#### Questions for this assignment

Explain inheritance in C# and how it works.

How do you invoke a parent class constructor from a derived class in C#?

What are rules of method overriding and in which scenarios to use it?

What are rules of method hiding and in which scenarios to use it?

What is the difference between method overriding and method hiding and when to use which one. Explain with real world scenario.

What are sealed classes and sealed methods in C#?

Can you explain the use of the 'base' keyword in C# with respect to inheritance?

Difference between ‘base’ keyword and ‘this’ keyword in C#?

What is constructor chaining in C#? Explain with an example.

Can we override a constructor in C#?

What is the difference between method overloading and method overriding in C#?

Explain inheritance in C# and how it works.

Inheritance is a concept in C# that allows a class to inherit properties and behaviors from another class, known as the base or parent class. The class that inherits from the base class is called the derived or child class. Inheritance promotes code reuse, encapsulation, and polymorphism. In C#, inheritance is achieved using the colon (:) symbol, followed by the name of the base class after the class declaration.

Eg:

class ParentClass

{

// Parent class implementation

}

class ChildClass : ParentClass

{

// Child class implementation

}

The child class 'ChildClass' will inherit all the members (fields, properties, and methods) of the parent class 'ParentClass'. The child class can also override or extend the members of the parent class as needed.

How do you invoke a parent class constructor from a derived class in C#?

In C#, you can invoke a parent class constructor from a derived class using the **base** keyword. The **base** keyword is used to access members of the base class, including constructors.

To invoke a parent class constructor, you can use the following syntax in the derived class constructor:

public class DerivedClass : ParentClass

{

public DerivedClass(parameters) : base(baseParameters)

{

// Derived class constructor code

}

}

Here's a breakdown of the syntax:

1. Define the derived class using the **class** keyword.
2. Specify the name of the derived class, followed by a colon (**:**) and the name of the parent class.
3. Declare the constructor of the derived class, providing any necessary parameters.
4. Use the **base** keyword followed by parentheses to call the constructor of the parent class, passing any required parameters.

By using **base(baseParameters)**, you can pass the required parameters from the derived class constructor to the parent class constructor. This allows you to initialize the parent class's state before executing the code in the derived class constructor.

What are rules of method overriding and in which scenarios to use it?

Method overriding in C# is a feature that allows a derived class to provide a new implementation for a virtual or abstract method that is already defined in its base class. The derived class provides its own implementation for the method, which overrides the implementation in the base class.

The key points about method overriding in C# are:

1. Method overriding is only applicable to virtual or abstract methods in the base class. Virtual methods are marked with the 'virtual' keyword, and abstract methods are marked with the 'abstract' keyword.
2. The signature (name, return type, and parameters) of the overriding method in the derived class must match the signature of the virtual or abstract method in the base class.
3. The access level of the overriding method cannot be more restrictive than the access level of the virtual or abstract method in the base class. It can be less restrictive or the same.
4. The 'override' keyword is used in the derived class to indicate that a method is intended to override a virtual or abstract method in the base class.
5. Method overriding is used in scenarios where a derived class wants to provide its own implementation for a method that is already defined in its base class. This allows for polymorphism, where an object of the derived class can be treated as an object of the base class, but the appropriate implementation of the method in the derived class will be called at runtime.

Some scenarios where method overriding is commonly used include:

* **Inheritance of behavior:** When a derived class inherits from a base class and wants to inherit the behavior (methods) of the base class but provide its own implementation for certain methods.
* **Customization:** When a derived class wants to customize the behavior of a method inherited from the base class to suit its specific requirements.
* **Extension:** When a derived class wants to extend the functionality of a method inherited from the base class by adding additional logic.
* **Polymorphism:** When you want to treat objects of different derived classes as objects of a common base class, and call the appropriate implementation of the method based on the actual type of the object at runtime.

Method overriding in C# is a powerful feature that enables customization and extension of behavior in derived classes, and facilitates polymorphism in object-oriented programming.

What are rules of method hiding and in which scenarios to use it?

Method hiding in C# is a feature that allows a derived class to provide a new implementation for a non-virtual, non-abstract method that is already defined in its base class. The new implementation in the derived class "hides" the original implementation in the base class, and the hidden method in the base class is not accessible through objects of the derived class.

The key points about method hiding in C# are:

1. Method hiding is only applicable to non-virtual, non-abstract methods in the base class. Virtual methods can be overridden using method overriding, while non-virtual methods can be hidden using method hiding.
2. The signature (name, return type, and parameters) of the hiding method in the derived class must match the signature of the hidden method in the base class.
3. The 'new' keyword is used in the derived class to indicate that a method is intended to hide a method in the base class.
4. Unlike method overriding, method hiding does not participate in polymorphism. When a method is hidden in a derived class, the version of the method in the base class is not accessible through objects of the derived class. Instead, the version of the method in the derived class is called, regardless of the actual type of the object.

