static long Factorial(int n)

{

if (n <= 1)

{

return 1;

}

else

{

return n \* Factorial(n - 1);

}

}

**private** **static** **int** NthFibonacciNumber**(int** number**)**

**{**

**if** **((**number == 0**)** || **(**number == 1**))**

**{**

**return** number;

**}**

**else**

**{**

**return** **(**NthFibonacciNumber**(**number - 1**)** + NthFibonacciNumber**(**number - 2**))**;

**}**

**}**

for (int i = 2; i < n; i++)

{

nextNumber = firstNumber + secondNumber;

Console.Write(nextNumber + " ");

firstNumber = secondNumber;

secondNumber = nextNumber;

}

using System;

namespace FirstProgram

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Prints on ");

Console.WriteLine("New line");

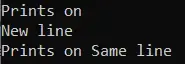
Console.Write("Prints on ");

Console.Write("Same line");

}

}

}



using System;

namespace FirstProgram

{

class Program

{

static void Main(string[] args)

{

int userInput;

Console.WriteLine("Press any key to continue...");

Console.ReadKey();

Console.WriteLine();

Console.Write("Input using Read() - ");

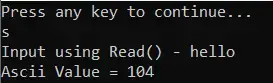
userInput = Console.Read();

Console.WriteLine("Ascii Value = {0}", userInput);

}

}

}



String

* Class vs keyword
* Methods
* Substring, subsequence, subset

Substring: contiguous, order- n(n+1)/2

Subsequence: order – 2n

Subset: no order, no contiguous (presence also no problems) 2n

* Both [3, 1] and [1, 3] are indeed valid subsets of the set {1, 2, 3}. The presentation order doesn't negate their validity as subsets.
* Int x=0; x=1 => same memory, only value changes
* String s=”pip”; s=” hi” => different memory so strings are Immutable (can’t manipulate the data)
* **They made Strings Immutable for Thread Safety, Security, concurrency and synchronization**

| **Aspect** | **Synchronization** | **Asynchronization** |
| --- | --- | --- |
| **Execution** | **Tasks are executed sequentially one after another. Blocks the execution of subsequent tasks until the current task completes.** | **Tasks are executed independently, not necessarily waiting for previous tasks to complete. Allows the initiation of a new task before the previous finishes.** |

**String builder, string buffer, string (string buffer is not present in C#)**

| **Feature/Aspect** | **String** | **StringBuffer** | **StringBuilder** |
| --- | --- | --- | --- |
| **Mutability** | **Immutable (cannot be changed)** | **Mutable (can be changed)** | **Mutable (can be changed)** |
| **Thread Safety** | **Thread-safe,** | **Thread-safe (synchronized)** | **Not thread-safe** |
| **Performance** | **Fast for operations that do not require modifications.** | **Slower due to synchronization overhead when threading is not needed.** | **Faster in scenarios requiring frequent modifications, especially in a single-threaded context.** |

**Memory string pool Heap Heap**

**StringBuilder is part of the System.Text**

**The String Intern in C# is a process that uses the same memory location if the value is the same**

**String builder methods**

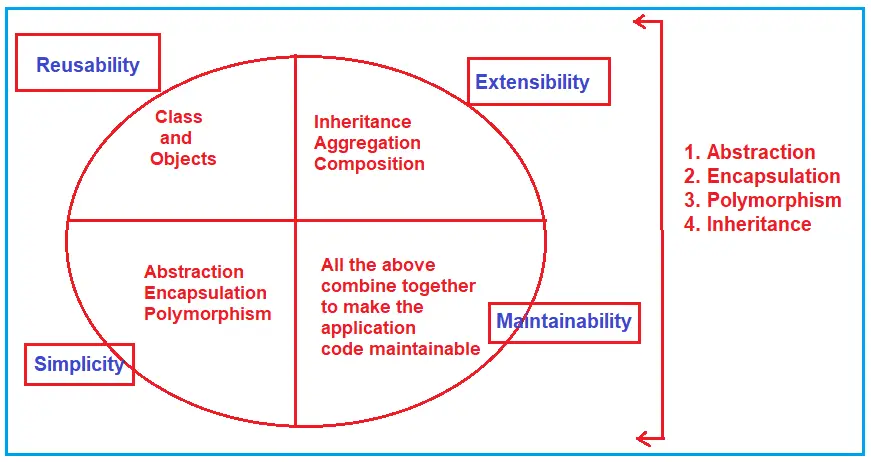
StringBuilder s = new StringBuilder(20);

or

StringBuilder s = new StringBuilder("GeeksForGeeks", 20); capacity

* Append(string value)
* AppendFormat()
* Insert(int index, string value)
* Remove(int start, int length)
* Replace(old\_val, new|\_val)

