C# PROGRAMMING TUTORIAL
FROM FRACE MARTEJA

COMMON BASIC PRINCIPLES

- **KISS** - KEEP IT SIMPLE STUPID - IN CODING; THE **SHORTER**, THE **BETTER**.
- **DAMP** - DESCRIPTIVE AND MEANING PHRASES - DIFFERENT **VARIABLES** WITH DIFFERENT **UNIQUE NAMES**.
- **WET** - WRITE EVERYTHING TWICE - TO **ENSURE** THE CODE.
- **DRY** - DO NOT REPEAT YOURSELF - TO **AVOID** SOME MISTAKES OF **REPEATING EVERY CODE**.

DATATYPE - A DATATYPE, IN PROGRAMMING, IS A CLASSIFICATION THAT SPECIFIES WHICH TYPE OF VALUE A VARIABLE HAS AND WHAT TYPE OF MATHEMATICAL, RELATIONAL OR LOGICAL OPERATIONS CAN BE APPLIED TO IT WITHOUT CAUSING AN ERROR. STRING FOR ALPHANUMERIC CHARACTERS; INTEGER FOR WHOLE NUMBERS; FLOAT FOR NUMBER WITH A DECIMAL POINT; CHARACTER FOR ENCODING TEXT NUMERICALLY; AND BOOLEAN FOR REPRESENTING LOGICAL VALUES.

DATATYPE CATEGORIES

VALUE TYPE - IT **STORE** ITS CONTENT IN **MEMORY** ALLOCATED ON THE **STACK**.
REFERENCE TYPE - IT BASICALLY **STORED** ON THE **HEAP** AND PASSED BY CREATING A REFERENCE.
POINTER TYPE - IT IS **NOT USUALLY USE** IN **C#** PROGRAMMING.

DATATYPE - VALUETYPE

BOOL(bool) - **TRUE** OR **FALSE** - IT IS USUALLY USE FOR **IF STATEMENT** USING TRUE OR FALSE.

CHAR(char) - **SINGLE CHARACTER** - IT IS BASICALLY USE FOR **SINGLE CHARACTER VARIABLES**.

DECIMAL(decimal) - **0.0M** - IF YOU WANT TO HAVE AN **ACCURATE** DECIMAL RESULT. IT IS **MORE ACCURATE** THAN **DOUBLE** AND **FLOAT**.

DOUBLE(double) - **0.0D** - **MORE ACCURATE** THAN **FLOAT** BUT **LESS ACCURATE** THAN **DECIMAL**. IT IS THE **DEFAULT DATATYPE** IN C# IF YOU ARE USING NUMBERS WITH DECIMAL.

FLOAT(float) - IT IS **LESS ACCURATE** THAN **DECIMAL** AND **DOUBLE**. IT IS ALSO USEFULL IF YOU DO NOT NEED AN ACCURATE RESULT OF DECIMAL.

INT(int) - IT IS **INTEGER** THAT USED FOR **WHOLE NUMBERS**. IT IS **RELATED** TO **BYTE**, **SBYTE**, **LONG** AND **SHORT** FOR USING AS A DATATYPE FOR WHOLE NUMBERS.

DATATYPE - REFERENCE TYPE

OBJECT (object) - IT IS THE **ULTIMATE BASE CLASS** OF ALL .NET CLASS. IT IS WHERE **INHERIT** ALL DATATYPE.

STRING (string) - IT IS THE **SERIES OF CHARACTER**. IT IS USED FOR **WORDS** AND **SENTENCES**. SEQUENCE OF ZERO OR MORE UNICODE CHARACTERS.

CLASS (class) - **USER-DEFINED BLUEPRINT** OR **PROTOTYPE** FROM WHICH OBJECTS ARE CREATED.

DYNAMIC (dynamic) - INDICATES THE **USE** OF THE **VARIABLE** AND **REFERENCES** TO ITS MEMBERS BYPASS COMPILE-TIME TYPE CHECKING.

INTERFACE (interface) - **DEFINES** A **CONTRACT**.

DELEGATE (delegate) - **SIMILAR** TO A **METHOD SIGNATURE**.

RECORD (record) - KEYWORD TO DEFINE A **REFERENCE** THAT PROVIDES **BUILT-IN FUNCTIONALITY** FOR ENCAPSULATING DATA.

STACK AND HEAP DEFINITIONS

STACK - IT IS A **PLACE** IN THE **COMPUTER MEMORY** WHERE ALL THE **VARIABLES** THAT ARE **DECLARED** AND **INITIALIZED** BEFORE **RUNTIME** ARE STORED.

HEAP - IT IS THE **SECTION** OF **COMPUTER MEMORY** WHERE ALL THE **VARIABLES CREATED** OR **INITIALIZED** AT **RUNTIME** ARE STORED.

THE HEAP SPACE CONTAINS ALL OBJECT ARE CREATED, BUT THE STACKS CONTAINS ANY REFERENCE TO THOSE OBJECTS. OBJECTS STORED IN THE HEAP CAN BE ACCESSED THROUGHOUT THE APPLICATION. PRIMITIVE LOCAL VARIABLES ARE ONLY ACCESSED THE STACK MEMORY BLOCK THAT CONTAIN THEIR METHOD.

VARIABLE - IT IS A **NAME GIVEN** TO A STORAGE AREA THAT IS USED TO **STORE VALUES** OF V**ARIOUS DATATYPES**. EACH VARIABLE IN C# NEEDS TO HAVE A **SPECIFIC TYPE**, WHICH **DETERMINES** THE **SIZE** AND **LAYOUT** OF THE VARIABLE'S MEMORY. IT IS THE **TEMPORARY STORAGE**.

THE **NAME** OF THE **VARIABLE** MUST BE **ITS PURPOSE** TO NOT VIOLATE THE **DAMP PRINCIPLE**.

VARIABLES CAN BE **DECLARED** IN **ONE LINE** OR **ONE VARIABLE PER LINE**.

THE **LIFE** OF **EVERY VARIABLE NAME** IS ONLY ON THE **BLOCK OF CODES** WHERE IT **LOCATES**. IF THE **VARIABLE NAME** HAD ALREADY **REACHED** THE **END** OF **BLOCK OF CODES**, IT WILL **RELEASE ITS ALLOCATION** TO THE MEMORY. WE **CANNOT** USE THE **SAME** VARIABLE NAME IN THE **SAME** BLOCK OF CODES OR INSIDE OF CURLY BRACKETS BUT IT IS **POSSIBLE** IF WE ARE GOING TO USE **SAME VARIABLE** NAME BUT ON A **DIFFERENT BLOCK OF CODES**.
Example: static void Main(string[] args)

```
{
// this is the block of code. it used open and close curly brackets to make block of code.
}
```

TO SAVE MEMORY SPACE YOU CAN USE DATATYPE WITH LESS STORAGE IF YOU DO NOT NEED LARGE STORAGE.

INT CAN BE TRANSFERRED TO **LONG** BUT CANNOT BE TRANSFERRED TO **SHORT**. LIKEWISE, **SHORT** AND **INT** CAN BE TRANSFERRED TO **LONG** BUT **LONG** CANNOT BE TRANSFERRED TO **SHORT** AND **INT** BECAUSE OF ITS STORAGE CAPACITY.

