c# notes

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

.NET Framework runs on the Windows operating system, there are some alternative versions that work on other operating systems. **Mono** is an open-source version of the .NET Framework

C# program consists of the following parts −

* Namespace declaration
* A class
* Class methods
* Class attributes
* A Main method
* Statements and Expressions
* Comments

**-using** keyword is used to include the **System** namespace in the program

-**namespace** is a collection of classes.

-Classes generally contain multiple methods.

Methods define the behavior of the class.

-**Main** method, which is the **entry point** for all C# programs.

The **Main** method states what the class does when executed.

-/\*...\*/ multi-line comments

-// single line comments

• C# is case sensitive.

* + All statements and expression must end with a semicolon (;).
  + The program execution starts at the Main method.
  + Unlike Java, program file name could be different from the class name.

- In Object-Oriented Programming methodology, a program consists of various objects that interact with each other by means of actions. The actions that an object may take are called methods. Objects of the same kind are said to have the same type or, are said to be in the same class.

Functions are set of statements that perform a specific task

- identifier is a name used to identify a class, variable, function, or any other user- defined item. The basic rules for naming classes in C# are as follows −

* + - * A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.
      * It must not contain any embedded space or symbol such as? - + ! @ # % ^ & \* ( ) [ ] { } . ; : " ' / and \. However, an underscore ( \_ ) can be used.
      * It should not be a C# keyword.

- C# reserved keywords and contextual keywords

<https://www.tutorialspoint.com/csharp/csharp_basic_syntax.htm>

- variables in C#, are categorized into the following types −

* + - Value types
    - Reference types
    - Pointer types

-value types directly contain data. //data type

—Some examples are **int, char, and float**,

—get the exact size of a type or a variable on a particular platform, you can use the **sizeof** method. The expression *sizeof(type)* yields the storage size of the object or type in bytes.

-reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.

—If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value

—Example of **built-in** reference types are: **object**, **dynamic,** and **string**.

**Object Type** is the ultimate base class for all data types in C# Common Type System (CTS)

Object is an alias for System.Object class.

The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types.

value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

Dynamic Type

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

String Type

**String Type** allows you to assign any string values to a variable.

The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

Pointer Type

Pointer type variables store the memory address of another type.

Syntax for declaring a pointer type is −

type\* identifier;

For example,

char\* cptr;

int\* iptr;

Type conversion is converting one type of data to another type.

also known as Type Casting// In C#, type casting has two forms −

**• Implicit type conversion** − These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.

* + - * + **Explicit type conversion** − These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

C# provides the following built-in type conversion methods −

<https://www.tutorialspoint.com/csharp/csharp_type_conversion.htm>

ToString converts another data type to string

variable is nothing but a name given to a storage area that our programs can manipulate

C# also allows defining other value types of variable such as **enum** and reference types of variables such as **class**

Syntax for variable definition in C# is −

<data\_type> <variable\_list>;

int i, j, k;

The **Console** class in the **System** namespace provides a function **ReadLine()** for accepting input from the user and store it into a variable.

**• lvalue** − An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.

* + - **rvalue** − An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and hence they may appear on the left-hand side of an assignment. Numeric literals are rvalues and hence they may not be assigned and can not appear on the left-hand side.

constants refer to fixed values that the program may not alter during its execution.

Here are some examples of integer literals −

212 /\* Legal \*/

215u /\* Legal \*/

0xFeeL /\* Legal \*

85 /\* decimal \*/

0x4b /\* hexadecimal \*/

30 /\* int \*/

30u /\* unsigned int \*/

30l /\* long \*/

30ul /\* unsigned long \*/

escape sequences

<https://www.tutorialspoint.com/csharp/csharp_constants.htm>

Constants are defined using the **const** keyword.

const <data\_type> <constant\_name> = value;

operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

• Arithmetic Operators

* + - * Relational Operators
      * Logical Operators
      * Bitwise Operators
      * Assignment Operators
      * Misc Operators

Arithmetic Operators//Relational Operators//Logical Operators//Bitwise Operators// Assignment Operators//Miscellaneous Operators

<https://www.tutorialspoint.com/csharp/csharp_operators.htm>

**conditional operator ?** used to replace **if...else** statements.

decision making statements

<https://www.tutorialspoint.com/csharp/csharp_decision_making.htm>

Loops

<https://www.tutorialspoint.com/csharp/csharp_loops.htm>

**Encapsulation** is defined 'as the process of enclosing one or more items within a physical or logical package'.

Abstraction allows making relevant information visible and encapsulation enables a programmer to *implement the desired level of abstraction*.