Method hiding is used in scenarios where a derived class wants to provide a completely new implementation for a method that is already defined in its base class, without any connection to the original implementation. This is typically done when the behavior of the method in the derived class is conceptually different from the behavior of the same-named method in the base class.

Some scenarios where method hiding is commonly used include:

* **Code maintenance:** When a base class is part of a third-party library or framework, and you want to provide a different implementation for a method in the derived class without modifying the original code of the base class.
* **Versioning:** When you need to introduce a new version of a class with a different implementation for a method that has the same name as a method in the base class, without affecting the existing code that uses the base class.
* **Customization:** When a derived class wants to provide its own implementation for a method that has the same name as a method in the base class, to customize the behavior of the method in the derived class without affecting the base class.

Method hiding in C# should be used with caution, as it can lead to confusion and unexpected behavior. It is typically used in specific scenarios where you want to provide a completely new implementation for a method in a derived class, without any connection to the original implementation in the base class.

What is the difference between method overriding and method hiding and when to use which one. Explain with real world scenario.

Method overriding and method hiding are two concepts in C# that involve the use of methods with the same name in a class hierarchy. However, they are used in different scenarios and have some key differences.

**Method Overriding:**

Method overriding allows a derived class to provide a new implementation for a method that is already defined in the base class. The derived class provides its own implementation for the base class method with the same name, return type, and parameter list. The base class method is marked as virtual or abstract, and the derived class method is marked with the 'override' keyword.

Example of method overriding:

class Animal

{

public virtual void MakeSound()

{

Console.WriteLine("Animal makes sound");

}

}

class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("Dog barks");

}

}

// Usage:

Animal animal = new Animal();

animal.MakeSound(); // Outputs "Animal makes sound"

Dog dog = new Dog();

dog.MakeSound(); // Outputs "Dog barks"

Real-world scenario: Consider a scenario where you have a base class 'Animal' with a virtual method 'MakeSound', and you want to create a derived class 'Dog' that inherits from the base class. The 'Dog' class can provide its own implementation of the 'MakeSound' method, which represents the specific behavior of a dog making a sound. This allows for polymorphic behavior, where you can call the 'MakeSound' method on objects of both the base class and the derived class, and the appropriate implementation will be invoked based on the object's actual type.

**Method Hiding:**

Method hiding allows a derived class to define a new method with the same name as a method in the base class, but without changing the implementation of the base class method. The new method in the derived class hides the base class method, and the base class method is still accessible through the base class object. Method hiding is achieved by using the 'new' keyword in the derived class method.

Example of method hiding:

class Animal

{

public void MakeSound()

{

Console.WriteLine("Animal makes sound");

}

}

class Dog : Animal

{

public new void MakeSound()

{

Console.WriteLine("Dog barks");

}

}

// Usage:

Animal animal = new Animal();

animal.MakeSound(); // Outputs "Animal makes sound"

Dog dog = new Dog();

dog.MakeSound(); // Outputs "Dog barks"

Animal animal2 = new Dog();

animal2.MakeSound(); // Outputs "Animal makes sound"

Real-world scenario: Consider a scenario where you have a base class 'Animal' with a method 'MakeSound', and you want to create a derived class 'Dog' that inherits from the base class. The 'Dog' class can provide its own implementation of the 'MakeSound' method, which represents additional behavior specific to a dog, but the base class method is still accessible through the base class object. This allows for extending the behavior of the base class method without modifying the existing implementation.

What are sealed classes and sealed methods in C#?

**Sealed class:**

In C#, a sealed class is a class that cannot be inherited by other classes. Once a class is declared as sealed, it cannot be used as a base class for any other class. Sealed classes are used to prevent further modification or extension of a class, and they are marked with the 'sealed' keyword.

Example of a sealed class:

sealed class MySealedClass

{

// Class members and implementation

}

In this example, the 'SealedClass' is marked as sealed, and it cannot be inherited by any other class.

**Sealed method:**

A sealed method, on the other hand, is a method in a class that cannot be overridden by methods in derived classes. Sealed methods are used to prevent further modification or extension of a method's implementation in derived classes, and they are marked with the 'sealed' keyword.

Example of a sealed method:

using System;

class ParentClass

{

public virtual void Foo()

{

Console.WriteLine("ParentClass Foo");

}

public virtual void Bar()

{

Console.WriteLine("ParentClass Bar");

}

}

class ChildClass : ParentClass

{

public sealed override void Bar()

{

Console.WriteLine("ChildClass Bar");

}

}

class GrandChildClass : ChildClass

{

// Error: Cannot override sealed method 'Bar' in ChildClass

//public override void Bar()

//{

// Console.WriteLine("GrandChildClass Bar");

//}

}

class Program

{

static void Main(string[] args)

{

ChildClass childObj = new ChildClass();

childObj.Foo(); // Calls Foo() method in ParentClass

childObj.Bar(); // Calls Bar() method in ChildClass

GrandChildClass grandChildObj = new GrandChildClass();

grandChildObj.Foo(); // Calls Foo() method in ParentClass

grandChildObj.Bar(); // Calls Bar() method in ChildClass

}

}

In this example, the 'Bar' method in the 'ChildClass' is marked as sealed, which means that no further derived class can override this method.