FLOAT AND **DOUBLE** ARE USE FOR NUMERICS **WITH DECIMAL PLACES**, IT IS ACCURATE FOR **MEASUREMENT**. **DECIMAL** DATATYPE IS ALSO USE FOR NUMERICS **WITH DECIMAL PLACES** BUT **MORE ACCURATE** THAN **FLOAT** AND **DOUBLE** SO IT IS THE DATATYPE THAT USED FOR **MONEY** OR **CURRENCIES.**

DECLARATION OF VARIABLES (EXPLICIT, IMPLICIT & DYNAMIC)

- DIFFERENT WAYS OF DECLARING LOCAL VARIABLES.

EXPLICIT - IT IS THE **NORMAL DECLARATION** OF VARIABLE. IT CONTAINS THE **DATATYPE** WE NEED AND ITS **VARIABLE NAME** WITH ITS **VALUE**. IT **NOT REQUIRED** THE **INITIALIZATION** BECAUSE THE DATATYPE IS ALREADY CONCLUDED.

```
int sampleNumber = 3; // using explicit as a declaration of variable
```

IMPLICIT - IT IS THE DECLARATION OF VARIABLE THAT **YET TO BE KNOWN** THE DATAYPE NEEDED **AFTER** THE **ASSIGN** OR **INPUT** OF **VARIABLE VALUE**. IT **REQUIRED** THE **INITIALIZATION** TO GET THE DATATYPE.

```
var sampleNumberImplicit = 3; // using implicit as a declaration of variable
```

DYNAMIC - IT IS THE DECLARATION OF VARIABLE THAT HAS THE **SIMILARITIES WITH IMPLICIT**, BUT DYNAMIC **DETERMINED** THE DATAYPE **DURING RUNTIME**.

```
Dynamic sampleNumberDynamic = 5; // using dynamic as a declaration of variable
```

RUNTIME - IT IS WHERE WE **RUN** THE PROGRAM.
DESIGN TIME - IT IS WHERE WE WERE **WRITING** THE CODES.

SCOPE OF VARIABLES (LOCAL, PUBLIC AND PRIVATE)

THE PART OF THE PROGRAM WHERE A PARTICULAR VARIABLE IS ACCESSIBLE IS TERMED AS THE SCOPE OF THE VARIABLES. EVERY VARIABLE IS ONLY EXCLUSIVE IN ITS OWN OPEN AND CLOSE CURLY BRACKETS.

LOCAL VARIABLE - IT IS A TYPE OF VARIABLE THAT CAN BE USED WHERE THE SCOPE AND EXTENT OF THE VARIABLE IS WITHIN THE METHOD OR STATEMENT BLOCK IN WHICH IT IS DECLARED.

FIELD VARIABLE - IT IS A VARIABLE THAT IS DECLARED AS A MEMBER OF A CLASS OR DECLARED OUTSIDE ANY METHOD OR CONSTRUCTOR WITHIN THE CLASS OR STRUCT. FIELDS ARE MEMBERS OF THEIR CONTAINING TYPE. A CLASS OR STRUCT MAY HAVE INSTANCE FIELDS, STATIC FIELDS, OR BOTH. INSTANCE FIELDS ARE SPECIFIC TO AN INTANCE OF A TYPE.

THE DIFFERENCE BETWEEN VARIABLE AND FIELD IN C#

VARIABLES REPRESENT STORAGE LOCATIONS. EVERY VARIABLE HAS A TYPE THAT DETERMINES WHAT VALUES CAN BE STORED IN THE VARIABLE. A **FIELD** IS A VARIABLE OF ANY TYPE THAT IS DECLARED DIRECTLY IN A CLASS OR STRUCT.

```
public class Episode1Class
{
        int sampleFieldInt = 5; // this field is accessible for method 1 & 2

        public void SampleMethod1()
        {
                int sampleLocalInt = 1;
                var sampleLocalDouble = 1.5;
        }

        public void SampleMethod2()
        {
                int sample2Int = 1;
        }
}
```

PUBLIC - THE TYPE OR MEMBER CAN BE ACCESSED BY ANY OTHER CODE IN THE SAME ASSEMBLY OR ANOTHER ASSEMBLY THAT REFERENCES IT. THE ACCESSIBILITY LEVEL OF PUBLIC MEMBERS OF A TYPE IS CONTROLLED BY THE ACCESSIBILITY LEVEL OF THE TYPE ITSELF.

PRIVATE - THE TYPE OR MEMBER CAN BE ACCESSED ONLY BY CODE IN THE SAME CLASS OR STRUCT.

PROTECTED - THE TYPE OR MEMBER CAN BE ACCESSED ONLY BY CODE IN THE SAME CLASS, OR IN A CLASS THAT IS DERIVED FROM THAT CLASS.

INTERNAL - THE TYPE OR MEMBER CAN BE ACCESSED BY ANY CODE IN THE SAME ASSEMBLE, BUT NOT FROM ANOTHER ASSEMBLY. IN OTHER WORDS, INTERNAL TYPES OR MEMBERS CAN BE ACCESSED FROM CODE THAT IS PART OF THE SAME COMPILATION.

PROTECTED INTERNAL - THE TYPE OR MEMBER CAN BE ACCESSED BY ANY CODE IN THE ASSEMBLE IN WHICH IT IS DECLARED, OR FROM WITHIN A DERIVED CLASS IN ANOTHER ASSEMBLY.

PRIVATE PROTECTED - THE TYPE OR MEMBER CAN BE ACCESSED BY TYPES DERIVED FROM THE CLASS THAT ARE DECLARED WITHIN ITS CONTAINING ASSEMBLY.

CALLER'S LOCATION	PUBLIC	PROTECTED INTERNAL	PROTECTED	INTERNAL	PRIVATE PROTECTED	PRIVATE
WITHIN THE CLASS	✔	✔	✔	✔	✔	✔
DERIVED CLASS (SAME ASSEMBLY)	✔	✔	✔	✔	✔	✖
NON-DERIVED CLASS (SAME ASSEMBLY)	✔	✔	✖	✔	✖	✖
DERIVED CLASS (DIFFERENT ASSEMBLY)	✔	✔	✔	✖	✖	✖
NON-DERIVED CLASS (DIFFERENT ASSEMBLY)	✔	✖	✖	✖	✖	✖

C# SCOPE RULES OF VARIABLES CAN BE DIVIDED INTO THREE CATEGORIES AS FOLLOWS:

- CLASS LEVEL SCOPE
- METHOD LEVEL SCOPE
- BLOCK LEVEL SCOPE

CLASS LEVEL SCOPE

- DECLARING THE VARIABLES IN A CLASS BUT OUTSIDE ANY METHOD CAN BE DIRECTLY ACCESSED ANYWHERE IN THE CLASS.
- THESE VARIABLES ARE ALSO TERMED AS THE FIELDS OR CLASS MEMBERS.
- CLASS LEVEL VARIABLE CAN BE ACCESSED BY THE NON-STATIC METHODS OF THE CLASS IN WHICH IT IS DECLARED.

- ACCESS MODIFIER OF CLASS LEVEL VARIABLES DOES NOT AFFECT THEIR SCOPE WITHIN A CLASS.
- MEMBER VARIBLES CAN ALSO BE ACCESSED OUTSIDE THE CLASS BY USING THE ACCESS MODIFIERS.