Encapsulation is implemented by using **access specifiers**. An **access specifier** defines the scope and visibility of a class member. C# supports the following access specifier

* + - Public
    - Private
    - Protected
    - Internal
    - Protected internal

Public access specifier allows a class to expose its member variables and member functions to other functions and objects

Any public member can be accessed from outside the class

Private access specifier allows a class to hide its member variables and member functions from other functions and objects.

Only functions of the same class can access its private members

Protected access specifier allows a child class to access the member variables and member functions of its base class.

Internal access specifier allows a class to expose its member variables and member functions to other functions and objects in the current assembly.

default access specifier of a class member if we don't mention any? It is **private**.

protected internal access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application

method is a group of statements that together perform a task

To use a method, you need to −

* + - Define the method
    - Call the method

syntax for defining a method in C# is as follows −

<Access Specifier> <Return Type> <Method Name>(Parameter List) {

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 data 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.

**Parameter list** − Enclosed between parentheses, 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.

You can call a method using the name of the method

When method with parameters is called, you need to pass the parameters to the method. There are three ways that parameters can be passed to a method

|  |
| --- |
| [**Value parameters**](https://www.tutorialspoint.com/csharp/csharp_value_parameters.htm)  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |

|  |
| --- |
| [**Reference parameters**](https://www.tutorialspoint.com/csharp/csharp_reference_parameters.htm)  This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument. |
| [**Output parameters**](https://www.tutorialspoint.com/csharp/csharp_output_parameters.htm)  This method helps in returning more than one value. |

<https://www.tutorialspoint.com/csharp/csharp_methods.htm>

you can store any value from -2,147,483,648 to 2,147,483,647 or null in a Nullable<Int32> variable. Similarly, you can assign true, false, or null in a Nullable<bool> variable.

Syntax for declaring a **nullable** type is as follows −

< data\_type> ? <variable\_name> = null;

null coalescing operator is used with the nullable value types and reference types

used for converting an operand to the type of another nullable (or not) value type operand, where an implicit conversion is possible.

If the value of the first operand is null, then the operator returns the value of the second operand, otherwise it returns the value of the first operand.

double? num1 = null;

double? num2 = 3.14157;

double num3;

num3 = num1 ?? 5.34;

Console.WriteLine(" Value of num3: {0}", num3);

num3 = num2 ?? 5.34;

Console.WriteLine(" Value of num3: {0}", num3);

Console.ReadLine();

declare an array

datatype[] arrayName;

Declaring an array does not initialize the array in the memory. When the array variable i s initialized, you can assign values to the array.

Array is a reference type, so you need to use the **new** keyword to create an instance of the array. For example,

double[] balance = new double[10];

important concepts related to array which should be clear to a C# programmer

|  |
| --- |
| [**Multi-dimensional arrays**](https://www.tutorialspoint.com/csharp/csharp_multi_dimensional_arrays.htm)  C# supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. |
| [**Jagged arrays**](https://www.tutorialspoint.com/csharp/csharp_jagged_arrays.htm)  C# supports multidimensional arrays, which are arrays of arrays. |

|  |
| --- |
| [**Passing arrays to functions**](https://www.tutorialspoint.com/csharp/csharp_passing_arrays_to_functions.htm)  You can pass to the function a pointer to an array by specifying the array's name without an index. |

|  |
| --- |
| [**Param arrays**](https://www.tutorialspoint.com/csharp/csharp_param_arrays.htm)  This is used for passing unknown number of parameters to a function. |

|  |
| --- |
| [**The Array Class**](https://www.tutorialspoint.com/csharp/csharp_array_class.htm)  Defined in System namespace, it is the base class to all arrays, and provides various properties and methods for working with arrays. |

you can use strings as array of characters

You can create string object using one of the following methods −

* + By assigning a string literal to a String variable
  + By using a String class constructor
  + By using the string concatenation operator (+)
  + By retrieving a property or calling a method that returns a string
  + By calling a formatting method to convert a value or an object to its string representation

String class has the following two properties −

|  |  |
| --- | --- |
| **Sr.No.** | **Property & Description** |
| 1 | **Chars**  Gets the *Char* object at a specified position in the current *String* object. |
| 2 | **Length**  Gets the number of characters in the current String object. |

visit MSDN library for the complete list of methods and String class constructors.

structure is a value type data type // helps you to make a single variable hold related data of various data types.

