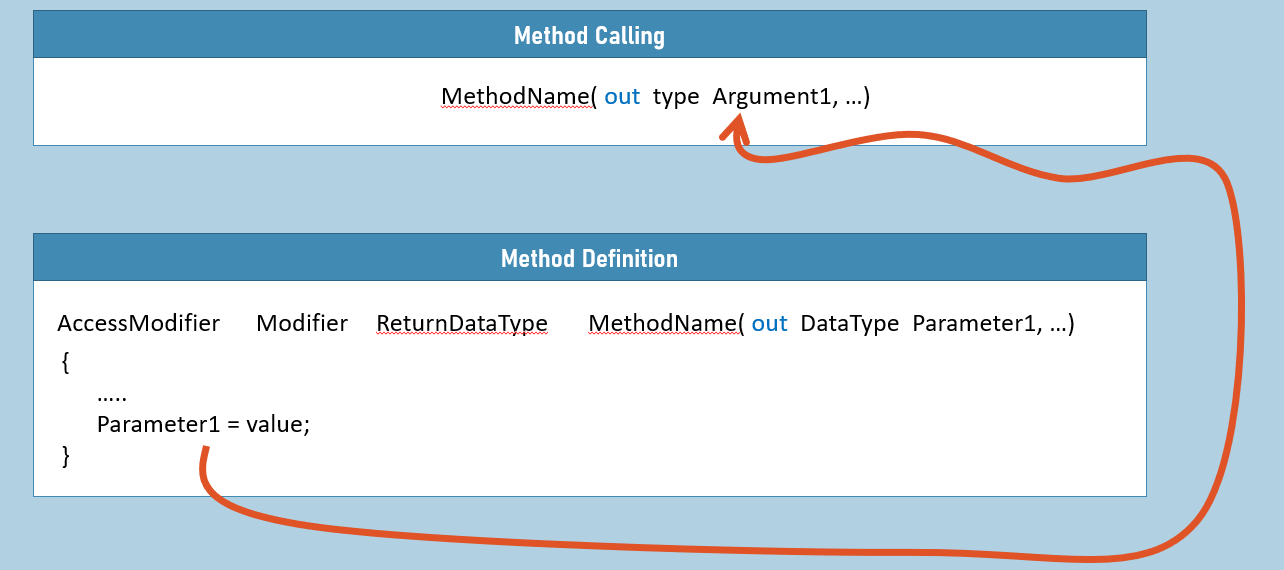
The Argument must be a variable; The Argument can be un-initialized.



#### 'out' variable declaration

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

New feature in C# 7.0.

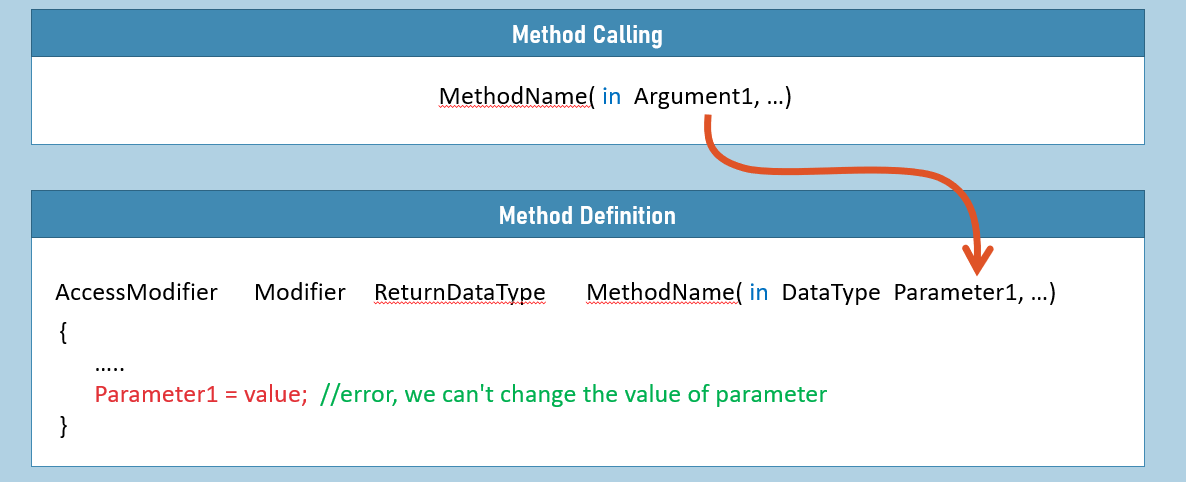


#### Parameter Modifiers (in)

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.