OOPS:

****

**In C# we have the below types of classes**

1. [**Abstract Class**](https://dotnettutorials.net/lesson/abstract-class-abstract-methods-csharp/)
2. **Concrete class/ normal class**
3. [**Sealed Class**](https://dotnettutorials.net/lesson/sealed-class-methods-csharp/) **(sealedMethod should be always overrided)**
4. [**Partial Class**](https://dotnettutorials.net/lesson/partial-classes-partial-methods-csharp/)
5. [**Static Class**](https://dotnettutorials.net/lesson/static-class-in-csharp/)

The private method is not inherited whereas the sealed method is inherited but cannot be overridden in C#. So, a private method cannot be called from sub-classes whereas a sealed method can be called from sub-classes.

**a constructor is a block of codes similar to the method. It is called when an instance of the**[**class**](https://www.javatpoint.com/object-and-class-in-java)**is created**

**Constructor Rules:**

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. Implicitly Defined Constructors are parameter less and these constructors are also known as Default Constructors. This is because they are used to initialize the variables with default values.
4. Implicitly Defined Constructors are public.
5. We can also define a constructor under the class and if we define it, we can call it an Explicit Constructor and an Explicit Constructor can be parameter less and parameterized also.
6. It can have access specifiers

There are five types of constructors available in C#, they are as follows

1. **Default or Parameter Less Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor**
4. **Static Constructor**
5. **Private Constructor**

Copy Constructor : when we want to create multiple objects.  
A screenshot of a computer program

Description automatically generated

 the memory is separate for each instance but the value is going to be the same for both instances

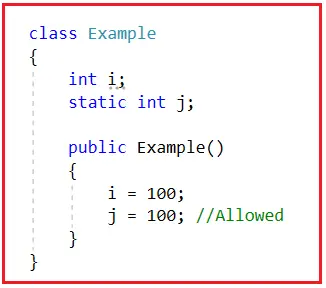
Static Constructor: in C# word

**Can we initialize non-static data members within a static constructor in C#?**

It is not possible to initialize non-static data members within a static constructor, it raises a compilation error.

**Can we initialize static data fields within a non-static constructor in C#?**

Yes, we can initialize static data members within a non-static constructor.



**Points To Remember about C# Private Constructor:**

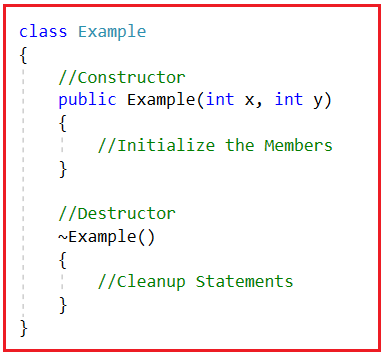
1. Using Private Constructor in C# we can implement the singleton design pattern.
2. We need to use the private constructor in C# when the class contains only static members.
3. Using a private constructor is not possible to create an instance from outside the class.
4. Class cannot be inherited with only one private constructor
5. Overloaded private constructor is possible

Destructor:

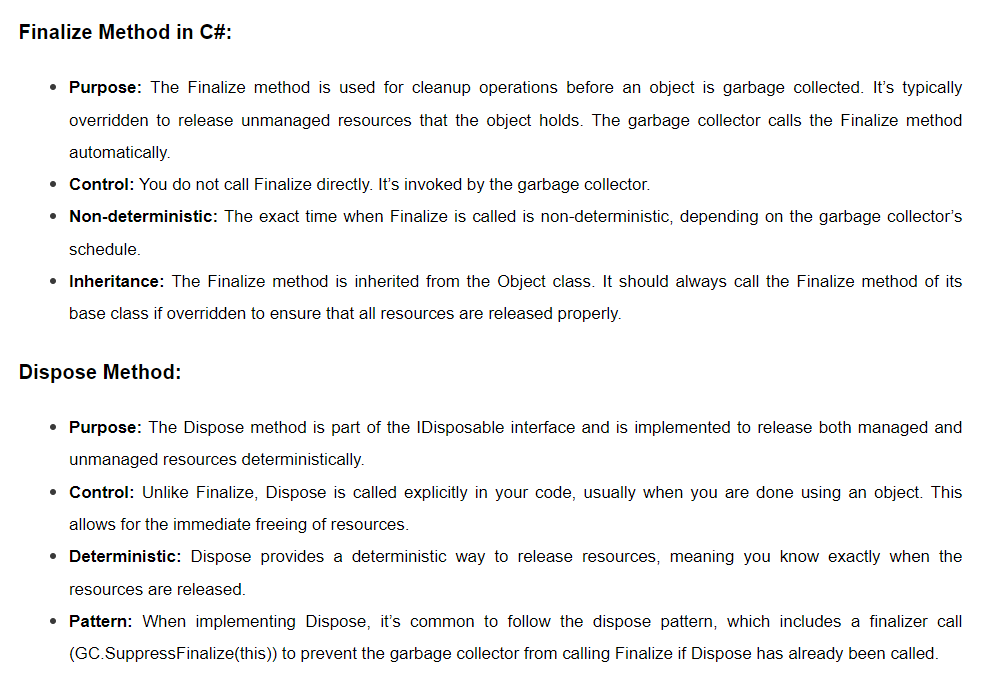
According to [**MSDN**](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/finalizers), Destructors which are also called Finalizers in C# are used to perform any necessary final clean-up when object is being collected by the garbage collector

