C# (Mr. Bangar Raju)

[ <https://www.youtube.com/watch?v=0u9k-kOR3KE&list=PLVlQHNRLflP-jc5Fbhfdhzv52AWYq836j> ]

* Constructor & Types of Constructor, Why Constructor?
* Variable vs Instance vs Reference
* Access Specifier
* Different kinds of Variables
* Inheritance
* Polymorphism (Method overloading, Method Overriding, Method Hiding, Operator overloading)
* Abstraction
* Interface
* Structure
* Enumeration
* Properties
* Indexers
* Delegate
* Extension Method
* String vs StringBuilder
* Exception & Exception handling
* Multithreading
* Collection
* LINQ to SQL

**C# Feature**

* Object Oriented

|  |  |  |
| --- | --- | --- |
|  | **Procedure Oriented Programming (POP)** | **Object Oriented Programming (OOP)** |
| Basic | Procedure/Structure Oriented | Object Oriented |
| Basis | How to get the task done | Data security |
| Division | Large program is divided into units called function | Entire program is divided into object |
| Accessing  mode | No Access specifier | Public, Protected, Private |
| Overloading/Polymorphism | No | Yes |
| Inheritance | No | Yes |
| Data hiding & security | There is no proper way of hiding the data, so data is insecure | Data hiding is in three model public, private and protected, hence data is secure |
| Example | C, VB, Fortran, Pascal | C#, C++, Java |
| Approach | Top-Down | Down-Bottom |

* Platform Independent

**#Constructor**: -

* It is special method present in class responsible for initializing the variables of that class.
* The name of the constructor method is exactly same name of the class in which it was present. Its non-value returning method.
* Each and every class requires this constructor if we want to create instance of that class

Example:



* Its responsibility of programmer to define the constructor under his class and if he fails to do so, on the behalf of programmer an implicit constructor gets defined in that class by the compiler



* Implicitly defined constructors are **parameter less** and these constructors are also known as **Default Constructor.**
* Implicitly defined constructors are public.
* We can also define the constructor under the class and if we defined it we can calls as **explicit constructor** and explicit constructors can be **parameter less or parameterized** also.
* **Syntax**

[<modifiers>] <Name> ([<parameter list>])

{

-Statements

}

* **Defining**: Implicit or Explicit.
* **Calling**: Implicit only.

**#Types of Constructor:**

1. Default or Parameter less Constructor
2. Parameterized constructor
3. Copy constructor
4. Static Constructor

* If constructor method doesn’t take any parameters, then we call that as **default or parameter less constructor**. These constructors can be defined by a programmer Explicitly or else will be default implicitly provided there is no explicit constructor under the class.
* **Parameterized Constructor:** If a constructor method is defined with any parameters we call that as parameterized constructor and these constructors can be defined by the programmers only but never can be defined implicitly.
* **Copy Constructor:** If we want to create multiple instances with the same values then we use these copy constructors, in a copy constructor the constructor takes the same class as parameter to it.
* **Static Constructor:** If a constructor is explicitly declared by using static modifier we called that as static constructor. All the constructor we have defined till now are non-static or instance constructor.
* If a class contains any static variable, then only implicit static constructor will be present or else we need to define them explicitly where as non-static constructors will be implicitly defined in every class (except static class) provided we did not define them explicitly
* Static constructors are responsible in initializing static variables and these constructors are never called explicitly they are implicitly called and over these constructers are first to execute under any class
* Static constructors can’t be parametrized so overloading static constructors not possible.

**#Why Constructors?**

* Every class requires a constructor to be present in it if we want to create the instance of that class.
* Every class contains and implicit constructor if not defined explicitly and with the help of that implicit constructor instance of class can be created.
* What is the need of defining constructor explicitly again?
* Implicit constructor of a class will initialize variables of class with the same value even if we create multiple instances of that class.
* If we define constructors explicitly with parameters, then we get chance of initializing the fields or variables of the class with new value every time we are going to create instance of that class.
* Whenever we define class first identify whether if the class variables require some values to execute and if they are required then define constructor explicitly and pass values through that constructors, so every time the instance of class is created we get chance of passing new values.
* Note: Generally, every class requires some values for extension and the values that is required for class to execute are always sent to that class by using the constructor only.