EXAMPLE:

```
// C# program to illustrate the Class Level Scope of Variables
using System;

// declaring a class
class GFG { // from here class level scope starts

        // this is a class level variable
        // having class level scope
        int a = 10;

        // declaring a method
        Public void display()
        {
                // accessing class level variable
                Console.WriteLine(a);

        } // here method ends

} // here class level scope ends
```

METHOD LEVEL SCOPE

- VARIABLES THAT ARE DECLARED INSIDE A METHOD HAVE METHOD LEVEL SCOPE. THESE ARE NOT ACCESSIBLE OUTSIDE THE METHOD.
- HOWEVER, THESE VARIABLES CAN BE ACCESSED BY THE NESTED CODE BLOCKS INSIDE A METHOD.
- THESE VARIABLES ARE TERMED AS THE LOCAL VARIABLES.
- THERE WILL BE A COMPILE-TIME ERROR IF THESE VARIABLES ARE DECLARED TWICE WITH THE SAME NAME IN THE SAME SCOPE.
- THESE VARIABLES DO NOT EXIST AFTER METHOD'S EXECUTION IS OVER.

EXAMPLE:

```
// C# program to illustrate the Method Level Scope of Variables
using System;

// declaring a class
class GFG { // from here class level scope starts

        // declaring a method
        public void display()

        { // from here method level scope starts
```

```
            // this variable has method level scope
            int m = 47;

            // accessing method level variable
            Console.WriteLine(m);

        } // here method level ends

        // declaring a method
        public void display1()

        { // from here method level scope starts

            // it will give compile time error as
            // you are trying to access the local
            // variable of method display()
            Console.WriteLine(m);

        } // here method level scope ends

} // here class level scope ends
```

BLOCK LEVEL SCOPE

- THESE VARIABLES ARE GENERALLY DECLARED INSIDE THE FOR, WHILE STATEMENTS ETC.
- THESE VARIABLES ARE ALSO TERMED AS THE LOOP VARIABLES OR STATEMENTS VARIABLES AS THEY HAVE LIMITED THEIR SCOPE UP TO THE BODY OF THE STATEMENT IN WHICH IT DECLARED.
- GENERALLY, A LOOP INSIDE A METHOD HAS THREE LEVEL OF NESTED CODE BLOCKS(I.E. CLASS LEVEL, METHOD LEVEL, LOOP LEVEL).
- THE VARIABLE WHICH IS DECLARED OUTSIDE THE LOOP IS ALSO ACCESSIBLE WITHIN THE NESTED LOOPS. IT MEANS A CLASS LEVEL VARIABLE WILL BE ACCESSIBLE TO THE METHODS AND ALL LOOPS. METHOD LEVEL VARIABLE WILL BE ACCESSIBLE TO LOOP AND METHOD INSIDE THAT METHOD.
- A VARIABLE WHICH IS DECLARED INSIDE A LOOP BODY WILL NOT BE VISIBLE TO THE OUTSIDE OF LOOP BODY.

EXAMPLE:

```
// C# program to illustrate the Method Level Scope of Variables
using System;

// declaring a class
class GFG

{ // from here class level scope starts
```

```
        // declaring a method
        public void display()

        { // from here method level scope starts

                // this variable has method level scope
                int i = 0;
                for (i = 0; i < 4; i++) {

                        // accessing method level variable
                        Console.WriteLine(i);
                }
                {
                // here j is block of level variable
                // it is only accessible inside
                // this for loop
                for (int j = 0; j < 5; j++) {
                        // accessing block level variable
                        Console.WriteLine(j);
                }

                // this will give error as block level
                // variable can not be accessed outside the block
                Console.WriteLine(j):
        } // here method level scope ends

} // here class level scope ends
```

KEYWORDS (STATIC, READONLY & CONSTANT)

CONSTANT (const) - IT IS A VARIABLE OF WHICH THE VALUE IS CONSTANT BUT A COMPILE TIME, AND IT IS MANDATORY TO ASSIGN A VALUE TO IT. BY DEFAULT A CONST IS STATIC AND WE CANNOT CHANGE THE VALUE OF A CONST VARIABLE THROUGHOUT THE ENTIRE PROGRAM.

WHEN TO USE CONST - WE COULD USE THE CONST KEYWORD WHEN THE VARIABLES ARE PRIMITIVE TYPES AND THEIR VALUE WILL NEVER CHANGE. IF WE CHANGE THE VALUE OF THE CONST VARIABLE, THEN WE HAVE TO RECOMPILE THE PROJECT.

READONLY (readonly) - READONLY IS THE KEYWORD WHOSE VALUE WE CAN CHANGE DURING RUNTIME OR WE CAN ASSIGN IT AT RUN TIME BUT ONLY THROUGH THE NON-STATIC CONSTRUCTOR. NOT EVEN A METHOD.

WHEN TO USE READONLY - WE COULD USE A READONLY VARIABLE WHEN WE HAVE TO INITIALIZE THE VARIABLE DURING THE CLASS CONSTRUCTION AND THEN WILL NEVER BE CHANGED LATER ON.

STATIC READONLY (static) - A STATIC READONLY TYPE VARIABLE'S VALUE CAN BE ASSIGNED AT RUNTIME OR ASSIGNED AT COMPILE TIME AND CHANGED AT RUNTIME. BUT THIS VARIABLE'S VALUE CAN ONLY BE CHANGE IN THE STATIC CONSTRUCTOR, AND CANNOT BE CHANGED FURTHER. IT CAN CHANGE ONLY ONCE AT RUNTIME.

WHEN TO USE STATIC READONLY VARIABLE - STATIC READONLY VARIABLES ARE SIMILAR TO CONST VARIABLES AND WE COULD USE THEM WHEN WE NEED TO SHARE THEM BY ALL INSTANCES. BECAUSE, IF THE VALUE OF A FIELD IN ANOTHER ASSEMBLY IS CHANGED, THE CHANGES WILL BE VISIBLE AS SOON AS THE ASSEMBLY IS LOADED, WITHOUT THE NEED TO RECOMPILE.