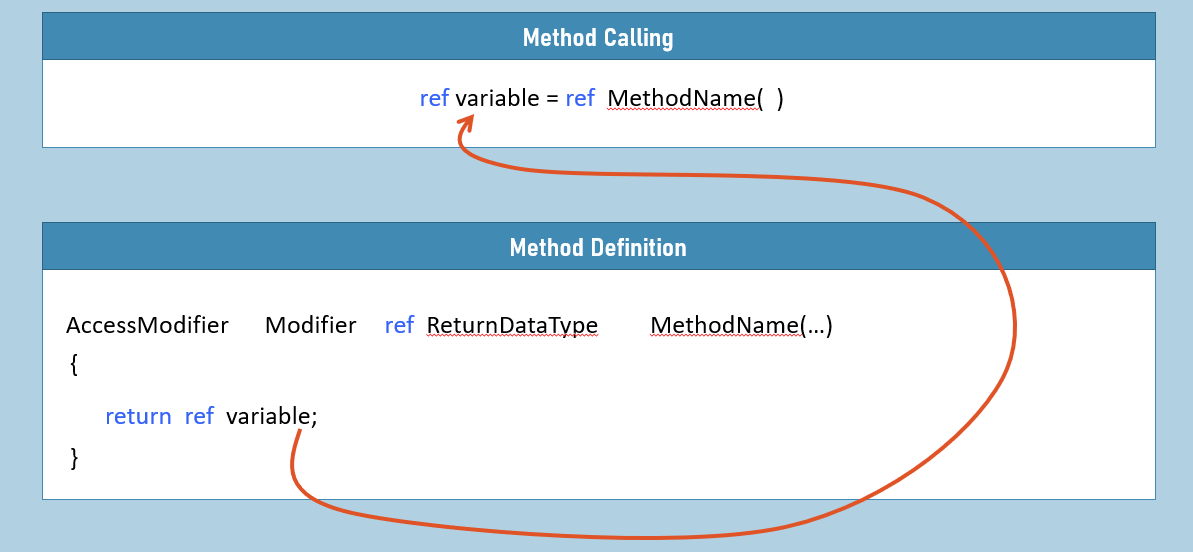
New feature of C# 7.2.



#### ref returns

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

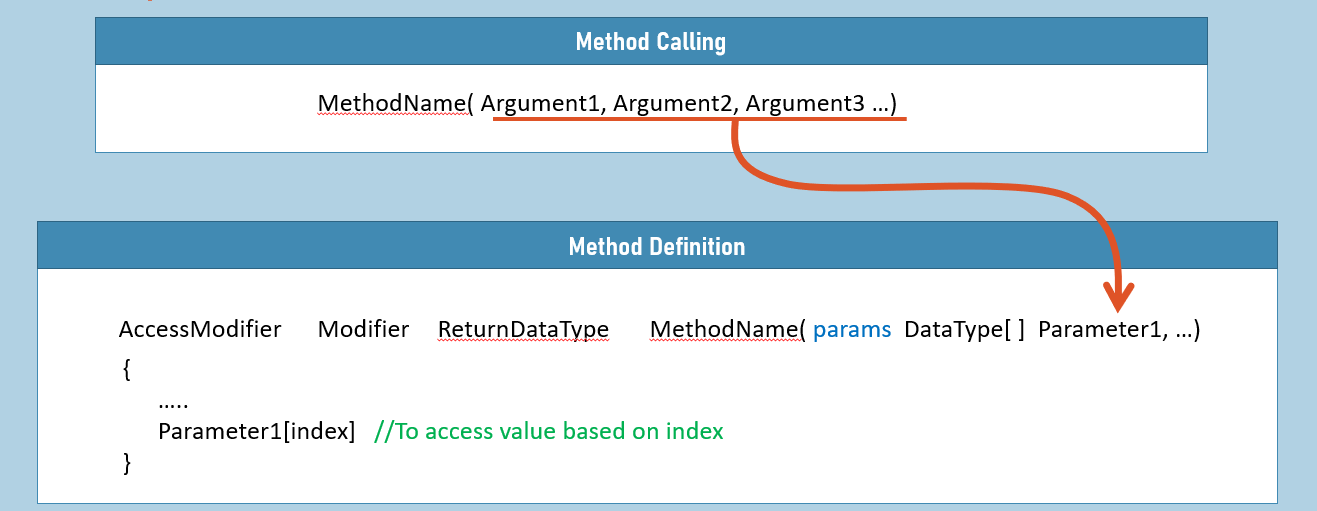
New feature in C# 7.3.