A destructor method gets called automatically by the garbage collector when the object of the class is destroyed.

Explicitly we can call destructor using GC.Collect() method



1. Destructors (or Finalizers) cannot be defined in structs. In C#, they are only used with only classes
2. we can only define one Destructor (or Finalizer). That means Destructors cannot be overloaded in a class.
3. Destructors cannot be called Explicitly.  It’s invoked by the garbage collector.
4. A Destructor does not take any modifiers or does not have any parameters



**Key Differences Between Finalize and Dispose in C#:**

* **Timing:** Finalize is called by the garbage collector in a non-deterministic manner, while Dispose is called explicitly at a known point in the program.
* **Resources:** Finalize is typically used for unmanaged resources, whereas Dispose can be used for both managed and unmanaged resources.
* **Control:** Dispose gives you more control over resource management compared to Finalize.

An assembly is a compiled code library used for deployment, versioning, and security.

* **Executable (EXE)**: These are standalone applications that can be executed directly by the user.
* **Library (DLL)**: These are not standalone applications but are libraries of code used by other applications.
* **Abstract class:** both abstract and non- abstract methods are used.
* override keyword should be used. Abstract class have constructor
* Yes, we can create a reference for the abstract class in C#. But we cannot create an instance of an abstract class in C#.
* We can define all static and non-static members including properties, fields, indexes, and abstract methods
* Static members can be executed directly by using the class name and its non-static members are executed by using its concrete sub-class or child class object.
* No, we are not allowed to declare an abstract method as static, private, sealed.
* The **virtual** keyword is essential for method overriding in C# when you want to allow derived classes the ability to modify or replace the implementation of a method defined in the base class.

Interface:

* The first point that you need to remember is that the default scope for an interface’s members is public, whereas it is private in the case of a class.
* The second point that you need to remember is by default, every member of an interface is abstract, so we aren’t required to use the abstract modifier on it again,
* You need to remember that we cannot declare fields/variables, constructors, and destructors in an interface in C#. (static, const variables can be declared)

An interface can contain

* + **Abstract methods**
  + **Properties**
  + **Indexes**
  + **Events**
* The fourth point you need to remember is that an interface can inherit from another interface in C# just like a class inherits from another.
* The fifth point that you need to remember is every member of an interface should be implemented under the child class without fail (mandatory), but while implementing, we aren’t required to use the override modifier just like we have done in the case of an abstract class.

**What are the Advantages of using Interface in C#?**

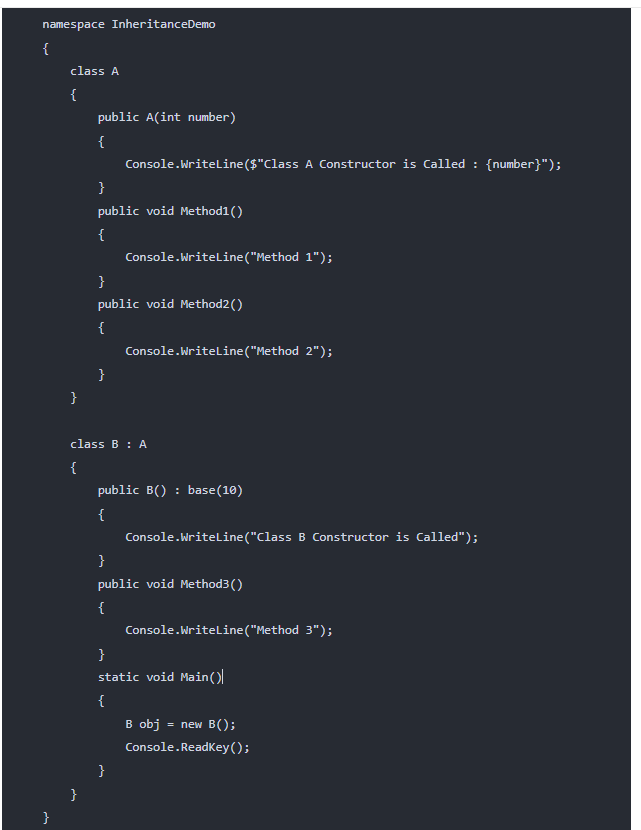
The following are the advantages of using Interface in the C# application.