**#Static Constructor vs Non Static Constructors:**

* If a constructor is explicitly declared by using a static modifier we call that constructor as static constructor whereas rest of other are non-static constructor only.
* Constructors are responsible for initialization fields/variables of a class, so static fields are initialized by static constructors and non-static fields are initialized by non-static constructors.
* Static constructors are implicitly called where as non-static constructor must be explicitly called.
* Static constructor executes immediately once the execution of a class starts. And more over it’s the first block of code to run under the class whereas non-static constructors execute only after creating the instance of class as well as each and every time the instance of class is created.
* In the life cycle of class static constructors executes one and only one time. Whereas non-static constructor executes for zero times if no instances are created and “n” times if “n” instances are created.
* Non-static constructors can be parameterized but static constructors can’t have any parameters. Because static constructors are explicitly called and moreover it’s the first block of code to run under the class.
* Non-Static constructors can be overloaded whereas static constructors can’t be overloaded.
* Every class contains an implicit constructor if not defined explicitly and those implicit constructors are defined based on the following criteria.
* Every class except a static class contains an implicit non-static constructor if not defined with any explicit constructors.
* Static Constructors are implicitly defined only if that class contains any static fields or else that constructor will not be present at all.

**#Variable Instance and Reference:**

* **Class:** It’s user defined type



* **#Variable of a class**

Copy of class which is initialized.

* **#Instance of a class**

A copy of class that is initialized by *using* keyword. Which has its own memory and never shared with another instance.

* **#Reference of a class**

a copy of class that is initialized by using an existing instance and references of class will not have any memory allocation they will be sharing the same instance that assigned for initializing the variable.

Reference of class can be called as pointer to the instance and every modification we perform on the members using instance reflects when we access those members through reference and vice-versa.

**#Access Specifiers:**

* It’s special type of modifiers using which we can define the scope of type and its members.
* Private
* Internal
* Protected
* Protected Internal
* Public

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Cases** | **Private** | **Internal** | **Protected** | **Protected Internal** | **Public** |
| **Case 1** | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Case 2** | ✗ | ✓ | ✓ | ✓ | ✓ |
| **Case 3** | ✗ | ✓ | ✗ | ✓ | ✓ |
| **Case 4** | ✗ | ✗ | ✓ | ✓ | ✓ |
| **Case 5** | ✗ | ✗ | ✗ | ✗ | ✓ |

**#Different kinds of variables:**

* Non-Static
* Static
* Constants
* ReadOnly
* If a variable is explicitly declared by using the static modifier or else if a variable is declared under any static block, then those variables are static whereas rest of the other are non-static



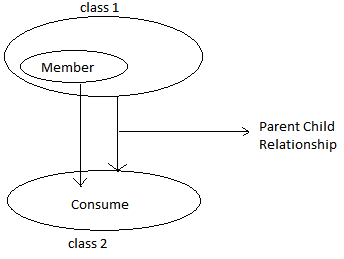
* Note: static member of class doesn’t require the instance of class for initialization or execution also, whereas non-static member of class require the instance of class both for initialization and execution.
* Static variable of class are initialized once the execution of class starts whereas instance variable is initialized only after creating class instance as well as each and every time the instance of class is created.
* In the life cycle of a class a static variable is initialized one and only one time, whereas instance variables are initialized for zero times if no instances are created and n times if n instances are created.
* Initialization of non-static variables is associated with instance creation and constructor calling. So non-static variable can be initialized through the constructor also.
* If a variable is declared by using the keyword “const” we call it as a constant variable and these constant variable can’t modify once after their declaration, so it’s must to initialize constant variables at the time of declaration only

Ex: float pi = 3.14f;

* The behavior of constant variables will be similar to the behavior of static variables i.e. initialized one and only one time in the life cycle of a class and doesn’t require the instance of class for accessing or initializing.
* The only difference between a static and constant variable is static variables can be modified but constant variable can’t be modified.
* If a variable is declared using “readonly” keyword we call that variable as a readonly variable and these variable also can’t be modified like constant but after initialization. It’s not compulsory to initialize a readonly variable at the time of declaration, they can also be initialized under constructor.
* The behavior of readonly variable will be similar to the behavior of non-static variable, i.e. initialized only after creating the instance of class and once for each instance of the class created.
* The only difference between readonly and instance variable is instance variables can be modified but not readonly variables.
* Constant variables are a fixed value for the whole class whereas readonly variable is a fixed value specific to an instance of class.

**#Inheritance**

* It’s a mechanism of consuming a member of one class in another class by establishing parent/child relationship between the classes. Which provides reusability.