#### 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.



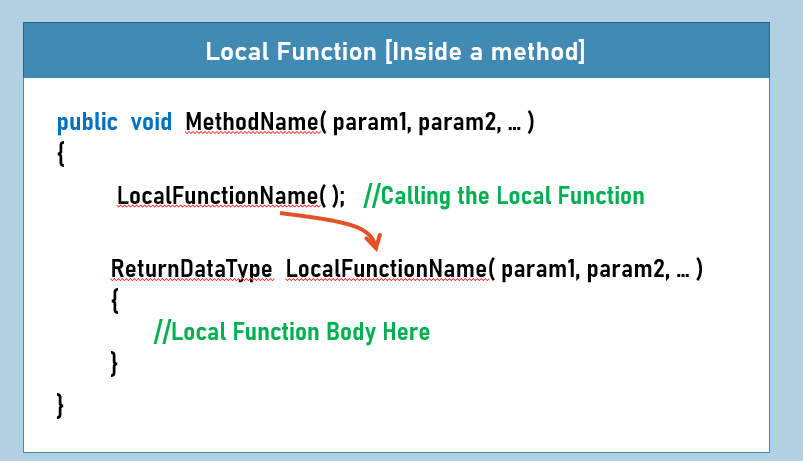
#### Local Functions

"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.

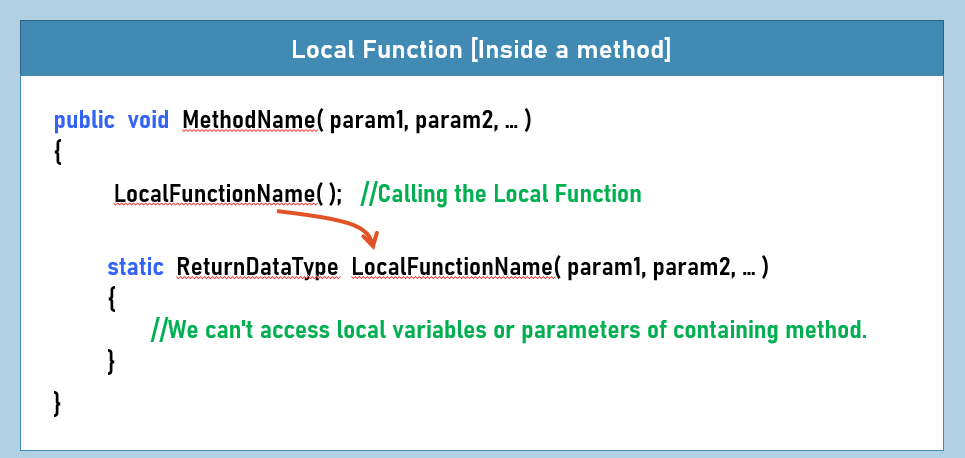


#### Static Local Functions

"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.

