C#

.Net

Keshav Singh

@keshavsingh4522

Contents

[Topics 2](#_Toc137401930)

[Basics of C# 2](#_Toc137401931)

[Solid principal 2](#_Toc137401932)

[Data Types 7](#_Toc137401933)

[Keywords 8](#_Toc137401934)

[Namespaces 8](#_Toc137401935)

[Comments 9](#_Toc137401936)

[Operators 9](#_Toc137401937)

[Task 10](#_Toc137401938)

[Cancellation Token 10](#_Toc137401939)

[Methods 13](#_Toc137401940)

[OOP 14](#_Toc137401941)

[RelationShip 14](#_Toc137401942)

[Association 15](#_Toc137401943)

[Composition 15](#_Toc137401944)

[Aggregation 16](#_Toc137401945)

[Inheritance 17](#_Toc137401946)

# Topics

1. [Data Types](https://www.tutlane.com/tutorial/csharp/csharp-data-types-with-examples" \o "Data Types in C# with Examples" \t "_blank)
2. Keywords
3. Namespace
4. Comments
5. Operators
6. [Conditional Statements](https://www.tutlane.com/tutorial/csharp/csharp-if-statement-with-examples" \o "Conditional Statements in C#" \t "_blank)
7. [Arrays](https://www.tutlane.com/tutorial/csharp/csharp-arrays-with-examples" \o "Arrays in C# with Examples" \t "_blank)
8. [Access Modifiers](https://www.tutlane.com/tutorial/csharp/csharp-access-modifiers-public-private-protected-internal" \o "Access Modifiers in C# with Examples" \t "_blank)
9. [Classes and Objects](https://www.tutlane.com/tutorial/csharp/csharp-classes-and-objects-with-examples" \o "Classes and Objects in C# with Examples" \t "_blank)
10. [Constructors](https://www.tutlane.com/tutorial/csharp/csharp-constructors-with-examples" \o "Constructor in C# with Examples" \t "_blank)
11. [Structures](https://www.tutlane.com/tutorial/csharp/csharp-structures-structs" \o "Structures in C# with Examples" \t "_blank)
12. [Inheritance](https://www.tutlane.com/tutorial/csharp/csharp-inheritance" \o "Inheritance in C# with Examples" \t "_blank)
13. [Events](https://www.tutlane.com/tutorial/csharp/csharp-events" \o "Events in C# with Examples" \t "_blank)
14. [Delegates](https://www.tutlane.com/tutorial/csharp/csharp-delegates" \o "Delegates in C# with Examples" \t "_blank)
15. [Collections](https://www.tutlane.com/tutorial/csharp/csharp-collections" \o "Collections in C# with Examples" \t "_blank)
16. [Generics](https://www.tutlane.com/tutorial/csharp/csharp-generics" \o "Generics in C# with Examples" \t "_blank) and many more topics in c#.

# Basics of C#

## Solid principal

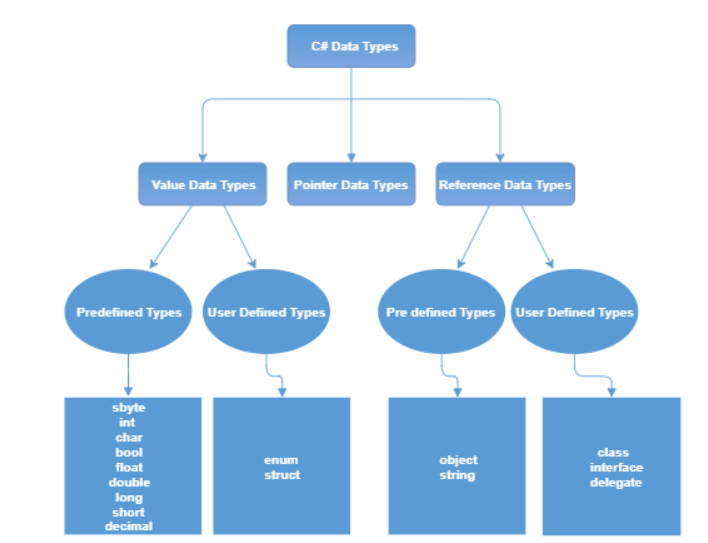
References -> <https://www.tutorialsteacher.com/csharp/solid-principles>