PARSE AND CONVERT

PARSE (int.Parse) - A METHOD IN C# THAT CONVERTS A STRING REPRESENTATION TO A CORRESPONDING NUMERICAL INTEGER VALUE. IF THERE IS AN ERROR IT WILL GIVE AN **ArgumentNullException**.
SYNTAX IS int.Parse(string s);

```
//Parse
string textOne = "3";
var intOne = int.Parse(textOne);
var result = intOne * 5;
Console.WriteLine(result);
// or in another way
Console.WriteLine($"(Parse) textOne result: {result}");
```

CONVERT (Convert.ToInt32) - A METHOD IN C# THAT CONVERTS A SPECIFIED STRING REPRESENTATION OF A NUMBER TO AN EQUIVALENT 32-BIT SIGNED INTEGER. IF THERE IS AN ERROR IT WILL GIVE A **ZERO**.
SYNTAX IS Convert.ToInt32(string s);

```
//Convert
var intConverted = Convert.ToInt32(textOne);
Result = intConverted * 5;
Console.WriteLine($"(Convert) textOne result: {result}");
```

TRYPARSE (TryParse) - IT IS A .NET C# METHOD THAT ALLOWS YOU TO TRY AND PARSE A STRING INTO A SPECIFIED TYPE. IT RETURNS A BOOLEAN VALUE INDICATING WHETHER THE CONVERSION WAS SUCCESSFUL OR NOT. IF CONVERSION SUCCEEDED, THE METHOD WILL RETURN TRUE AND THE CONVERTED VALUE WILL BE ASSIGNED TO THE OUPUT PARAMETER.
SYNTAX IS int.TryParse(string s);

```
//TryParse
String textThree = ":)";
int intThree;
if (int.TryParse(textThree, out intThree))
{
        Console.WriteLine($"Result Three using TryParse");
}
else
{
        Console.WriteLine($"The textThree is not a number");
}
```

USING SYSTEM IN C#

THE **USING SYSTEM LINE** MEANS THAT YOU ARE USING THE SYSTEM LIBRARY IN YOU PROJECT. WHICH GIVES YOU SOME USEFUL CLASSES AND FUNCTIONS LIKE CONSOLE CLASS OR THE WRITELINE FUNCTION/METHOD. THE NAMESPACE PROJECTNAME IS SOMETHING THAT IDENTIFIES AND ENCAPSULATES YOUR CODE WITHIN THAT NAMESPACE.

NAMESPACE - IT IS USED TO LOGICALLY ARRANGE CLASSES, STRUCTS, INTERFACES ENUMS, AND DELEGATES. THE NAMESPACES IN C# CAN BE NESTED. THAT MEANS ONE NAMESPACE CAN CONTAIN OTHER NAMESPACES ALSO.

COMMENT - IT IS AN ANNOTATION IN A SOURCE CODE WITH INTENTION TO GIVE PROGRAMMER A READABLE EXPLANATION OF THE CODE. THESE ANNOTATIONS ARE IGNORED BY COMPILERS WHEN COMPILING YOUR CODE.

SINGLE LINE COMMENT - SINGLE-LINE COMMENTS ALLOW NARRATIVE ON ONLY ONE LINE AT A TIME.
CODE: "//"

MULTI-LINE COMMENT - MULTI-LINE COMMENTS HAVE ONE OR MORE LINES OF NARRATIVE WITHIN A SET OF COMMENT DELIMITERS.
CODE: "/* AND */"

INPUTS, EXPRESSIONS AND OPERATORS

INPUT (READ) - USE THE **Console.ReadLine();** METHOD TO READ INPUT FROM THE CONSOLE IN C#. THIS METHOD RECEIVES THE INPUT AS STRING, THEREFORE YOU NEED TO CONVERT IT.

EXPRESSION - IT IS A SEQUENCE OF ONE OR MORE OPERANDS AND ZERO OR MORE OPERATORS THAT CAN BE EVALUATED TO A SINGLE VALUE, OBJECT, METHOD OR NAMESPACE.

FOR EXAMPLE:

- CONSTANT EXPRESSION: **10**
- LITERAL EXPRESSION WITH "*" (MULTIPLICATION) OPERATOR: **5 * 2**
- COMPOUND EXPRESSION: **2 + (4 * 2)**
- VOID EXPRESSION: **Console.WriteLine();**
- ASSIGNMENT EXPRESSION: **A = 10**, COMPOUND ASSIGNMENT: **A = 5 + 5**
- PRIMARY EXPRESSION: **Math.Pow(10, 2)** // it means **10 ^ 2**

OPERATORS - THESE ARE SYMBOLS THAT ARE USED TO PERFORM OPERATIONS ON OPERAND. OPERATORS ARE USED TO MANIPULATE VARIABLES AND VALUES IN A PROGRAM.

- **ASSIGNMENT OPERATOR** ("**=**")

```
var a = 5;
a = 5;
```

- **ARITHMETIC OPERATORS** ("*** / + - %**")

```
var num1 = 3;
var num2 = 7;
var num1And2Result = num1 + num2;

// precedence : () [] ++ -- * / % + -
var samplePrecedence = (1 + 1) * 5 + 2 / 2 - 10 + 3;
Console.WriteLine(samplePrecedence);
```

- **BOOLEAN/ LOGICAL OPERATORS** ("**&& ||**")

```
var isActive = true;
var isSuspended = false;

var testAnd = isActive && isSuspended;
var testOr = isActive || isSuspended;
```

- **EQUALITY OPERATORS** ("**== !=**")

```
var eq1 = 100;
```

```
var eq2 = 90;

var testEq = eq1 == eq2;
var testNotEq = eq1 != eq2;
```

- **COMPARISON OPERATORS** (“**> < >= <=**”)

```
var eq1 = 100;
var eq2 = 90;

var testGreaterThan = eq1 > eq2;
var testLessThan = eq1 < eq2;

var testGreaterThanOrEqual = eq1 >= eq2;
var testLessThanOrEqual = eq1 <= eq2;
```

- **NULL-COALESCING OPERATORS** (“**??**”)

```
var sampleString = “may laman”;
string nullNaString = null;

var sampleResult1 = sampleString ?? “nilagyan ng laman”;
var sampleResult2 = nullNaString ?? “nilagyan ng laman”;

Console.WriteLine(sampleResult1);
Console.WriteLine(sampleResult2);
```

- **UNARY OPERATORS** (“**! ++ --**”)

```
var trueValue = true;
trueValue = !trueValue;

var numberIncrementDecrement = 10;
Console.WriteLine("Post-Increment");
// equivalent to: numberIncrementDecrement = numberIncrementDecrement +
1;
Console.WriteLine(numberIncrementDecrement++);
// equivalent to: numberIncrementDecrement = numberIncrementDecrement -
1;
Console.WriteLine(numberIncrementDecrement--);

numberIncrementDecrement = 10;
Console.WriteLine("Pre-Increment");
// post incr and decr
Console.WriteLine(++numberIncrementDecrement);
Console.WriteLine(--numberIncrementDecrement);
```

OUTPUT:
Post-Increment
10

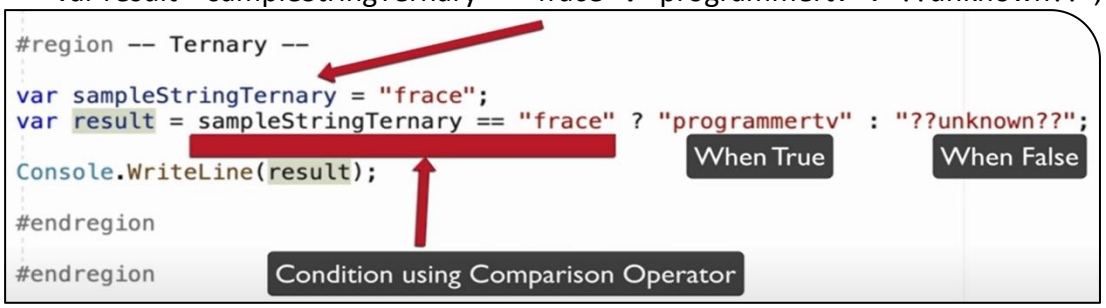
11
Pre-Increment
11
10

POST: <operand>**++** // EVALUATION BEFORE THE INCREMENT
PRE: **++**<operand> // INCREMENT BEFORE THE EVALUATION

POST: <operand>**--** // EVALUATION BEFORE THE DECREMENT
PRE: **--**<operand> // DECREMENT BEFORE THE EVALUATION

- **TERNARY OPERATORS** (“? :”)

var sampleStringTernary = “frace”;
var result = sampleStringTernary == “frace” **?** “programmertv” **:** “??unknown??”;