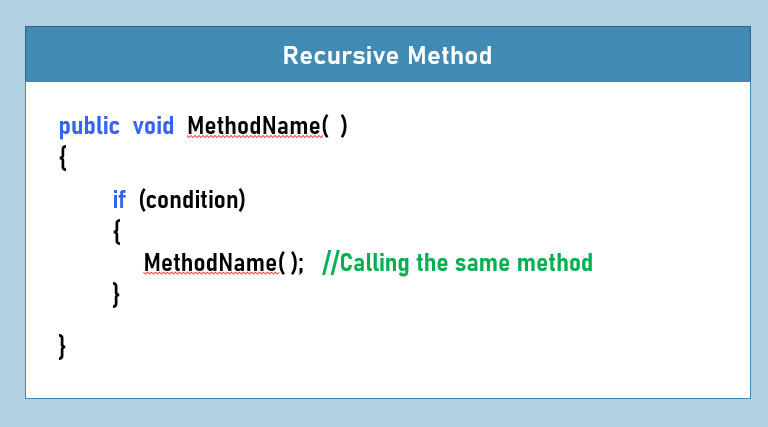
This is to avoid accidental access of local variables or parameters of containing method, inside the local function.



#### 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.

**Section Cheat Sheet (PPT)--7**

#### 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

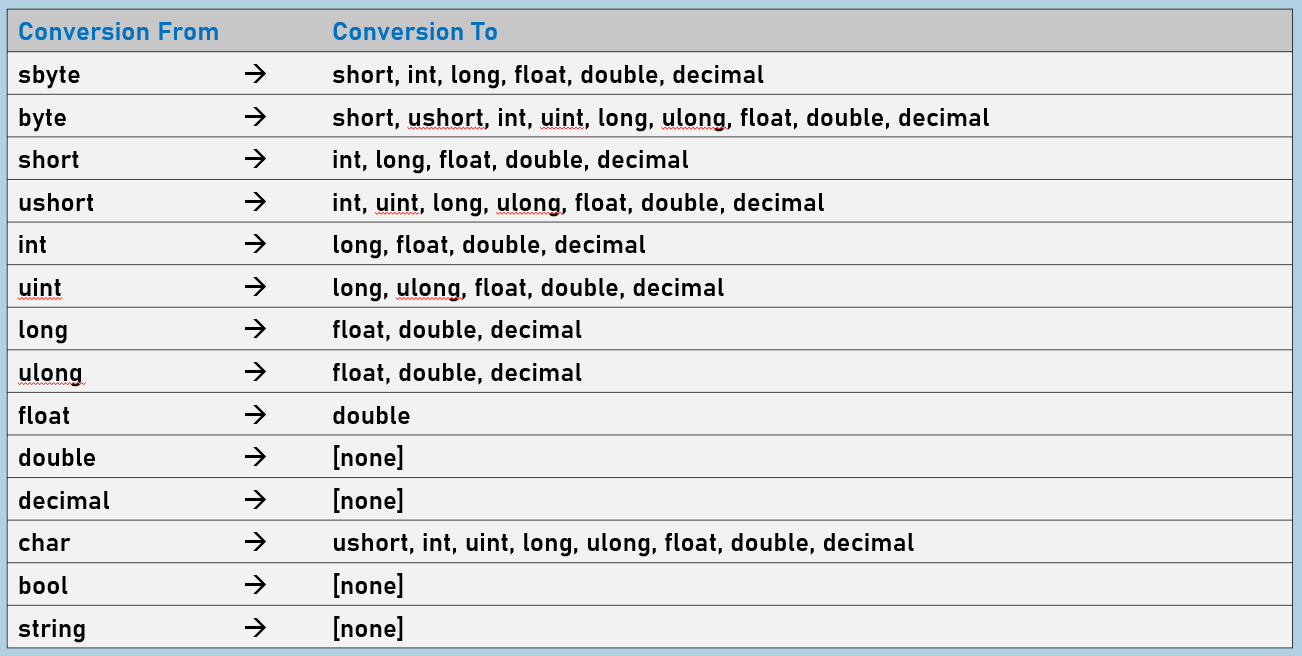
(from string to numerical-type)

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'.