|  |  |
| --- | --- |
| 1. Single Responsibility Principle (SRP)  * A class should have only one responsibility. * **A class should have only one reason to change.** * This principle helps in achieving separation of concerns in a software system. | // Bad example: Combining multiple responsibilities in a single class  public class Employee  {  public void CalculateSalary() { /\*...\*/ }  public void SaveEmployee() { /\*...\*/ }  }  // Good example: Separating responsibilities into different classes  public class SalaryCalculator  {  public void CalculateSalary() { /\*...\*/ }  }  public class EmployeeRepository  {  public void SaveEmployee() { /\*...\*/ }  } |
| 1. Open/Closed Principle (OCP)  * **Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.** * This means that the behavior of a class can be extended without modifying its source code. | // Bad example: Modifying the existing class to add new functionality  public class Rectangle  {  public double Width { get; set; }  public double Height { get; set; }  }  public class AreaCalculator  {  public double CalculateArea(Rectangle[] rectangles)  {  double area = 0;  foreach (var rectangle in rectangles)  {  area += rectangle.Width \* rectangle.Height;  }  return area;  }  }  // Good example: Using inheritance to extend the functionality without modifying the existing class  public abstract class Shape  {  public abstract double CalculateArea();  }  public class Rectangle : Shape  {  public double Width { get; set; }  public double Height { get; set; }  public override double CalculateArea()  {  return Width \* Height;  }  } |
| 1. Liskov Substitution Principle (LSP)  * **Subtypes must be substitutable for their base type.**   **Example:**   * A derived class must be correctly substitutable for its base class. * When you derived a class from a base class then the derived class should correctly implement all the methods of the base class. * It should not remove some methods by throwing **NotImplementedException**   --- Example.  The example violates the Liskov Substitution principle because the MyReadOnlyCollection class implements the IMyCollection interface but it throws NotImplementedException for two methods Add() and Remove() because the MyReadOnlyCollection class is for the read-only collection so you cannot add or remove any item. LSP suggests that the subtype must be substitutable for the base class or base interface. In the above example, we should create another interface for read-only collection without Add() and Remove() methods. |  |
| 1. Interface Segregation Principle (ISP)  * **Clients should not be forced to depend on methods they do not use.** * This principle helps in creating focused and specialized interfaces, which are easier to understand and implement. | // Bad example: A single interface with multiple responsibilities  public interface IWorker  {  void Work();  void Sleep();  }  // Good example: Splitting the interface into specialized interfaces  public interface IWork  {  void Work();  }  public interface ISleep  {  void Sleep();  }  public class Human : IWork, ISleep  {  public void Work() { /\*...\*/ }  public void Sleep() { /\*...\*/ }  } |
| 1. [Dependency Inversion Principle](https://www.tutorialsteacher.com/csharp/dependency-inversion-principle)  * **High-level modules should not depend on low-level modules. Both should depend on abstraction**   High Level Module   * It is a module (class) that uses other modules (classes) to perform a task.   Low Level Module / Concrete Class   * It contains a detailed implementation of some specific task that can be used by other modules.   The high-level modules are generally the core business logic of an application whereas the low-level modules are input/output, database, file system, web API, or other external modules that interact with users, hardware, or other systems.   * Abstraction is something that is not concrete. * Abstraction should not depend on detail but details should depend on abstraction. * For example, an abstract class or interface contains methods declarations that need to be implemented in concrete classes. Those concrete classes depend on the abstract class or interface but not vice-versa. | Example: Classes without DIP  public class Student  {  public int StudentId { get; set; }  public string FirstName { get; set; }  public string LastName { get; set; }  public DateTime DoB { get; set; }  //tight coupling  private StudentRepository \_stdRepo = new StudentRepository();  public Student()  {  }  public void Save()  {  \_stdRepo.AddStudent(this);  }  }  public class StudentRepository  {  public void AddStudent(Student std)  {  //EF code removed for clarity  }  public void DeleteStudent(Student std)  {  //EF code removed for clarity  }  public void EditStudent(Student std)  {  //EF code removed for clarity  }  public IList<Student> GetAllStudents()  {  //EF code removed for clarity  }  }  Example: Classes after applying DIP  public class Student  {  public int StudentId { get; set; }  public string FirstName { get; set; }  public string LastName { get; set; }  public DateTime DoB { get; set; }  private IStudentRepository \_stdRepo;  public Student(IStudentRepository stdRepo)  {  \_stdRepo = stdRepo;  }  public void Save()  {  \_stdRepo.AddStudent(this);  }  }  public interface IStudentRepository  {  void AddStudent(Student std);  void EditStudent(Student std);  void DeleteStudent(Student std);  IList<Student> GetAllStudents();  }  public class StudentRepository : IStudentRepository  {  public void AddStudent(Student std)  {  //code removed for clarity  }  public void DeleteStudent(Student std)  {  //code removed for clarity  }  public void EditStudent(Student std)  {  //code removed for clarity  }  public IList<Student> GetAllStudents()  {  //code removed for clarity  }  } |
|  |  |
|  |  |
|  |  |