**struct** keyword is used for creating a structure.

Structures are used to represent a record.

C# structures have the following features −

* Structures can have methods, fields, indexers, properties, operator methods, and events.
* Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and cannot be changed.
* Unlike classes, structures cannot inherit other structures or classes.
* Structures cannot be used as a base for other structures or classes.
* A structure can implement one or more interfaces.
* Structure members cannot be specified as abstract, virtual, or protected.
* When you create a struct object using the **New** operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator.
* If the New operator is not used, the fields remain unassigned and the object cannot be used until all the fields are initialized.

Classes and Structures have the following basic differences −

* classes are reference types and structs are value types
* structures do not support inheritance
* structures cannot have default constructor

enumeration is a set of named integer constants.

enumerated type is declared using the **enum** keyword.

The general syntax for declaring an enumeration is −

enum <enum\_name> {

enumeration list

};

Each of the symbols in the enumeration list stands for an integer value, one greater than the symbol that precedes it. By default, the value of the first enumeration symbol is 0.

When you define a class, you define a blueprint for a data type

objects are instances of a class

• Access specifiers specify the access rules for the members as well as the class itself. If not mentioned, then the default access specifier for a class type is **internal**. Default access for the members is **private**.

* + - Data type specifies the type of variable, and return type specifies the data type of the data the method returns, if any.
    - To access the class members, you use the dot (.) operator.
    - The dot operator links the name of an object with the name of a member.

A member function of a class is a function that has its definition or its prototype within the class definition similar to any other variable.

Member variables are the attributes of an object

These variables can only be accessed using the public member functions.

class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

**default constructor** does not have any parameter but if you need, a constructor can have parameters.

Such constructors are called **parameterized constructors**.

A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope.

A **destructor** has exactly the same name as that of the class with a prefixed tilde (~) and it can neither return a value nor can it take any parameters.

We can define class members as static using the **static** keyword.

no matter how many objects of the class are created, there is only one copy of the static member.

Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it.

You can also declare a **member function** as **static**

Such functions can access only static variables. The static functions exist even before the object is created.

One of the most important concepts in object-oriented programming is inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **IS-A** relationship. For example, mammal **IS A** animal, dog **IS-A** mammal hence dog **IS-A** animal as well, and so on.

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base classes or interfaces.

The derived class inherits the base class member variables and member methods.

Therefore the super class object should be created before the subclass is created.

**C# does not support multiple inheritance**.

**polymorphism** means having many forms.

In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

**static polymorphism**, the response to a function is determined at the compile time

**dynamic polymorphism**, it is decided at run-time.

The mechanism of linking a function with an object during compile time is called early binding.

also called static binding.

C# provides two techniques to implement static polymorphism. They are −

* Function overloading
* Operator overloading

You can have multiple definitions for the same function name in the same scope.

definition of the function must differ from each other by the types and/or the number of arguments in the argument list.

C# allows you to create abstract classes that are used to provide partial class implementation of an interface.

Implementation is completed when a derived class inherits from it.

**Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

rules about abstract classes −

* You cannot create an instance of an abstract class
* You cannot declare an abstract method outside an abstract class
* When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed.

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions

virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

**Overloading**

Overloaded operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined.

overloaded operator has a return type and a parameter list.