#### 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.

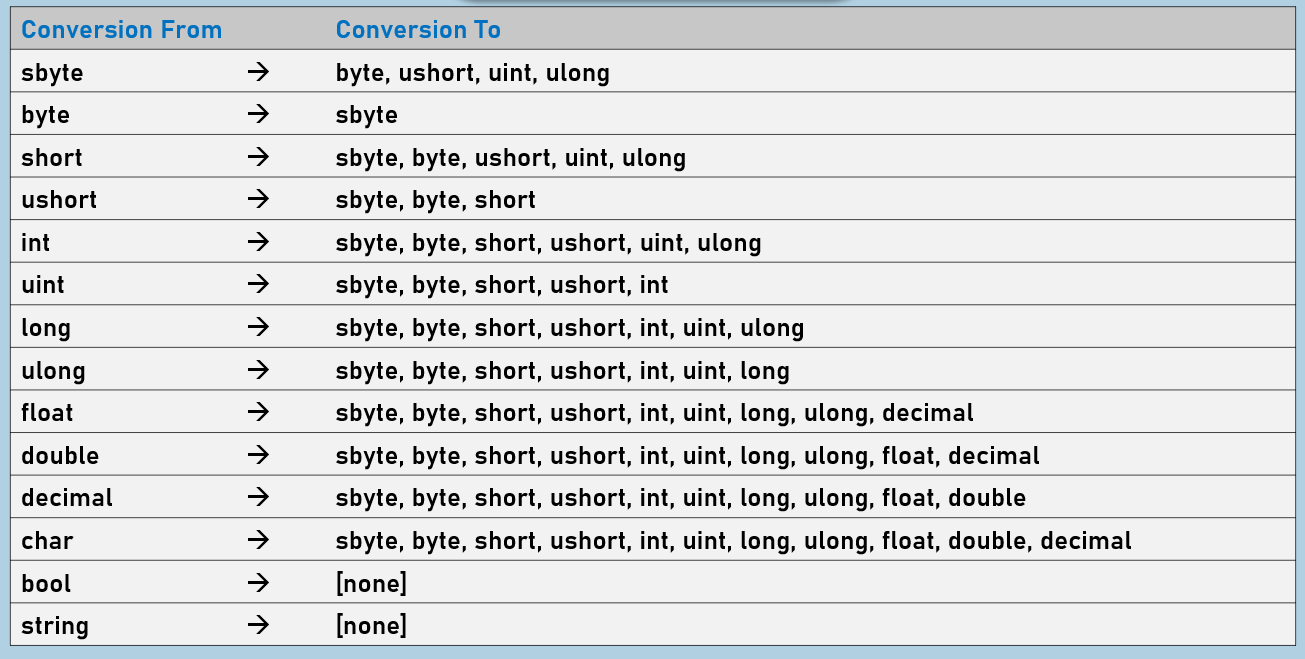
**Loosy conversion:** If the destination type is not sufficient-enough to store the converted value, the value may loose.