## Data Types

Three types of data type

* Value type => stored on stack =>
* Reference type => stored in heap [contain a memory address of variable value, won’t store the variable value directly in memory.]
* Pointer type => & and \*.



## Keywords

* **Keywords** are the predefined set of reserved words that have special meaning for the compiler.
* So, the keywords in c# cannot be used as identifiers such as variable name, class name, etc., in our applications.

**Use Keywords as Variable Names**

* If you want to use Keywords as variable names (identifiers), then include @ as a prefix for your variable names. For example, **@switch** is a valid identifier, but the **switch** is not.

**Different Types of Keywords**

In c#, Keywords are differentiated into two types those are

* Reserve Keywords =>
* Contextual Keywords => Generally, whenever the new keywords are added to the C# language, those are treated as **Contextual** keywords to avoid breaking c# programs that we wrote in older versions.

## Namespaces

* Used to organize multiple [classes](https://www.tutlane.com/tutorial/csharp/csharp-classes-and-objects-with-examples" \o "C# Classes and Objects with Examples" \t "_blank) in our applications, reducing the code redundancy in our .NET applications.
* namespace keyword used to define/create the namespace in c#.

|  |  |
| --- | --- |
| Multiple Namespaces | namespace Tutlane {     class Program     {     } } namespace CustomNameSpace {     class Welcome {     } } |
| Nested Namespaces in C# | namespace namespace1 {      namespace nestednamespace {      // Nested Namespace Code     }} |

## Comments

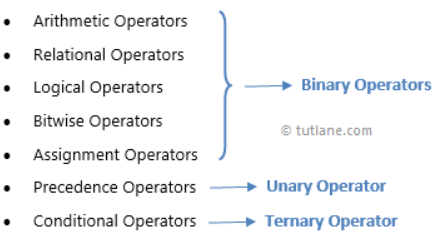
* **Comments** are self-explanatory notes to provide detailed information about the code which we wrote in our applications.

3 types of comment.

|  |  |
| --- | --- |
| Single Line Comments | // Single Line Comment |
| Multiline comments | /\* Multi Line Comment \*/ |
| XML Comments | ///<summary> /// This class does something. ///</summary>  public class SomeClass {  } |

# Operators

|  |  |
| --- | --- |
| **Binary Operator =>** two operand (+, -) | |
| Arithmetic Operator | +, -, /, %, \* |
| Assignment Operator | =,  Arithmetic Operator +=, -=, /=,%=,\*=,  Bitwise Operator &=, !=, ^=,~=, >>=, <<= |
| Relation Operator | ==, >, >=, <, <=, != |
| Logical Operator | &&, ||, ! |
| Bitwise Operator | &, !, ^, ~, >>, << |
| **Unary operator** => one operand (++) | |
| Precedence Operator |  |
| **Ternary operator** => three operand (?:) | |
| Conditional Operator |  |
|  |  |



# Task

## Cancellation Token

* It provides a simple and efficient way to manage cancellation of long-running or asynchronous tasks.
* It allows you to cancel an operation cooperatively, meaning that the operation itself monitors the cancellation request and stops executing when it is requested.
* This helps in avoiding unnecessary work and improving the responsiveness of your application.

Example:

* Consider a scenario where you have an application that downloads large files from the internet. The user might want to cancel the download operation if it takes too long. You can use a CancellationToken to handle this cancellation request.