* Example:

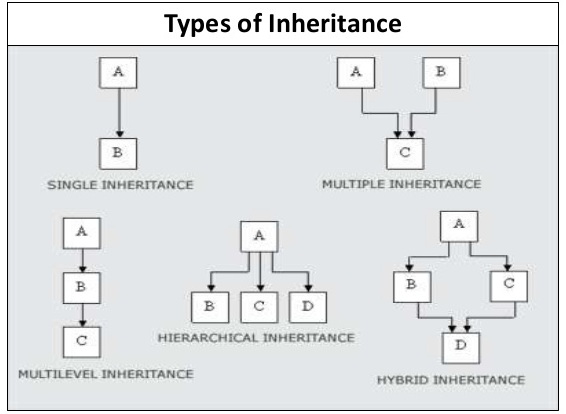


* [<Modifiers>] class <child class>: <parent class>
* A => Parent/Base/Super
* B => Child/Derived/Sub
* Note: In inheritance child class can consume members of its parent class as it is the owner of those members (except private member of parent).
* Important things about Inheritance

1. Parent classes constructor must be accessible to child class otherwise inheritance will not be possible
2. In inheritance child class can access parent classes but parent classes can never access any member that is purely defined under the child class.
3. We can initialize a parent classes variables by using the child class instance to make it as references. So that the reference will be consuming the memory of child class instance but in this case also we can’t call any pure child class members by using the reference.
4. Every class that is defined by us or predefined in the libraries of the language has a default parent class i.e. **Object class** of system namespace so the members of object class (Equals, GetHashCode, GetType, ToString) are accessible from anywhere.

Types of Inheritance: No. of parent classes a child can have or the no of child classes can have.

* 1. Single
  2. Multi-Level
  3. Hierarchical
  4. Hybrid
  5. Multiple



1. In c# we don’t have support for multiple inheritance through classes, what we are provided is only single inheritance through classes.
2. In the first point we learnt whenever child class instances are created child class constructor will implicitly call its parent classes constructor but only if the constructor is parameter less, whenever as if the constructor of parent class is parameterized child class constructor can’t implicitly call its parents constructor so to overcome to problem it is responsibility of programmer to explicitly call parent classes constructor from child class constructor and pass value to those parameters. To call parents constructor from child class we need to use the **base** keyword.



* Entity: it’s a living or non-living object associated with a set of attributes.

**Step 1:** identify the entities that are associated with the application we are developing.

Ex: School Application: Student, Teaching staff, Non-Teaching Staff.

**Step 2:** identify the attributes of each and every entity.

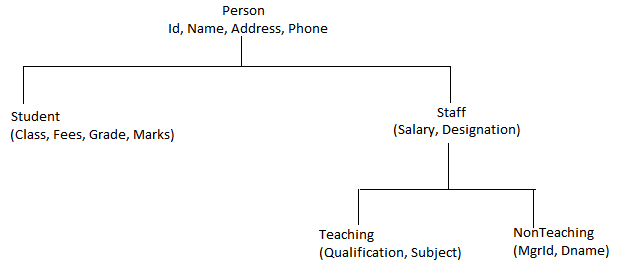
Ex:

Student: Id, Name, Address, Phone, Class, Marks, Grade, Fees

Teaching staff: Id, Name, Address, Phone, Designation, Marks, Salary, Qualification, Subject

Non-teaching staff: Id, Name, Address, Phone, designation, Salary, Department, Manager

**Step 3:** identify the common attributes of each entity and put them in a hierarchal order.

****

**Step 4:** Define the class representing the entities that are put in hierarchal order



**#Polymorphism:**

* Method Overloading
* Method Overriding

It’s approach of re-implementing a parent classes method under the child class with the same signature.

* **Overloading**:
* In this case we define multiple methods with the same name by changing their parameters.
* This can be performed either within a class as well as between parent child classes also.
* While overloading the parent classes methods under the child class, child class doesn’t require to take any permission from the parent class.
* Overloading is all about ***defining multiple behavior*** to a method.
* **Overriding**:
* Method overriding is an approach of re-implementing a parent classes method under the child class exactly with the same name and signature.
* In this case we define multiple methods with the same name and same parameter.
* This can be performing only between parent child classes can never be performed within the same class
* While overriding the parent method under child class child class requires permission from its parent
* **Note**: If we want override a parent’s method under child class first that method should be declared by using the virtual modifier in parent class
* Any virtual method of the parent class *can be* overridden by the child class if required by using the “**override**” modifier
* Overriding is all about ***changing the behavior*** of parent method under child class.
* **Method** **Hiding**:
* Method hiding/Shadowing is also an approach of re-implementing a parent classes method under the child class exactly with the same name and signature.
* In the 1st case child class re-implements its parent classes methods which are declared as virtual. Where as in the 2nd case child class can re-implement any parent’s method even if the method is not declared as virtual.
* We can re-implement a parent classes method under child class by using 2 approach

1. Method Overloading
2. Method Hiding/Shadowing

* After re-implementing parent classes methods under child classes the child class instance will start calling the local methods only i.e. re-implemented methods, but if required in any case we can also call the parent classes methods from child classes methods from child classes by using 2 approaches.