* It is used to achieve loose coupling.
* It is used to achieve total abstraction.
* To achieve multiple inheritance and abstraction

**Inheritance:**

all parents’ properties belong to their children.

* **Child Cannot Consume Private Members of Parent**
* the parent classes constructor must be accessible to the child class; otherwise, the inheritance would not be possible (the Class A constructor is private, so it is not accessible to Class B.)
* In inheritance, the child class can access the parent class members, but the parent classes can never access any members that are purely defined in the child class.
* We can Initialize a Parent class variable by using the child class instance to make it a reference variable so that the reference will be consuming the memory of the child class instance.
* Every class that is defined by us or predefined in the libraries of the language has a default parent class, i.e., the Object class of the System namespace
* In C#, we don’t have support for multiple inheritances through classes.





**Code reusability**

A screenshot of a computer program

Description automatically generated

**cohesion** refers to the degree to which the elements (i.e., methods, variables) inside a class are related to each other

Coupling : dependency of one class on another class. How closely they are interconnected

Association represents the relationship between the objects.

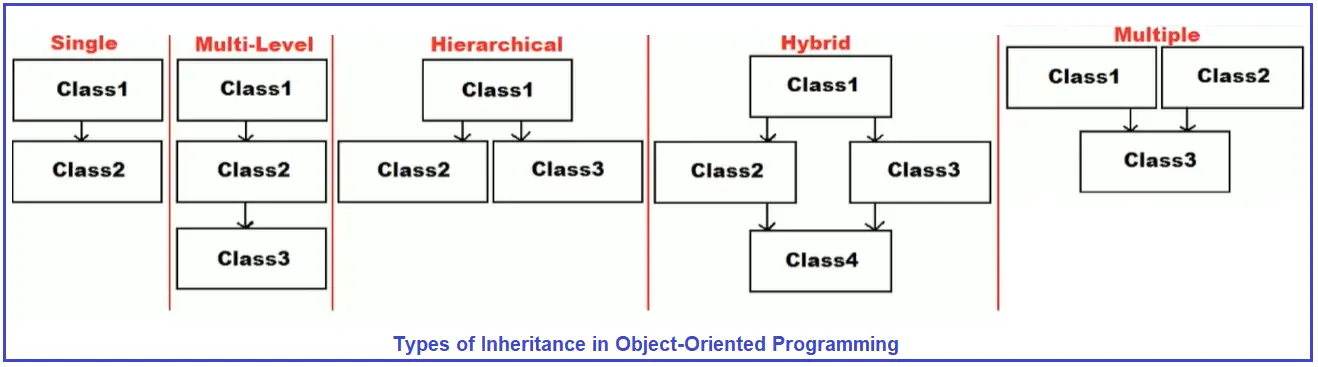
* One to One
* One to Many
* Many to One, and
* Many to Many
* Aggregation: Depicts a 'has-a' relationship where the lifetime of the contained objects does not depend on the lifetime of the container, suggesting a weak ownership.
  + Ex: A library has books. 'Library' does not have exclusive control over 'Books'.
* Composition: Represents a stronger 'has-a' relationship where the contained objects' lifetime is tightly linked to the lifetime of the container, indicating strong ownership.
  + Ex: A human has a heart. 'Heart' cannot exist independently from 'Human'.

A screen shot of a computer

Description automatically generated

**Types of Inheritance:**

* Single Inheritance
* Multi-Level Inheritance
* Hierarchical Inheritance
* Hybrid Inheritance
* Multiple Inheritance

****

**Multiple inheritance is not possible in C# but achieved with interfaces**

But with interfaces, we don’t have this ambiguity problem. Suppose, one class is inherited from two interfaces, and if both the interfaces contain the same method, then also we will not face the ambiguity problem. The reason is, in this case, the interface provides the method to the child class for implementation, but not for consumption. Consumption creates ambiguity problems, not implementation. For a better understanding, please have a look at the below diagram.