To request cancellation, call the **Cancel method** on the **CancellationTokenSource**:

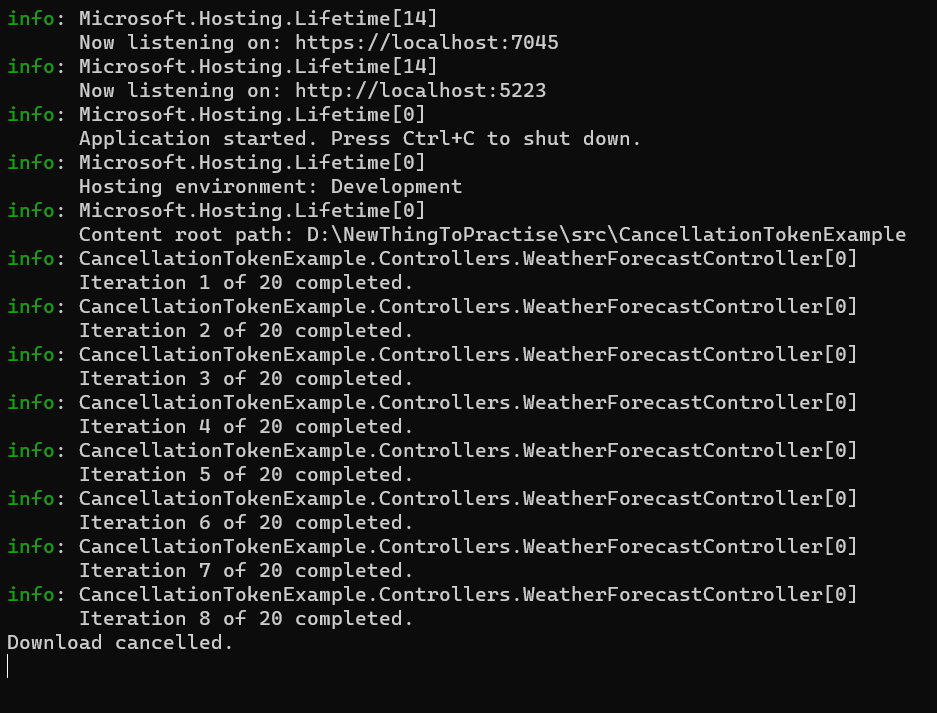
cancellationTokenSource.Cancel();

Let us understand with example, using code:

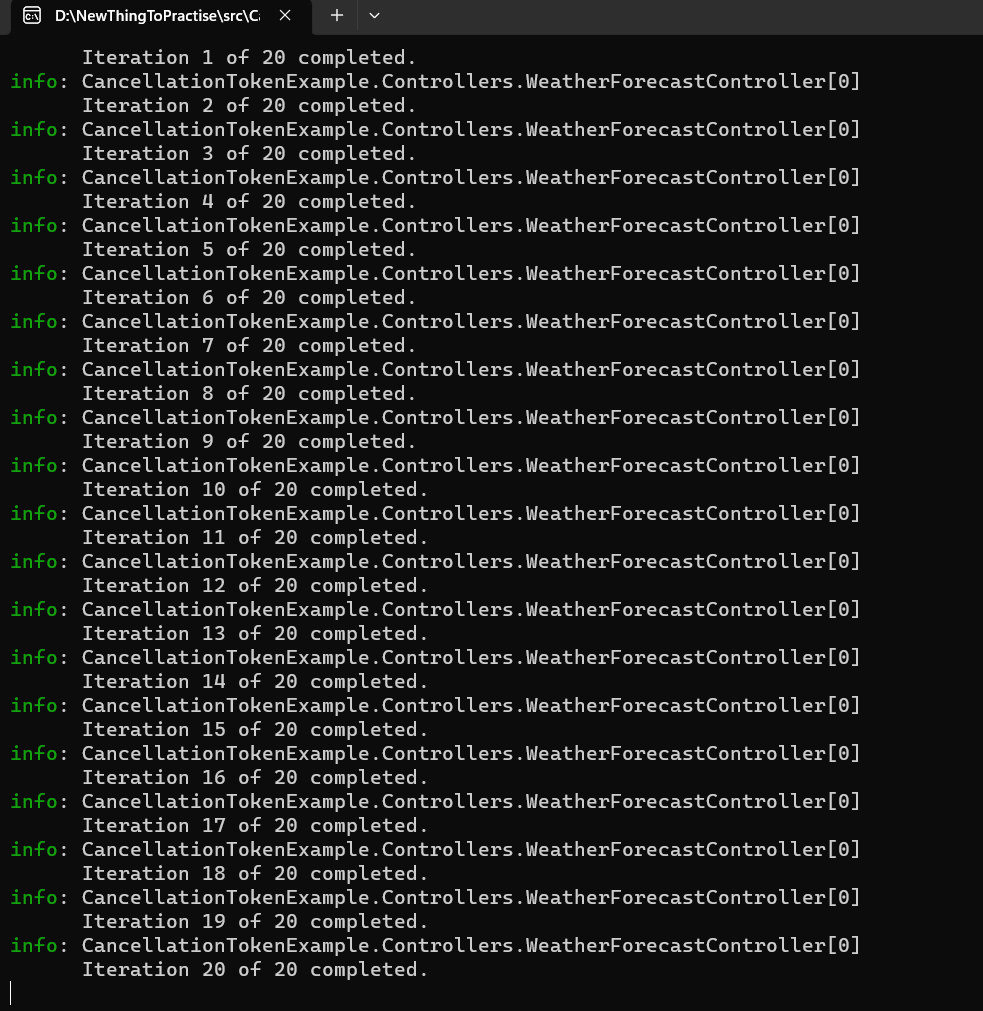
* A below method simulates a long-running task by delaying for 1 second in each iteration of a loop. The **CancellationToken** is passed as a parameter, and we use the `**ThrowIfCancellationRequested**()` method to stop the operation if a cancellation request is received.