The following table describes the overload ability of the operators in C# −

|  |  |
| --- | --- |
| **Sr.No.** | **Operators & Description** |
| 1 | **+, -, !, ~, ++, --**  These unary operators take one operand and can be overloaded. |
| 2 | **+, -, \*, /, %**  These binary operators take one operand and can be overloaded. |
| 3 | **==, !=, <, >, <=, >=**  The comparison operators can be overloaded. |
| 4 | **&&, ||**  The conditional logical operators cannot be overloaded directly. |
| 5 | **+=, -=, \*=, /=, %=**  The assignment operators cannot be overloaded. |
| 6 | **=, ., ?:, ->, new, is, sizeof, typeof**  These operators cannot be overloaded. |

**Interfaces**

interface is defined as a syntactical contract that all the classes inheriting the interface should follow.

The interface defines the **'what'** part of the syntactical contract and the deriving classes define the **'how'** part of the syntactical contract.

Interfaces define properties, methods, and events, which are the members of the interface.

Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members.

Abstract classes to some extent serve the same purpose, however, they are mostly used when only few methods are to be declared by the base class and the deriving class implements the functionalities.

Interfaces are declared using the interface keyword. It is similar to class declaration. Interface statements are public by default.

**Namespaces**

**namespace** is designed for providing a way to keep one set of names separate from another.

class names declared in one namespace does not conflict with the same class names declared in another.

namespace definition begins with the keyword **namespace** followed by the namespace name

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows

namespace\_name.item\_name;

**using** keyword states that the program is using the names in the given namespace.

avoid prepending of namespaces with the **using** namespace directive.

directive tells the compiler that the subsequent code is making use of names in the specified namespace

You can define one namespace inside another namespace as follows

namespace namespace\_name1 {

// code declarations

namespace namespace\_name2 {

// code declarations

}

}

You can access members of nested namespace by using the dot (.) operator as follows

**Preprocessor Directives**

preprocessor directives give instruction to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not statements, so they do not end with a semicolon (;).

directives are processed as if there was one.

the preprocessor directives are used to help in conditional compilation.

Unlike C and C++ directives, they are not used to create macros. A preprocessor directive must be the only instruction on a line.

Preprocessor Directives in C#

<https://www.tutorialspoint.com/csharp/csharp_preprocessor_directives.htm>

**Regular Expressions**

**regular expression** is a pattern that could be matched against an input text.

There are various categories of characters, operators, and constructs that lets you to define regular expressions. Click the following links to find these constructs.

* [Character escapes](https://www.tutorialspoint.com/csharp/csharp_character_escapes.htm)
* [Character classes](https://www.tutorialspoint.com/csharp/csharp_character_classes.htm)
* [Anchors](https://www.tutorialspoint.com/csharp/csharp_anchors.htm)
* [Grouping constructs](https://www.tutorialspoint.com/csharp/csharp_grouping_constructs.htm)
* [Quantifiers](https://www.tutorialspoint.com/csharp/csharp_quantifiers.htm)
* [Backreference constructs](https://www.tutorialspoint.com/csharp/csharp_backreference_constructs.htm)
* [Alternation constructs](https://www.tutorialspoint.com/csharp/csharp_alternation_constructs.htm)
* [Substitutions](https://www.tutorialspoint.com/csharp/csharp_substitutions.htm)
* [Miscellaneous constructs](https://www.tutorialspoint.com/csharp/csharp_miscellaneous_constructs.htm)

Regex class is used for representing a regular expression.

has the following commonly used methods −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **public bool IsMatch(string input)**  Indicates whether the regular expression specified in the Regex constructor finds a match in a specified input string. |
| 2 | **public bool IsMatch(string input, int startat)**  Indicates whether the regular expression specified in the Regex constructor finds a match in the specified input string, beginning at the specified starting position in the string. |
| 3 | **public static bool IsMatch(string input, string pattern)**  Indicates whether the specified regular expression finds a match in the specified input string. |
| 4 | **public MatchCollection Matches(string input)**  Searches the specified input string for all occurrences of a regular expression. |
| 5 | **public string Replace(string input, string replacement)**  In a specified input string, replaces all strings that match a regular expression pattern with a specified replacement string. |
| 6 | **public string[] Split(string input)**  Splits an input string into an array of substrings at the positions defined by a regular expression pattern specified in the Regex constructor. |

**Exception Handling**

exception is a problem that arises during the execution of a program.

C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

**• try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.

* + - * **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
      * **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
      * **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.

A try/catch block is placed around the code that might generate an exception.