Console.WriteLine(result);

ANOTHER EXAMPLE:

Console.Write(“Input your age: ”);
var ageString = Console.ReadLine();
var age = int.Parse(ageString);

OR Console.Write(“Input your age: ”);
var age = Convert.ToInt32(Console.ReadLine));

Console.WriteLine(age < 18 ? “minor” : “adult”);

EXPRESSION & OPERATORS

// CLOSING: INPUTS + EXPRESSIONS + OPERATORS
// SAMPLE: PYTHAGOREAN THEOREM
// FORMULA TO GET C: C = SQUARE ROOT OF (A SQUARE + B SQUARE)

// INPUTS IS A AND B
// OUTPUT IS C

Console.WriteLine(“Input a value: ”);
var formulaA = double.Parse(Console.ReadLine());
Console.WriteLine(“Input b value: ”);

```
var formulaB = double.Parse(Console.ReadLine());

var formulaResult = Math.Sqrt(Math.Pow(formulaA, 2) + Math.Pow(formulaB, 2));
Console.WriteLine($"Sample C result: {formulaResult}");
```

FLOW CONTROL (IF, ELSE IF, ELSE, SWITCH, DO WHILE, WHILE & FOR LOOP)

FLOW CONTROL - LIKE IN ANY PROGRAMMING LANGUAGE, THERE ARE SEVERAL KEYWORDS THAT ARE USED TO ALTER THE FLOW OF THE PROGRAM. WHEN THE PROGRAM IS RUN, THE STATEMENTS ARE EXECUTED FROM THE TOP OF THE SOURCE FILE TO THE BOTTOM. ONE BY ONE. THIS FLOW CAN BE ALTERED BY SPECIFIC KEYWORDS. STATEMENTS CAN BE EXECUTED MULTIPLE TIME. SOME STATEMENTS ARE CALLED CONDITIONAL STAMENTS. THEY ARE EXECUTED ONLY IF A SPECIFIC CONDITION IS MET.

IF STATEMENT - THE **IF STATEMENT** CHECKS THE VALUE OF A BOOL. WHEN THE VALUE IS TRUE, THE STATEMENT FOLLOWING THE IF EXECUTES.

ELSE STATEMENT - THE **ELSE KEYWORD** EXECUTES ONLY WHEN THE CONDITION BEING TESTED IS FALSE.

ELSE IF STATEMENT - THE **ELSE IF KEYWORD** TESTS FOR ANOTHER CONDITION IF AND ONLY IF THE PREVIOUS CONDITION WAS NOT MET. NOTE THAT WE CAN USE MULTIPLE ELSE IF KEYWORDS IN OUR TESTS.

SYNTAX

```
If (condition)
{
  // statement(s)
}
else if (condition)
{
  // statement(s)
}
else
{
  // statement(s)
}
```

SWITCH - WHEN WE WANT TO CONTROL THE PROGRAM FLOW BASED ON THE VALUE OF ONE VARIABLE OR EXPRESSION ONLY, THEN THE “SWITCH” STATEMENT GIVES US A VERY CLEAN WAY DOING IT.

THE **BREAK**; STATEMENT IS OPTIONAL BUT NECESSARY. IF THERE IS NO BREAK; STATEMENT, THE PROGRAM FLOW WILL CONTINUE TO THE NEXT CASE AND WILL EXECUTE THAT AS WELL.

THE "**DEFAULT**" CASE IS EXECUTED WHEN THERE IS NO OTHER MATCHING CASE IN THE "SWITCH" STATEMENT.

SYNTAX:

```
Switch(expression)
{
 case x:
  // statement(s)
  break:
 case y:
  // statement(s)
  break:
 default:
  // statement(s)
  break:
}
```

WHILE - THE **WHILE STATEMENT** EXECUTES A STATEMENT OR A BLOCK OF STATEMENTS WHILE A SPECIFIED BOOLEAN EXPRESSION EVALUATIONS TO TRUE.

SYNTAX:

```
while (expression)
{
   statement;
}
```

DO WHILE - THE **DO STATEMENT** EXECUTES A STATEMENT OR A BLOCK OF STATEMENTS WHILE A SPECIFIED BOOLEAN EXPRESSION EVALUATES TO TRUE.

SYNTAX:

```
do {
 // statements(s)
} while (expression)
```

FOR LOOP - THE **FOR STATEMENT** EXECUTES A STATEMENT OR A BLOCK OF STATEMENTS WHILE A SPECIFIED BOOLEAN EXPRESSION EVALUATES TO TRUE.

FOREACH - THE **FOREACH STATEMENT** EXECUTES A STATEMENT OR A BLOCK OF STATEMENTS FOR EACH ELEMENT IN AN INSTANCE OF THE TYPE THAT IMPLEMENTS THE IEnumerable or IEnumerable<T> interface.

SYNTAX:

```
for (initialization; condition ;incrementation)
```

```
{
  // statement(s)
}

foreach (type variableName in enumerable)
{
  // statement(s)
}
```

PROGRAM FOR IF, ELSE IF & ELSE STATEMENT:

```
const int MAX_MINOR_AGE = 17;
const int MIN_SENIOR_AGE = 60;
var isPWD = true;
var age = 25;
if(age <= MAX_MINOR_AGE || age >= MIN_SENIOR_AGE)
{
        Console.WriteLine("Add minor and senior discount");
}  else if (isPWD)
{
        Console.WriteLine("Add PWD discount");
}
else
{
        Console.WriteLine("no Discount");
}

// Client Requirements: No 5 discount for ADULT.
// Implementation #1
if (age > MAX_MINOR_AGE && age < MIN_SENIOR_AGE)
{
}
else
{
        Console.WriteLine(“Added 5 Discount”);
}

// Implementation #2
if (!(age >= MAX_MINOR_AGE && age <= MIN_SENIOR_AGE))
{
        Console.WriteLine(“Added 5 Discount”);
}
```

PROGRAM FOR SWITCH STATEMENT:

```
var age = 15;
switch (age)
{
        case 1:
                Console.WriteLine(“Baby”);
                break;
        case 10:
        case 11:
        case 12:
        case 13:
        case 14:
        case 15:
        case 16:
        case 17:
        case 18:
        case 19:
                Console.WriteLine(“Teen”);
                break;
        default:
                Console.WriteLine(“”Unknown)
                break;
}
```

PROGRAM FOR WHILE:

```
var count = 0;
while (count < 3)
{
        count++;
        Console.WriteLine(count);
}
```

PROGRAM FOR DO WHILE:

```
do
{
        Console.WriteLine(“Print Using Do While”);
        Console.WriteLine(count);
        count++;
}  while (count < 3);
```

PROGRAM For FOR LOOP:

```
for (var index = 0; index <= 5; index+= 2)
{
        Console.WriteLine($“index value {index}”);
```

```
}
```

PROGRAM For FOREACH

```
var nameSegments = new [] { “pro”, “grammer”, “tv” };
foreach(var segment in nameSegments)
{
        Console.WriteLine(&“segment value: {segment}”)
}
```

ARRAY, ENUMERABLE, COLLECTION & LIST

ARRAY - YOU CAN STORE MULTIPLE VARIABLES OF THE SAME TYPE IN AN ARRAY DATA STRUCTURE. YOU DECLARE AN ARRYA BY SPECIFYING THE TYPE OF ITS ELEMENTS.