1. By creating the instance of parent classes under child class we can call parent’s methods from child class.
2. By using the **base** keyword also, we can call parents method from child class but keywords like *this* and *base* can’t be used from static block.

* A parent class reference even if created by using the child class instance cannot access any member that are purely defined under the child class but can call overridden members of child class, because overridden members are not considered as pure child class members which are re-implemented by using the approach of hiding are considered as pure child class members and not accessible to parent’s reference.
* **Operator Overloading:**
* Method overloading is approach of defining multiple behaviors to a method and those behaviors will vary based on the parameters of the method.
* Operator overloading is also an approach of defining multiple behaviors to an operator and those behaviors will vary based on the operand types between which the operator is used. For example, + is an addition operator when used between 2 numeric operands and it is a concatenation operator when used between two string operands.

Numbers **+** Numbers => Addition

String **+** String => Concatenation

* **Example:**

public static int operator +(int a, int b)

public static int operator -(int a, int b)

public static bool operator >(int a, int b)

public static bool operator +(string a, string b)

public static bool operator ==(string a, string b)

public static bool operator !=(string a, string b)

* **Syntax:**

[<Modifiers>] static <return type> operator <opt>(<operand types>)

{

--Logic

}

**#Abstraction:**

* **Abstract** **Method**: a method without any method body is known as abstract method.
* **Abstract** **Class**: a class which contains any abstract members in it is known as abstract class

**Entities**: Rectangle, Circle, Triangle, Cone

Rectangle: Width, Height

Circle: Radius, Pi

Triangle: Width, Height

Cone: Radius, Height, Pi

Width, Height, Radius, Pi



**#Interface**:

* **Class**: it is user defined datatype
* **Interface**: It is also a user defined datatype
* **Abstract** **Class**: Non-abstract methods (Method with method body) and also abstract methods (method without method body)
* **Interface**: it contains only abstract methods (Method without method body)
* **Note**: Every abstract method of an interface should be implemented by the child class of the interface without fail (Mandatory)
* Generally, class inherits from another class to consume the members of its parent, where as if class is inherited from an interface it is to implement the member of its parent.
* Note: A class can inherit from a class & interface at a time.
* Syntax:

[<modifiers>] class <Name>

{

--Members here

}

[<modifiers>] interface <Name>

{

--abstract member declaration here

}

* The default scope for the members of the interface is public where as its private incase of class.
* By default, every method/member of interface is abstract so we don’t require to use abstract modifier on it again just like we do in case of abstract class.
* Interface cannot contain any fields/variables.
* If required, Interface can inherit from another interface.
* Every member of interface should be implemented under the child class of the interface without fail, but while implementing we don’t require to use override modifier just like we have done in case of abstract class.

Example:



* What is an Interface?

An interface is also user defined type like class but can contain only abstract methods in it.

* How to define Interface?

[<modifiers>] Interface <Name>

{

--Abstract member declaration

}

* Multiple inheritance with Interface?

Types of Interface

1. Single
2. Multiple
3. Hierarchical
4. Multiple
5. Hybrid

* Even if multiple inheritance is not supported through classes in c#, it is still supported through interface
* Class can have one and only one immediate parent class. Whereas the same class can have any no. of interfaces as its parent i.e. multiple inheritance is supported in c# through interface.
* Why multiple inheritance is not supported through classes & how it is supported through interface?

***Ambiguity***

* Abstract Classes & Abstract Method
* Method without method body is called as abstract method what the method contain is only declaration of the method
* Ex:



* If Method is declared as abstract under any class, then the child class of that class is responsible for implementing the method without fail
* The concept of abstract method will be nearly similar to concept of method overloading

**Method Overriding**:



Abstract Method



* Abstract class:
* Abstract Method
* Non-Abstract Method
* Child class of abstract class
* Implement each & every abstract method of parent class
* Now only we can consume non-abstract method of parent class.
* A class under which abstract method is defined is known as abstract class.
* Note: to define a method or class as abstract we need to use abstract keyword on them
* We cannot create the instance of abstract class or interface but we can create reference of that

**#Structure:**