Note that a method can only be marked as sealed if it is declared as 'virtual' in the derived class.

Can you explain the use of the 'base' keyword in C# with respect to inheritance?

In C#, the 'base' keyword is used to refer to the base class from within a derived class. It is used to call the members (fields, properties, and methods) of the base class, or to explicitly invoke the constructor of the base class from the constructor of the derived class.

The 'base' keyword is often used in the following scenarios:

To call a base class constructor from a derived class constructor, using the 'base' keyword followed by parentheses, and passing the appropriate arguments.

Example:

class ParentClass

{

public ParentClass(int x, int y)

{

// Constructor implementation

}

}

class ChildClass : ParentClass

{

public ChildClass(int x, int y, int z) : base(x, y)

{

// Constructor implementation in ChildClass that calls base class constructor

}

}

To access a member of the base class that is hidden by a member with the same name in the derived class, using the 'base' keyword followed by the member name.

Example:

class ParentClass

{

public void Foo()

{

Console.WriteLine("ParentClass Foo");

}

}

class ChildClass : ParentClass

{

public new void Foo()

{

Console.WriteLine("ChildClass Foo");

}

public void Bar()

{

base.Foo(); // Calls the Foo method in the ParentClass

}

}

Difference between ‘base’ keyword and ‘this’ keyword in C#?

In C#, the 'base' keyword is used to refer to the base class from within a derived class. It is used to access the members (methods, properties, fields) of the base class, or to call the base class constructor.

**base**

The 'base' keyword is used to explicitly specify which member or constructor of the base class should be accessed when there is a member or constructor with the same name in the derived class. This is useful in cases where the derived class wants to extend or override the behavior of the base class, but still needs to access the original implementation of the base class.

**this**

The 'this' keyword, on the other hand, is used to refer to the current instance of the class. It is used to access the members of the current instance, or to call the constructors of the current class. While 'base' refers to the base class, 'this' refers to the current class.

What is constructor chaining in C#? Explain with an example.

Constructor chaining in C# is the process of calling one constructor from another constructor from a derived class to a base class. It allows for reusing code and initializing objects in a more efficient and organized manner.

In C#, a class can have multiple constructors, each with different parameters or no parameters. Constructor chaining enables a class to call a different constructor in the base class using the 'base' keyword.

A constructor of a derived class calls a constructor of the base class using the 'base' keyword. This allows for the initialization of the base class members before the derived class members, ensuring that the base class is properly initialized before the derived class.

Example of constructor chaining:

class Animal

{

protected string species;

public Animal(string species)

{

this.species = species;

}

}

class Dog : Animal

{

public Dog(string species, string name) : base(species)

{

this.name = name;

}

// Additional logic for Dog class

}

Constructor chaining in C# is a powerful feature that allows for efficient and organized object initialization, enabling the reuse of code and ensuring proper initialization of base class members in derived classes.

Can we override a constructor in C#?

Unlike other methods in C#, constructors cannot be overridden, as they are not inherited by derived classes and do not participate in polymorphism.

When a class is derived from a base class, it can provide its own constructors to initialize its own state, but it cannot override or inherit constructors from the base class. This means that you cannot use the override keyword on a constructor in C#.

What is the difference between method overloading and method overriding in C#?

In C#, method overloading and method overriding are two different concepts that are used to achieve polymorphism in object-oriented programming.

Here's a brief overview of the differences between method overloading and method overriding in C#:

**Method Overloading:**

* Method overloading allows a class to define multiple methods with the same name but with different parameters.
* Method overloading is resolved at compile-time based on the number, types, and order of arguments passed to the method.
* Method overloading is achieved within the same class or within a class hierarchy.
* Method overloading does not require any relationship between the methods in terms of inheritance or polymorphism.
* Method overloading does not override the base class method, but rather provides additional overloaded methods with different parameter signatures.
* Method overloading is denoted by the same method name with different parameter lists.

**Method Overriding:**

* Method overriding allows a derived class to provide a new implementation for a method that is already defined in its base class.
* Method overriding is resolved at runtime based on the actual type of the object being referred to (runtime polymorphism).
* Method overriding is only possible in classes that have an inheritance relationship, where a derived class inherits a method from its base class and provides a new implementation for that method in the derived class.
* Method overriding requires the use of the override keyword in the derived class to indicate that a method is intended to override a base class method.
* Method overriding allows a derived class to provide a specialized implementation of a method in the derived class, which is invoked when the method is called on an object of the derived class, even if the object is referred to by a base class reference.

In summary, method overloading allows a class to define multiple methods with the same name but different parameters, while method overriding allows a derived class to provide a new implementation for a method that is already defined in its base class. Method overloading is resolved at compile-time, while method overriding is resolved at runtime based on the actual type of the object being referred to.