* Now, if we cancel the request after a certain period of time, the ongoing operation will be interrupted and will not proceed with further processing. This ensures that the operation stops as soon as the cancellation request is made.



* If the CancellationToken is not canceled, the process will execute to completion without interruption. The operation will continue until it has finished, and it will not stop prematurely.



# Methods

|  |  |
| --- | --- |
| 1. **[Extension Method](https://www.tutorialsteacher.com/csharp/csharp-extension-method)**  => Extension methods, as the name suggests, are additional methods.  => Extension methods allow you to inject additional methods without modifying, deriving or recompiling the original class, struct or interface.  => An extension method is actually a special kind of static method defined in a static class. => To define an extension method, first of all, define a static class. | - Create Extension method  namespace ExtensionMethods  {  public static class IntExtensions  {  public static bool IsGreaterThan(this int i, int value)  {  return i > value;  }  }  }  - Use the method  using ExtensionMethods;  class Program  {  static void Main(string[] args)  {  int i = 10;  bool result = i.IsGreaterThan(100);  Console.WriteLine(result);  }  } |
| **2. [Generic Methods](https://www.tutorialsteacher.com/csharp/csharp-generics)**  => A method declared with the type parameters for its return type or parameters is called a generic method. | class DataStore<T>  {  private T[] \_data = new T[10];    public void AddOrUpdate(int index, T item)  {  if(index >= 0 && index < 10)  \_data[index] = item;  }  public T GetData(int index)  {  if(index >= 0 && index < 10)  return \_data[index];  else  return default(T);  }  } |
|  |  |
|  |  |
|  |  |
|  |  |

# OOP

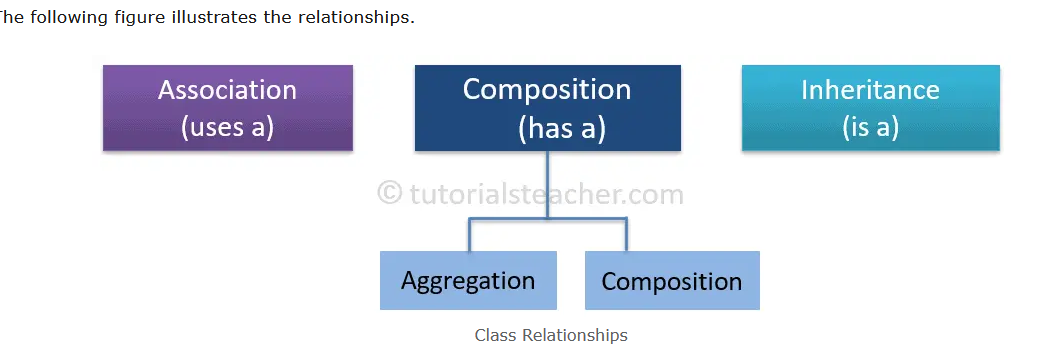
## RelationShip

Reference:

* <https://www.tutorialsteacher.com/csharp/association-and-composition#composition>

There are three types of relationships in object-oriented programming based on how a class interacts with another class.