C#.NET classified the inheritance into two categories, such as

1. **Implementation inheritance:**Whenever a class is derived from another class then it is known as implementation inheritance.
2. **Interface inheritance:**Whenever a class is derived from an interface then it is known as interface inheritance.

**IsA Relationship (Inheritance):**

**Definition:** The IsA relationship is achieved through inheritance. It represents a hierarchical relationship between a base class (parent) and derived classes (children). In an IsA relationship, the derived class is a specialized version of the base class.  
**Characteristics:**

* **Inheritance**: The derived class inherits properties and methods from the base class.
* **Polymorphism**: The derived class can override or extend the functionalities of the base class.
* **Substitutability**: Objects of the derived class can be treated as base class objects.

**Definition:** The HasA relationship is used to denote usage or composition. It indicates that an object of one class “contains” or “is composed of” objects from another class. This relationship is less tightly coupled than inheritance.  
**Characteristics:**

* **Composition:** A class contains or is composed of objects from another class.
* **Independence:** The contained object (e.g., Engine) can exist independently of the container (e.g., Car).
* **Encapsulation:** The internal workings of the contained object are usually hidden from the outside class.
* **Flexibility and Reusability:** Objects can be easily replaced or changed, providing more flexibility in code design.

Composition:

using System;

namespace IsAHasADemo

{

public class Program

{

static void Main()

{

Address address = new Address("B1-3029", "BBSR", "Odisha");

Employee employee = new Employee(1001, "Ramesh", address);

employee.Display();

Console.ReadKey();

}

}

class Address

{

public string AddressLine, City, State;

public Address(string addressLine, string city, string state)

{

AddressLine = addressLine;

City = city;

State = state;

}

}

class Employee

{

//Using Address in Employee class

//Establishing Has-A relationship i.e. Employee HAS-A Address

public Address address;

public int Id;

public string Name;

public Employee(int id, string name, Address adrs)

{

Id = id;

Name = name;

address = adrs;

}

public void Display()

{

Console.WriteLine($"Employee Id: {Id}");

Console.WriteLine($"Employee Name: {Name}");

Console.WriteLine($"AddressLine: {address.AddressLine}");

Console.WriteLine($"City: {address.City}");

Console.WriteLine($"State: {address.State}");

}

}

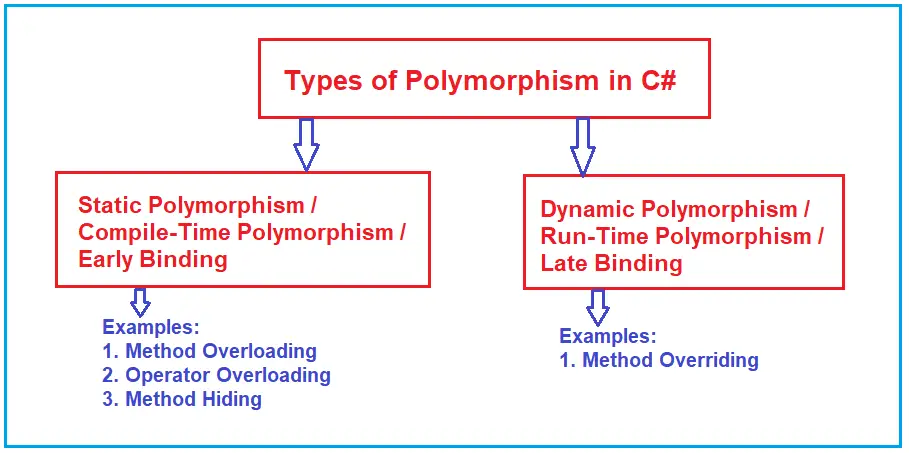
}

So, the Rectangle class is already existing and from the Rectangle class, we have defined a new class with some extra features i.e. we have a specialized class that is Cuboid. This is a specialization in C#.