- AN ARRAY CAN BE **SINGLE-DIMENSIONAL**, **MULTIDIMENSIONAL** OR **JAGGED**.
- THE NUMBER OF DIMENSIONS AND THE LENGTH OF EACH DIMENSION ARE ESTABLISHED WHEN THE ARRAY INSTANCE IS CREATED.
- ARRAYS ARE ZERO INDEXED: AN ARRAY WITH N ELEMENTS IS INDEXED FROM 0 TO N-1.
- ARRAY ELEMENTS CAN BE OF ANY TYPE, INCLUDING AN ARRAY TYPE.

```
int[] numbers = new int[10];

int[] numbersWithInIt = new int[] { 1, 2, 3, };

int[][] multiNumbers = new int[3][]
{
        new int[3] { 1, 2, 3},
        new int[3] { 4, 5, 6},
        new int[3] { 7, 8, 9}
} ;
int[,] otherNumbers = new int[2,2];
```

SINGLE-DIMENSIONAL ARRAY

SYNTAX: int[] numbers = new int[] { 1, 2, 3, 4 };

INDEX	VALUE
0	1
1	2
2	3
3	4

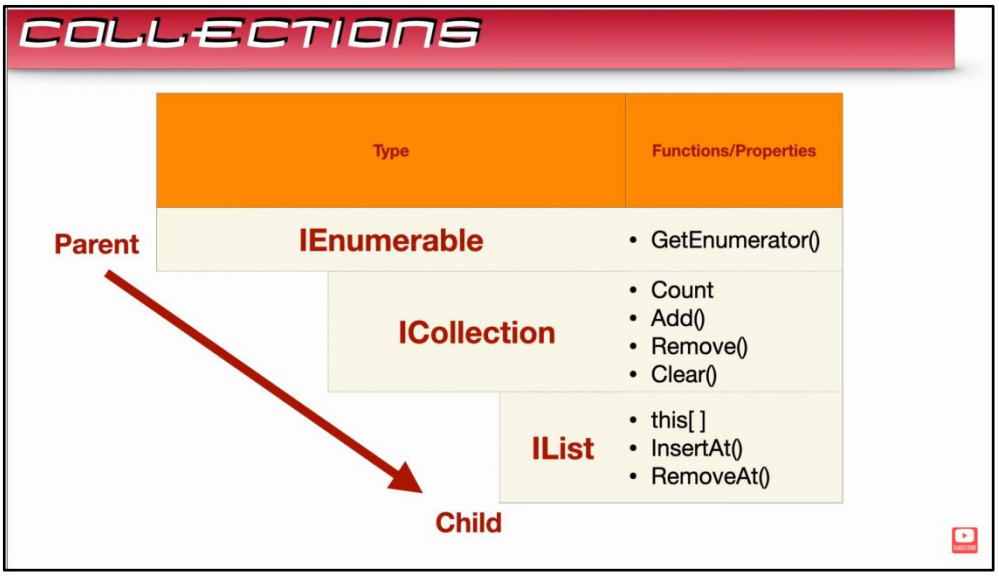
MULTIDIMENSIONAL OR JAGGED ARRAY

SYNTAX: int[][] numbers = new int [][]
{
new int[]{1, 2, 3, 4},
new int[]{5, 6, 7, 8}
};

INDEX	VALUE	
0	**INDEX**	**VALUE**
	0	1
	1	2
	2	3
	3	4
1	**INDEX**	**VALUE**
	0	5
	1	6
	2	7
	3	8

COLLECTIONS: ENUMERABLE, COLLECTION & LIST

AN **ILIST(IList)** IS AN **ICOLLECTION(ICollection)** THAT CAN BE ACCESSED BY INDEX. AN **ICOLLECTION(ICollection)** IS AN **IENUMERABLE(IEnumerable)** WITH EASY ACCESS TO THINGS LIKE ADD, REMOVE, AND COUNT. AN **ENUMERABLE(IEnumerable)** IS ANYTHING THAT CAN BE ENUMERATED, EVEN IF THE LIST OF THOSE THINGS DOESN'T EXIST UNTIL YOU ENUMERATE IT.



PROGRAM FOR ENUMERABLE, COLLECTION & LIST

```
using System;
using System.Collections.Generic;

namespace Training
{
        class Program
        {
                static void Main(string[] args)
                {
```

PROGRAM FOR ENUMERABLE

```
                        var countries = new[] { "Philippines", "USA", "Canada",
                        "Pakistan", "Afghanistan" };
                        IEnumerable<string> enumCountries = countries
                        // enumCountries[0] = "PHL";
                        Console.WriteLine("ENUMERABLE");
                        foreach(var country in enumCountries)
                        {
                                Console.WriteLine(country);
                        }
```

PROGRAM FOR COLLECTION

```
                ICollection<string> colCountries = new List<string>(countries);
                colCountries.Add("Australia");
                colCountries.Remove("USA");
                //colCountries[0] = "SampleError";

                Console.WriteLine("COLLECTION");
                foreach (var country in colCountries)
                {
                        Console.WriteLine(country);
                }
```

PROGRAM FOR LIST

```
                var listCountries = new List<string>(countries);
                // with initial value from array

                var listInitCountries = new List<string>()
                // with initial value
                {
                        "Philippines",
                        "USA"
                };

                var listCountriesNew = new List<string>();
                listCountriesNew.Add("Philippines");
                listCountriesNew.Add("USA");
                listCountriesNew.AddRange(new[] { "Canada", "Australia" });
                listCountriesNew.Remove("Philippines");
                listCountriesNew[0] = "America";
                Console.WriteLine("LIST OF COUNTRIES");
                for (var index = 0; index < listCountriesNew.Count; index++)
                {
                        Console.WriteLine($"{index}: {listCountriesNew[index]}");
                }
        }
    }
}
```

LINQ (LANGUAGE INTEGRATED QUERY) & LAMBDA

LINQ IS A **QUERY**-BASED WHILE **LAMBDA** IS A **METHOD**-BASED.

"A **QUERY** IS A SET OF INSTRUCTIONS THAT DESCRIBES WHAT DATA TO RETRIEVE FROM A GIVEN DATA SOURCE (OR SOURCES) AND WHAT SHAPE AND ORGANIZATION RETURNED DATA SHOULD HAVE. A QUERY IS DISTINCT FROM THE RESULTS THAT IT PRODUCES."