1. [Association](https://www.tutorialsteacher.com/csharp/association-and-composition#association)
2. Composition
   * [Composition](https://www.tutorialsteacher.com/csharp/association-and-composition#composition)
   * [Aggregation](https://www.tutorialsteacher.com/csharp/association-and-composition#aggregation)
3. [Inheritance](https://www.tutorialsteacher.com/csharp/inheritance)



### Association

* Association happens between the classes where one class provides a service to another class or the class delegates some kinds of behaviors to another class.



Important Points:

* A class only uses behaviors/functionalities (methods) of another class but does not change them by overriding them.
* A class does not inherit another class.
* A class does not include (own) another class as a public member.
* Both classes have independent lifetime where disposing of one does not automatically dispose of another.

### Composition

* In the composition relationships, a class that contains the reference to another class is the parent (owner) of that child class. The child class without parent class doesn't exist.

public class Student

{

public int StudentId { get; set; }

public string FirstName { get; set; }

public string MiddleName { get; set; }

public string LastName { get; set; }

public Address HomeAddress { get; set; }

}

public class Address

{

public int AddressId { get; set; }

public string Address1 { get; set; }

public string Address2 { get; set; }

public string City { get; set; }

public string State { get; set; }

public string ZipCode { get; set; }

public string Country { get; set; }

}



Important Points:

* A class (parent) contains a reference to another class (child).
* The child class doesn't exist without the parent class.
* Deleting the parent class will also delete the child class
* A class can also include a reference of the id property of another class.

### Aggregation

* Aggregation is another category of "has a" relationship where a class can contain other classes as properties but those classes can exist independently.

public class Student

{

public int StudentId { get; set; }

public string FirstName { get; set; }

public string MiddleName { get; set; }

public string LastName { get; set; }

public Course EnrolledCourse { get; set; }

}

public class Course

{

public int CourseId { get; set; }

public string CourseName { get; set; }

public IList Topics { get; set; }

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

}

In the above aggregation relationship, even if the Student object is deleted, the Course object will still exist. The Student class can also contain CourseId property instead of Course instance.

**Composition and Aggregation both are "has a" relationship but in the composition relationship, related classes don't exist independently whereas, in the aggregation, related classes exist independently.**

****

Important Points:

* Aggregation is another type of composition ("has a" relation).
* A class (parent) contains a reference to another class (child) where both classes can exist independently.
* A class can also include a reference of the id property of another class.

### Inheritance

* Inheritance is a mechanism of reusing the functionalities of one class into another related class.
* Inheritance is referred to as "is a" relationship.

#### Role of Access Modifiers in Inheritance

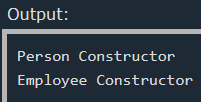
* Access modifiers of each member in the base class impact their accessibility in the derived class.

|  |  |  |
| --- | --- | --- |
| **Access Modifiers** | **Accessible in Derived class** | **Become part of derived class** |
| 1. Public members | ✅ | ✅ |
| 2. Private members | Close with solid fill | Close with solid fill |
| 3. Protected members | ✅ | Close with solid fill |
| 4. Internal members | ✅ | ✅ |

|  |  |
| --- | --- |
| **Public Members**   * The public members of the base class are accessible in the derived class and also become part of the derived class object. | class Person  {  public string FirstName { get; set; } // can be inherited  }  class Employee : Person  {  }  Employee emp = new Employee();  emp.FirstName = "Bill"; // valid |
| **Private Members**   * The private members of the base class cannot be accessed directly from the derived class and cannot be part of the derived class object. | class Person  {  private string FirstName { get; set; } // cannot be inherited  }  class Employee : Person  {  }  Employee emp = new Employee();  emp.FirstName; // Compile-time error |
| **Protected Members**   * The protected members of the base class can be accessible in the derived class but cannot be a part of the derived class object. | class Person  {  protected string FirstName { get; set; }  }  class Employee : Person  {  public int GetName()  {  return this.FirstName;// valid  }  }  Employee emp = new Employee();  emp.GetName();// valid  emp.FirstName; // Compile-time error. |
| **Internal Members**   * Internal members are accessible in the derived class and are part of the derived class object. |  |
|  |  |