Let’s say we have 3 classes that are Square, Rectangle, and Circle. Then for all of these three classes, we are defining one class that is Shape. Generalization

* When the **sealed** keyword is applied to a class, it prevents that class from being used as a base class. This means you cannot inherit from the sealed class, but you can still create instances of it.
* Static – No object creation, no override/inherited

|  |  |
| --- | --- |
| Inheritance is a mechanism by which a new class, known as a derived class, can inherit properties, methods, and other characteristics from another class known as a base class. | Method overriding is a feature that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. |

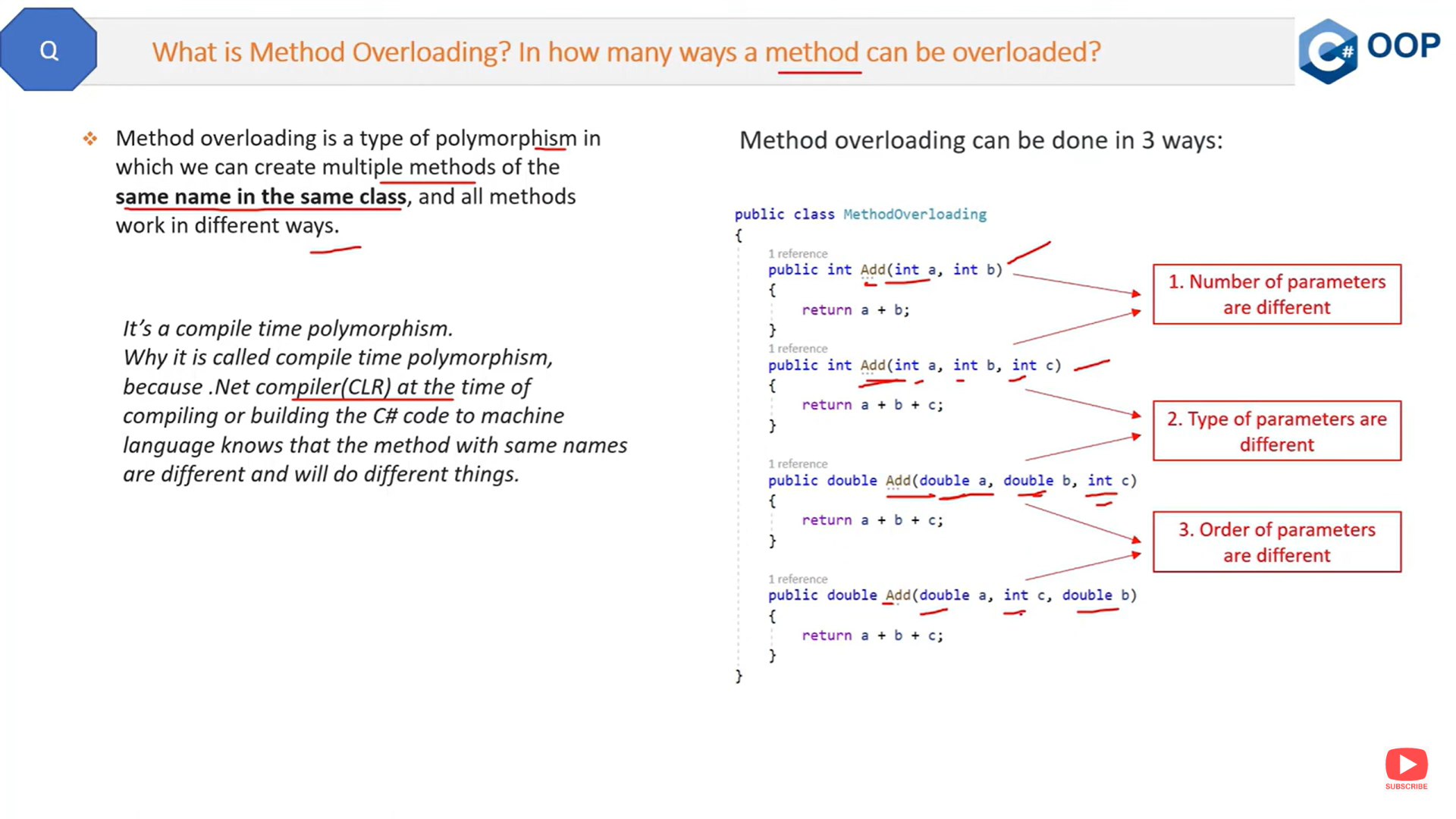


In C#, you cannot "override" variables in the same way you can override methods. Base.a or hide variables using new

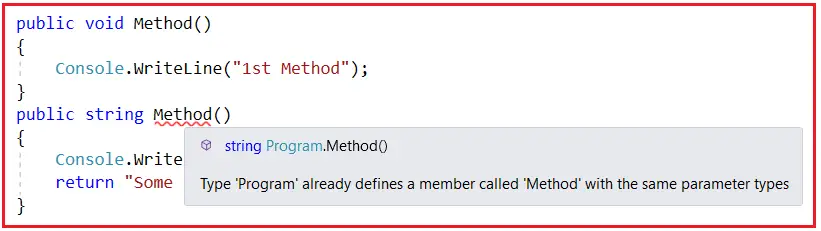
public new int number = 24;

In static polymorphism, the behavior of a method is decided at compile time

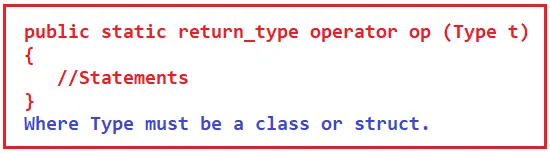
In dynamic polymorphism, the behavior of a method is decided at runtime.