Code within a try/catch block is referred to as protected code,

**System.ApplicationException** class supports exceptions generated by application programs.

**System.SystemException** class is the base class for all predefined system exception.

following table provides some of the predefined exception classes derived from the Sytem.SystemException class −

|  |  |
| --- | --- |
| **Sr.No.** | **Exception Class & Description** |
| 1 | **System.IO.IOException**  Handles I/O errors. |
| 2 | **System.IndexOutOfRangeException**  Handles errors generated when a method refers to an array index out of range. |
| 3 | **System.ArrayTypeMismatchException**  Handles errors generated when type is mismatched with the array type. |
| 4 | **System.NullReferenceException**  Handles errors generated from referencing a null object. |
| 5 | **System.DivideByZeroException**  Handles errors generated from dividing a dividend with zero. |
| 6 | **System.InvalidCastException**  Handles errors generated during typecasting. |
| 7 | **System.OutOfMemoryException**  Handles errors generated from insufficient free memory. |
| 8 | **System.StackOverflowException**  Handles errors generated from stack overflow. |

User-defined exception classes are derived from the **Exception** class.

You can throw an object if it is either directly or indirectly derived from the **System.Exception** class.

**File I/O**

**file** is a collection of data stored in a disk with a specific name and a directory path

When a file is opened for reading or writing, it becomes a **stream**.

stream is basically the sequence of bytes passing through the communication path.

two main streams: the **input stream** and the **output stream**

**input stream** is used for reading data from file (read operation)

**output stream** is used for writing into the file (write operation).

following table shows some commonly used non-abstract classes in the System.IO namespace −

|  |  |
| --- | --- |
| **Sr.No.** | **I/O Class & Description** |
| 1 | **BinaryReader**  Reads primitive data from a binary stream. |
| 2 | **BinaryWriter**  Writes primitive data in binary format. |
| 3 | **BufferedStream**  A temporary storage for a stream of bytes. |
| 4 | **Directory**  Helps in manipulating a directory structure. |
| 5 | **DirectoryInfo**  Used for performing operations on directories. |
| 6 | **DriveInfo**  Provides information for the drives. |
| 7 | **File**  Helps in manipulating files. |
| 8 | **FileInfo**  Used for performing operations on files. |
| 9 | **FileStream**  Used to read from and write to any location in a file. |
| 10 | **MemoryStream**  Used for random access to streamed data stored in memory. |
| 11 | **Path**  Performs operations on path information. |
| 12 | **StreamReader**  Used for reading characters from a byte stream. |
| 13 | **StreamWriter**  Is used for writing characters to a stream. |
| 14 | **StringReader**  Is used for reading from a string buffer. |
| 15 | **StringWriter**  Is used for writing into a string buffer. |

**FileStream** class in the System.IO namespace helps in reading from, writing to and closing files.

need to create a **FileStream** object to create a new file or open an existing file.

**syntax for creating a FileStream object is as follows −**

FileStream <object\_name> = new FileStream( <file\_name>, <FileMode Enumerator>,

<FileAccess Enumerator>, <FileShare Enumerator>);

example, we create a FileStream object **F** for reading a file named **sample.txt as shown** −

FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read,

FileShare.Read);

Commands

<https://www.tutorialspoint.com/csharp/csharp_file_io.htm>

Advanced File Operations in C#

<https://www.tutorialspoint.com/csharp/csharp_file_io.htm>

Advanced Section

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in your program.

questions

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

what is .net framework?

The .Net framework applications are multi-platform applications. The framework has been designed in such a way that it can be used from any of the following languages: C#, C++, Visual Basic, Jscript, COBOL, etc. All these languages can access the framework as well as communicate with each other.

compile time vs runtime?

whats Literals?

double? num3 = new double?(); ——— Whats with the question marks?

looks like it can stand for null

look up

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

few important features of C# −

* + - Boolean Conditions
    - Automatic Garbage Collection
    - Standard Library
    - Assembly Versioning
    - Properties and Events
    - Delegates and Events Management
    - Easy-to-use Generics
    - Indexers
    - Conditional Compilation
    - Simple Multithreading
    - LINQ and Lambda Expressions
    - Integration with Windows

Visual Web Developer

classe’s methods, objects