**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.



#### Parse

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

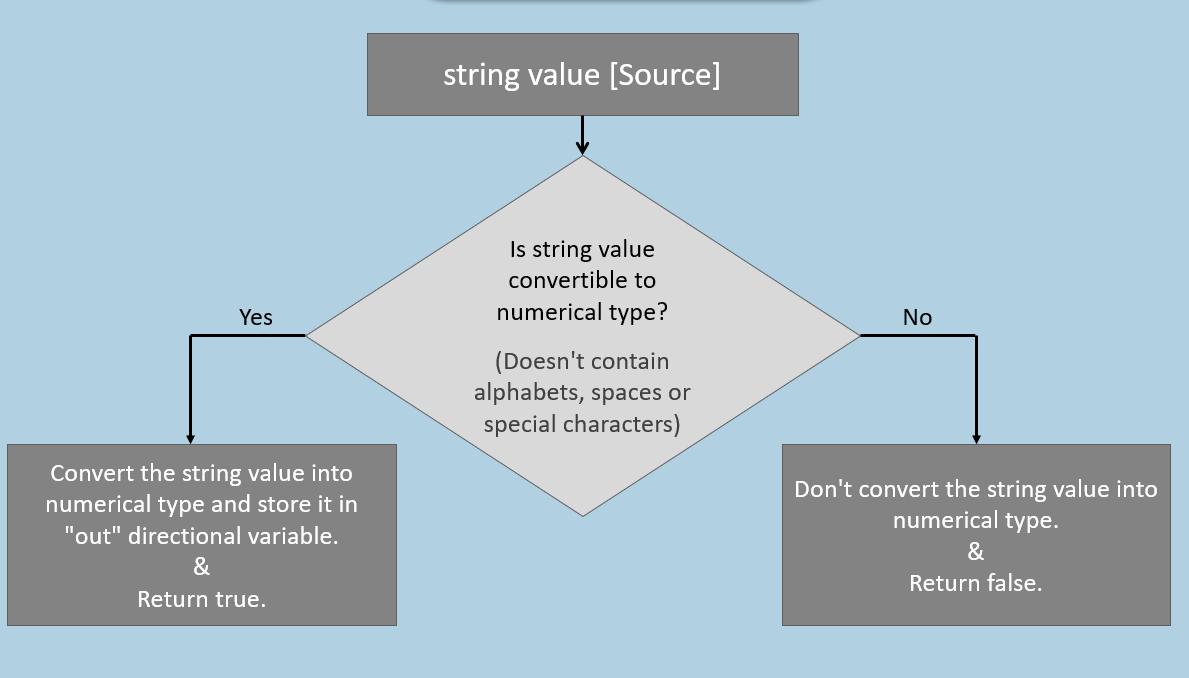
Eg:  string à int

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.

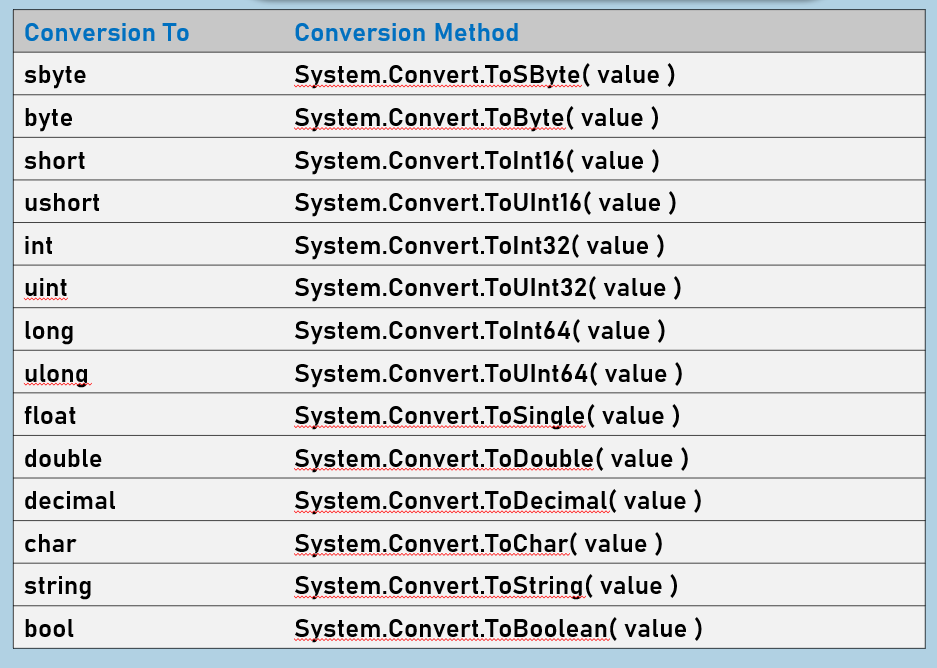
It raises FormatException, if the source value is invalid.

For each data type, we have a conversion method.

All conversion methods are static methods.

**Syntax:**

type destinationVariable = Convert.ConversionMethod (SourceValue )