**Why Return Type is not considered as part of Method Overloading in C#?**

****

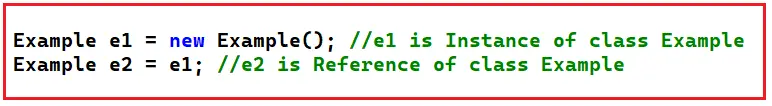
* **Can be different or the same among overloaded methods, but it alone cannot be used to distinguish overloaded methods.**
* Covariant return types allow a method in a derived class to have a return type that is a subtype of the return type declared in the method of the base class. This feature is more common in languages like Java but is limited in C#.
* As of C# 9.0 (introduced with .NET 5), you can use covariant return types with **override methods**.
* Methods can be overloaded in the same or in super and sub classes because overloaded methods are different methods.



1. The return type is the return type of the function.
2. the operator is a keyword.
3. Op is the symbol of the operator that we want to overload. Like: +, <, -, ++, etc.
4. The type must be a class or struct. It can also have more parameters.
5. It should be a static function.

 C#. So, a private method cannot be called from sub-classes whereas a sealed method can be called from sub-classes

1. **Variable of a Class:** A copy of the class that is not initialized.
2. **Instance of a Class:** A copy of the class that is initialized by using the new keyword which has its own memory and is never shared with another instance.
3. **Reference of a Class:**A copy of a class that is initialized by using an existing instance and references of a class will not have any memory allocation, they will be sharing the memory of the same instance that is assigned for initializing the variable.



**Stack Memory Key Points:**

* **Allocation:** Stack memory is allocated for static memory allocation and local variables. It’s managed by the CPU, making it faster and more efficient.
* **Usage:** When a method is called, a block of memory (a stack frame) is allocated on the stack for its local variables and parameters. When the method call returns, the block becomes unused and can be used for the next method call.
* **Lifespan:** Variables stored in the stack are only available during the lifetime of the method call.
* **Type of Data:** It stores value types in C#. These include primitive data types (like int, double, char), structs, and references to objects (the references themselves, not the objects).

**Heap Memory Key Points:**

* **Allocation:** Heap memory is used for dynamic memory allocation, which includes objects and complex data structures that require more flexibility and are managed by the garbage collector in .NET.
* **Usage:** Objects are allocated on the heap, and memory is managed at runtime. New objects are created using the new keyword, and the garbage collector automatically frees up heap memory when objects are no longer in use.
* **Lifespan:** Objects on the heap live from when they are created until they are no longer used and are garbage collected.
* **Type of Data:** It stores reference types like objects, arrays, and class instances.

**Key Differences Between Stack and Head Memory in .NET:**

* **Management:** Stack memory is automatically managed by the system, whereas heap memory is dynamically allocated and deallocated by the garbage collector.
* **Speed:** Stack memory is generally faster than heap memory because of its organization and the way it’s managed.
* **Size:** The stack has size limits based on the thread, but the heap can dynamically grow as needed (limited by the system’s available memory).
* **Access:** Stack memory access is more straightforward and faster, while heap memory requires more complex management.
* **Storage:** Value types are stored in stack memory, while reference types are stored in heap memory.

Stack Memory: values types, method calling, structures, local variables in methods

According to MSDN, The **Checked Keyword in C#** is used to explicitly enable overflow checking for integral-type arithmetic operations and conversions.

The **Unchecked Keyword in C#** is used to suppress overflow-checking for integral-type arithmetic operations and conversions.

**Array:**

1. Fixed Length
2. Cannot insert it into the middle
3. Cannot delete from middle
4. It is type-safe, so we can store only similar types of data based on the data type.
5. Boxing and Unboxing are not required.

**ArrayList:**

1. Variable Length
2. Can insert an element into the middle of the collection
3. Can delete elements from the middle of the collection
4. It is not type-safe, so we can store any type of data.
5. Boxing and Unboxing are required as it is operated on the object data type.