LINQ IS USING QUERY EXPRESSION (SQL-LIKE QUERY)
LAMBDA IS USING LAMBDA EXPRESSION

1. LINQ'S QUERY EXPRESSION IS AUTOMATICALLY CONVERTED/TRANSLATED TO LAMBDA EXPRESSION DURING COMPILATION.
2. WITH LAMBDA-ONLY FUNCTIONS (NOT AVAILABLE IN LINQ)

PROGRAM FOR LINQ AND LAMBDA COMPARING USING LOOP AND IF-ELSE

// EVEN NUMBERS
USING LOOP AND IF-ELSE STATEMENT

```
var numbers = new[] { 5, 6, 2, 9, 1 };

// USING LOOP AND IF-ELSE
var evenNumbers = new List<int>();
foreach (var number in numbers)
{
        if (number % 2 == 0)
        {
                evenNumbers.Add(number);
        }
}
Console.WriteLine("Foreach and IF");
foreach (var even in evenNumbers)
        Console.WriteLine(even);
```

USING LINQ

```
var linqEvenNumbers = from number in numbers where number % 2 == 0 select number;
foreach (var even in linqEvenNumbers)
        Console.WriteLine(even);
```

USING LAMBDA

```
var lambdaEvenNumbers = numbers.Where(number => number % 2 == 0);
foreach (var even in lambdaEvenNumbers)
        Console.WriteLine(even);
```

// PRINT ONLY THOSE NUMBERS LESS THAN 9
USING LINQ

```
var misedNumbers = new[] { 3, 5, 6, 9 };
var linqQuery = from number in mixedNumbers
                where number < 9
                orderby number descending
                select $"numero: {number}";
Console.WriteLine("LINQ");
foreach (var num in linqQuery)
        Console.WriteLine(num);
```

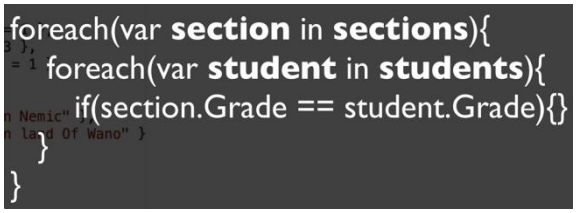
USING LAMBDA

```
var lambdaQuery = mixedNumbers
                        .Where(number => number < 9)
                        .OrderByDescending(number => number)
                        .Select(number => $"numero: {number}");
foreach (var num in lambdaQuery)
        Console.WriteLine(num);
```

// CLASS

```
public class Student
{
        public string Name { get; set;}
        public int Age { get; set;}
        public int Grade { get; set;}
}

public class Section
{
        public string SectionName { get; set; }
        public int Grade { get; set; }
}
{
        static void Main(string[] args)
        {
                var students = new List<Student>()
                {
                        new Student() { Name = "Jiro", Age = 32, Grade = 1 },
                        new Student() { Name = "Laurenz", Age = 25, Grade = 3 },
                        new Student() { Name = "Agad", Age = 30, Grade = 1 }
                };
                var sections = new List<Section>()
                {
                        new Section(){ Grade = 1, SectionName = "Section Nemic" },
                        new Section(){ Grade = 1, SectionName = "Section Land" },
                };
```




```
            //LINQ
            var query = from section in sections
                        join student in students on section.Grade equals
                        student.Grade
                        where student.Age > 20
                        orderby student.Age, student.Grade
                        select new
                        {
                            section.SectionName,
                            StudentName = student.Name
                        };
            foreach (var sectionAndStudent in query)
            Console.WriteLine($"Section {sectionAndStudent.SectionName},
            StudentName: {sectionAndStudent.StudentName}");
        }
}
```

STRING, DATETIME, NULLABLE & ENUMERATION(ENUM)

STRING - A **STRING** IS AN OBJECT OF TYPE STRING WHOSE VALUE IS TEXT. INTERNALLY, THE TEXT IS STORED AS A SEQUENTIAL READ-ONLY COLLECTION OF **CHAR** OBJECTS.

STRING VALUE OF "JIRO" IS EQUIVALENT TO **CHAR**[] { 'J', 'I', 'R', 'O' }

STRING OBJECTS ARE **IMMUTABLE**: THEY **CANNOT BE CHANGED** AFTER THEY HAVE BEEN CREATED. ALL OF THE STRING METHODS AND C# OPERATORS THAT APPEAR TO MODIFY/MANIPULATE A STRING ACTUALLY RETURN THE RESULTS IN A NEW STRING OBJECT.

FORMATTING:

- **COMPOSITE FORMATTING** - UTILIZES PLACEHOLDERS IN BRACES TO CREATE A FORMAT STRING.
- **STRING INTERPOLATION** - IDENTIFIED BY THE $ SPECIAL CHARACTER AND INCLUDE INTERPOLATED EXPRESSIONS IN BRACES.

Escape sequence	Character name
\'	Single quote
\"	Double quote
\\	Backslash
\0	Null
\a	Alert
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

DATETIME - IT REPRESENTS AN INSTANT IN TIME, TYPICALLY EXPRESSED AS A DATE AND TIME OF DAY. **DATETIME IS A STRUCT**.

STRUCT = VALUE TYPE
CLASS = REFERENCE TYPE

A CALCULATION USING A **DATETIME** STRUCTURE, SUCH AS **ADD** OR **SUBTRACT** DOES NOT MODIFY THE VALUE OF THE STRUCTURE. INSTEAD, THE CALCULATION RETURNS A NEW **DATETIME** STRUCTURE WHOSE VALUE IS THE RESULT OF THE CALCULATION.

NULLABLE - IT SUPPORTS A VALUE TYPE THAT CAN BE ASSIGNED NULL. THIS CALL CANNOT BE INHERITED.

A TYPE IS SAID TO BE **NULLABLE** IF IT CAN BE ASSIGNED A VALUE OR CAN BE ASSIGNED NULL, WHICH MEANS THE TYPE HAS NO VALUE WHATSOEVER. BY DEFAULT, ALL REFERENCE TYPES, SUCH AS **STRIN**G, ARE NULLABLE, BUT ALL VALUE TYPES, SUCH AS **Int32**, ARE NOT.

ENUMERATION - IT IS A SET OF NAMED CONSTANTS WHOSE UNDERLYING TYPE IS ANY INTEGRAL TYPE. IF NO UNDERLYING TYPE IS EXPLICITLY DECLARED, **Int32** IS USED.

EXAMPLE: FOR GENDER ENUMERATION, NAMED CONSTANTS WILL BE MALE, FEMALE & OTHERS.

FUNCTION/METHOD, RETURN TYPE, PARAMETERS(ref, out) & RECURSIVE

FUNCTION/METHOD - IT IS A CODE BLOCK THAT CONTAINS A SERIES OF STATEMENTS. A PROGRAM CAUSES THE STATEMENTS TO BE EXECUTED BY CALLING THE METHOD AND SPECIFYING ANY REQUIRED METHOD ARGUMENTS.

SYNTAX:
<Access Specifier><Return Type><Method Name>(Parameters)
{
    <Method Body>
}

ACCESS SPECIFIER - THIS DETERMINES THE VISIBILITY OF A VARIABLE OR A METHOD FROM ANOTHER CLASS.

RETURN TYPE - A METHOD MAY RETURN A VALUE. THE RETURN TYPE IS THE DATE TYPE OF THE VALUE THE METHOD RETURNS. IF THE METHOD IS NOT RETURNING ANY VALUES, THEN THE RETURN TYPE IS **VOID**.