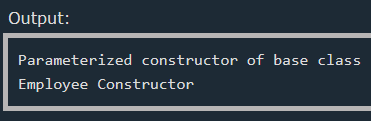
#### Constructors

* Creating an object of the derived class will first call the constructor of the base class and then the derived class.
* If there are multiple levels of inheritance then the constructor of the first base class will be called and then the second base class and so on.

* Use the base keyword in the derived class to access the public members of the base class. For example, the following calls the base class's parameterized constructor using the :base().





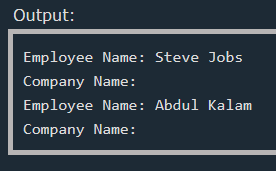
#### Object Initialization



#### Type Conversion

The base type converts to the base class implicitly whereas the derived type must be converted to the base class explicitly using the **as** operator.





#### Types of Inheritance

|  |  |
| --- | --- |
| **Single Inheritance**   * only one derived class inherits a single base class. |  |
| **Multi-level Inheritance**   * A derived class inherits from a base class and then the same derived class becomes a base class for another derived class. * Practically, there are no limits on the level of inheritance, but you should avoid it. |  |
| **Hierarchical Inheritance**   * Multiple derived classes inherit from a single base class. |  |
| **Hybrid Inheritance**   * It is a combination of multi-level and hierarchical inheritance. |  |
| **Multiple Inheritance**   * In multiple inheritance, a class inherits from multiple interfaces. * Note that **C# does not support deriving multiple base classes**. Use interfaces for multiple inheritance. |  |

#### Important Points:

* In C#, three types can participate in inheritance: Class, Struct, and Interface.
* A class can inherit a single class only. It cannot inherit from multiple classes.
* A class cannot inherit from a struct.
* A class can inherit (implement) one or more interfaces.
* A Struct can inherit from one or more interfaces. However, it cannot inherit from another struct or class.
* An interface can inherit from one or more interfaces but cannot inherit from a class or a struct.
* Constructors or destructors cannot be inherited.

# Delegate

# Interview Questions

### **1. What Is C#?**

* C# is a type-safe, object-oriented language used to create .Net applications with a component-oriented approach.
* You can use C# to create apps for Microsoft Windows, cloud-based API services, mobile apps for iOs and Android, software for AI and Machine Learning (ML), blockchain apps, and serverless apps.

### **2. C# Features.**

* **Simple language:** Lacks pointers or direct memory modification vs. C++ or C A simple language that lacks pointers or direct memory modification is one that does not allow users to directly access or manipulate memory addresses.
* **Type-safe:** Ensures variable data types are correctly defined
* **Object-oriented:** Define and use classes/objects
* **Component-oriented:** Reuse existing components without coding from scratch
* **Open-source:** Can develop apps on multiple OS, inc. Mac and Linux
* **Interoperability:** Can develop with managed and unmanaged 3rd party code
* **Structured-programming:** Can break programs into functions
* **Rich library:** Built-in functions speed up development
* **Fast:** Compilation and execution are very quick

Simple Langauge

C# does support pointers, but their usage is limited and generally discouraged in favor of safer alternatives like `ref` and `out`. Pointers in C# are considered an advanced feature and are typically used only in specific situations where performance or interoperability with native code is critical.

To use pointers in C#, you need to use the `unsafe` keyword, which indicates that the code block may contain unsafe code, such as pointer manipulation. You also need to enable the "Allow unsafe code" option in your project settings.

Here's an example of using pointers in C#:



### 3. Managed and Unmanaged code

* Managed code is executed by the .NET runtime and provides several benefits like memory management, security, and portability
* Unmanaged code is executed directly by the operating system and can offer better performance in some cases but comes with increased complexity and potential risks.