I Enumerable, Enumerator, IEquatable

**Rules of using Delegates in C#:**

1. A delegate in C# is a user-defined type and hence before invoking a method using a delegate, we must have to define that delegate first.
2. The signature of the delegate must match the signature of the method, the delegate points to otherwise we will get a compiler error. This is the reason why delegates are called type-safe function pointers.

**What are the Types of Delegates in C#?**

The Delegates in C# are classified into two types as

1. **fSingle Cast Delegate**
2. **Multicast Delegate**

**Where do we use Delegates in C#?**

Delegates are used in the following cases:

1. Event Handlers
2. Callbacks
3. Passing Methods as Method Parameters
4. LINQ
5. Multithreading

Delegates in C# are type-safe [function pointer in C/C++](https://www.geeksforgeeks.org/function-pointer-in-c/). It provides a way which tells which method is to be called when an event is triggered.

Anonymous class:

var myObject = new { Name = "John Doe", Age = 30 };

Anonymous method:

public delegate void MyDelegate(int x);

public class Program {

public static void Main() {

MyDelegate del = delegate (int x) {

Console.WriteLine("Value is " + x);

};

del(10); // Output: Value is 10

}

}

Lambdas:

MyDelegate del = x => Console.WriteLine("Value is " + x);

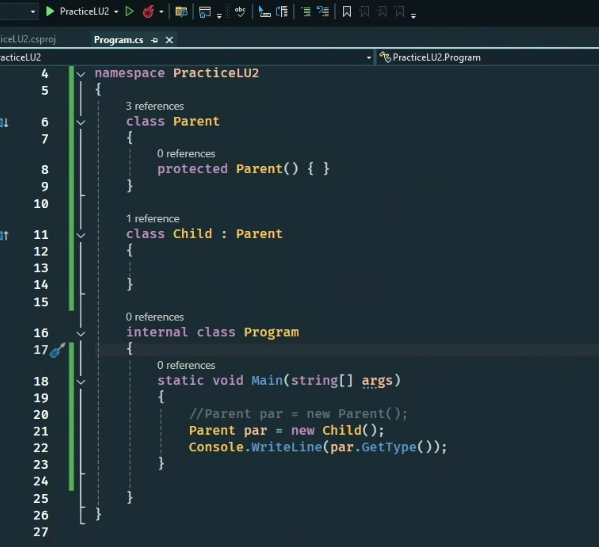
del(10); // Output: Value is 10

Fastest:

* If you frequently access elements by index, List<T> is suitable, especially if additions and deletions are primarily at the end of the list.
* If you need rapid insertions and deletions not necessarily at one end, LinkedList<T> might be appropriate if you manage node references effectively.
* If you perform many containment checks, or need quick add/remove operations without an implicit order, HashSet<T> is ideal.
* For key-value pairs with rapid lookups, adds, and deletes, Dictionary<TKey, TValue> is excellent.

According to MSDN, the volatile keyword indicates that a field might be modified by multiple threads that are executing at the same time.

AutoMapper is a popular open-source library in C# that simplifies mapping data between different classes or objects. It helps eliminate repetitive and error-prone code when copying data from one object to another. AutoMapper is especially useful in scenarios like mapping database entities to DTOs (Data Transfer Objects) or ViewModel objects.



inherited method, property, indexer, or event in a derived class.

Anonymous class: by default public readonly

new { Name =”Vishnu”}

Errors vs Exception:

* Errors can’t be handled, unrecover( Syntax errors, compilation errors, environmental and hardware malfunctions.)

|  |  |
| --- | --- |
| Inheritance allows a class to inherit attributes and methods from another class called a base class. | Extension methods allow new methods to be added to existing types without modifying the type itself. |
| Inheritance tightly coupled with concrete classes, loosely coupled with interfaces and abstract classes | Extension methods make high cohesion, loosely coupled. |
| In Inheritance method is overrided to change the functionality | Extension methods extend functionality and add new features like in LINQ |

We can Convert predefined types to identifier name using @

Int @class =12;

Class : blueprint or template of Object uses: reusable, can create many Objects

Object:

Inheritance: is introduced because class should follow set of rules, i.e. base class define rules or contracts, so child should definitely follow those rules or have methods

Abstract classes: is introduced because when we know implementation of only some Rules for now but don’t know about others we can declare them as abstract methods

Interface: is introduced to achieve multiple inheritance, dynamic polymorphism, only abstract methods

Abstraction can be achieved through abstract classes, Interfaces, delegates.