METHOD NAME - METHOD NAME IS A UNIQUE IDENTIFIER AND IT IS CASE SENSITIVE. IT CANNOT BE SAME AS ANY OTHER IDENTIFIER DECLARED IN THE CLASS.

PARAMETERS - ENCLOSED BETWEEN PARENTHESIS, THE PARAMETERS ARE USED TO PASS AND RECEIVE DATA FROM A METHOD. THE PARAMETER LIST REFERS TO THE TYPE, ORDER, AND NUMBER OF THE PARAMETERS OF A METHOD. PARAMETERS ARE OPTIONAL; THAT IS, A METHOD MAY CONTAIN NO PARAMETERS.

METHOD BODY - THIS CONTAINS THE SET OF INSTRUCTIONS NEEDED TO COMPLETE THE REQUIRED ACTIVITY.

CLASS, STRUCT & DELEGATE

CLASS - A CLASS IS **REFERENCE TYPE**. A CLASS IS LIKE A BLUEPRINT OF A SPECIFIC OBJECT. A CLASS ENABLES YOU TO CREATE YOUR CUSTOM TYPES BY GROUPING VARIABLES OF OTHER TYPES, METHODS, AND EVENTS.

ACCESS MODIFER COULD BE PUBLIC, PRIVATE AND INTERNAL.

SYNTAX:
<access modifier> class <identifier/name>
{

- FIELDS
- CONSTRUCTORS

- PROPERTIES
- METHODS/FUNCTIONS
- EVENTS

}

STRUCT - A STRUCT IS **VALUE TYPE**. A STRUCT IS LIKE A BLUEPRINT OF A SPECIFIC OBJECT. A STRUCT ENABLES YOU TO CREATE YOUR CUSTOM TYPES BY GROUPING VARIABLES OF OTHER TYPES, METHODS, AND EVENTS.

SYNTAX:
<access modifier> struct <identifier/name>
{
- FIELDS
- CONSTRUCTORS
- PROPERTIES
- METHODS/FUNCTIONS
- EVENTS

}

DELEGATE - A DELEGATE IS A **REFERENCE TYPE**. A DELEGATE IS A TYPE THAT REPRESENTS REFERENCES TO METHODS WITH A PARTICULAR PARAMETER LIST AND RETURN TYPE. A DELEGATES ARE USED TO PASS METHODS AS ARGUMENTS TO OTHER METHODS.

SYNTAX:
<access modifier> delegate <return type> <identifier/name> (<parameters>);

DEBUGGING, ERROR HANDLING AND UNIT TESTING

DEBUGGING - IT IS THE PROCESS OF FIXING A BUG IN THE SOFTWARE. USUALLY, THE SOFTWARE CONTAINS ERRORS AND BUGS, WHICH ARE ROUTINELY REMOVED.

STEP 1 - IDENTIFY THE ERROR
STEP 2 - FIND THE ERROR LOCATION
STEP 3 - ANALYZE THE ERROR
STEP 4 - PROVE THE ANALYSIS
STEP 5 - COVER LATERAL DAMAGE
STEP 6 - FIX & VALIDATE

ERROR/ EXCEPTION HANDLING - IT IS A TYPE OF ERROR THAT OCCURS DURING THE EXECUTION OF AN APPLICATION. ERRORS ARE TYPICALLY PROBLEMS THAT ARE NOT EXPECTED.

C# EXCEPTION HANDLING IS DONE WITH THE FOLLOWING KEYWORDS:

try, catch, finally, and throw.

SYNTAX:

```
try
{
}
catch (<Type of Exception>)
{
}
finally
{
}
```

UNIT TESTING - TO CHECK THAT YOUR CODE IS WORKING AS EXPECTED BY CREATING AND RUNNING UNIT TESTS.

IT IS CALLED **UNIT TESTING** BECAUSE YOU BREAK DOWN THE FUNCTIONALITY OF YOUR PROGRAM INTO DISCRETE TESTABLE BEHAVIORS THAT YOU CAN TEST AS INDIVIDUAL UNITS.

COMMON UNIT TEST PROJECTS IN VISUAL STUDIO

- Xunit
- MSTest
- NUnit

OOP, ABSTRACT VS INTERFACE & GENERIC

OOP - IT IS A PROGRAMMING PARADIGM THAT RELIES ON THE CONCEPT CLASSES AND OBJECTS.

- **ABSTRACTION** - IT IS MODELING THE RELEVANT ATTRIBUTES AND INTERACTIONS OF ENTITIES AS CLASSES TO DEFINE AN ABSTRACT REPRESENTATION OF A SYTEM.
- **ENCAPSULATION** - IT IS HIDING THE INTERNAL STATE AND FUNCTIONALITY OF AN OBJECT AND ONLY ALLOWING ACCESS THROUGH A PUBLIC SET OF FUNCTIONS.
- **INHERITANCE** - IT IS THE ABILITY TO RECEIVE (“INHERIT”) METHODS AND PROPERTIES FROM AN EXISTING CLASS.
- **POLYMORPHISM** - IT IS ABILITY TO IMPLEMENT INHERITED PROPERTIES OR METHODS IN DIFFERENT WAYS ACROSS MULTIPLE ABSTRACTIONS.

ABSTRACT VS INTERFACE

- AN **ABSTRACT CLASS** ALLOWS YOU TO CREATE FUNCTIONALITY THAT SUBCLASSES CAN IMPLEMENT OR OVERRIDE.
- AN **INTERFACE** ONLY ALLOWS YOU TO DEFINE FUNCTIONALITY, NOT IMPLEMENT IT.

- AND WHEREAS AS CLASS CAN EXTEND ONLY ONE ABSTRACT CLASS, IT CAN TAKE ADVANTAGE OF MULTIPLE INTERFACES.

GENERIC

- **C#** ALLOWS YOU TO DEFINE GENERIC CLASSES, INTERFACES, ABSTRACT, FIELDS, METHODS, STATIC METHODS, PROPERTIES, EVENTS, DELEGATES, AND OPERATORS USING THE TYPE PARAMETER AND WITHOUT THE SPECIFIC DATE TYPE.
- A TYPE PARAMETER IS A PLACEHOLDER FOR A PARTICULAR TYPE SPECIFIED WHEN CREATING AN INSTANCE OF THE GENERIC TYPE.
- A GENERIC TYPE IS DECLARED BY SPECIFYING A TYPE PARAMETER IS AN ANGLE BRACKETS AFTER A TYPE NAME, e.g TypeName<T>WHERE T IS A TYPE PARAMETER.

SIMPLE NON-OOP VS OOP & SOLID PRINCIPLE

SOLID PRINCIPLE:

- **S** - **SINGLE RESPONSIBILITY**: A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE.
- **O** - **OPEN FOR EXTENSION**, CLOSED FOR MODIFICATION: SOFTWARE ENTITIES SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION.
- **L** - **LISKOV'S SUBSTITUTION**: A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE.
- **I** - **INTERFACE SEGREGATION**: A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE.
- **D** - **DEPENDENCY INVERSION** - HIGH-LEVEL MODULES/ SHOULD NOT DEPEND ON LOW-LEVEL MODULES/CLASSES BOTH SHOULD DEPEND UPON ABSTRACTIONS. ABSTRACT SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.