**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.

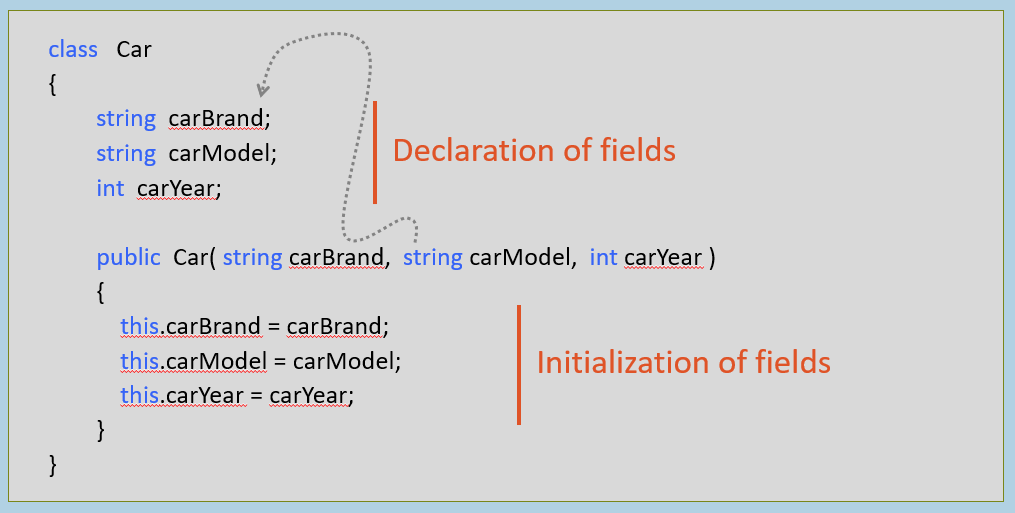
**Section Cheat Sheet (PPT)--8**

#### Constructors

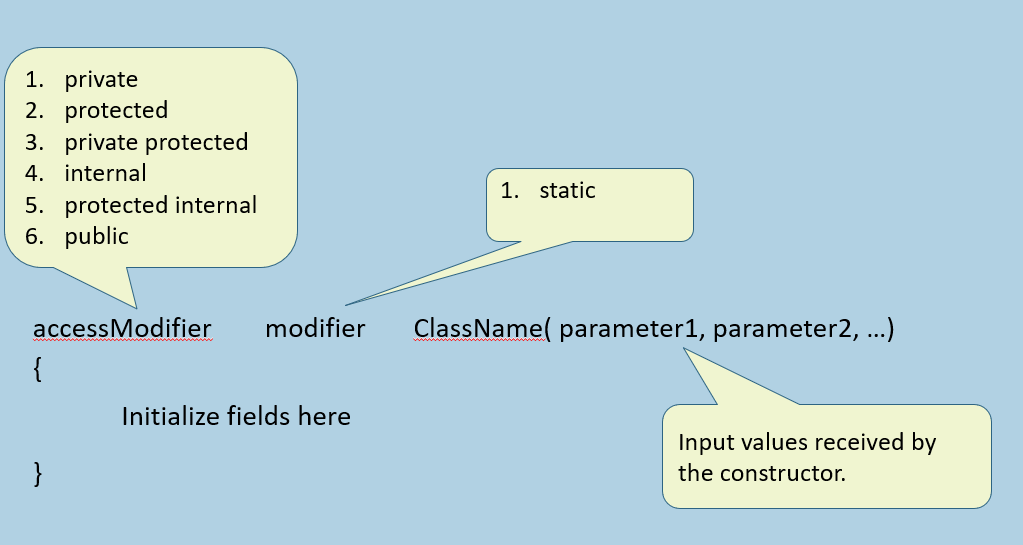
Special method of class, which contains initialization logic of fields.

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**

1. public ClassName( Parameter1, Parameter2, … )
2. {
3. …
4. }
5. Initializes instance fields.
6. Executes automatically every time when a new object is created for the class.
7. "private" by default; We can use any of access modifiers.
8. Can contain any initialization logic, that should be executed every time when a new object is created for the class.

**Static Constructor**

1. static ClassName( )
2. {
3. …
4. }
5. Initializes static fields.
6. 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.
7. "public" by default; Access modifier can't be changed.
8. 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**

1. public ClassName( )
2. {
3. …
4. }
5. Constructor without parameters.
6. 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. }
5. Constructor with one or more parameters.
6. It generally initializes fields by assigning values of parameters into fields.

#### Implicit (vs) Explicit Constructor

****Implicit Constructor (after compilation)****

1. public ClassName( )
2. {
3. }
4. 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".
5. 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. }
5. The constructor (parameter-less or parameterized) while is created by the developer is called as "Explicit Constructor"
6. 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. }
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:**

IMG_258

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.