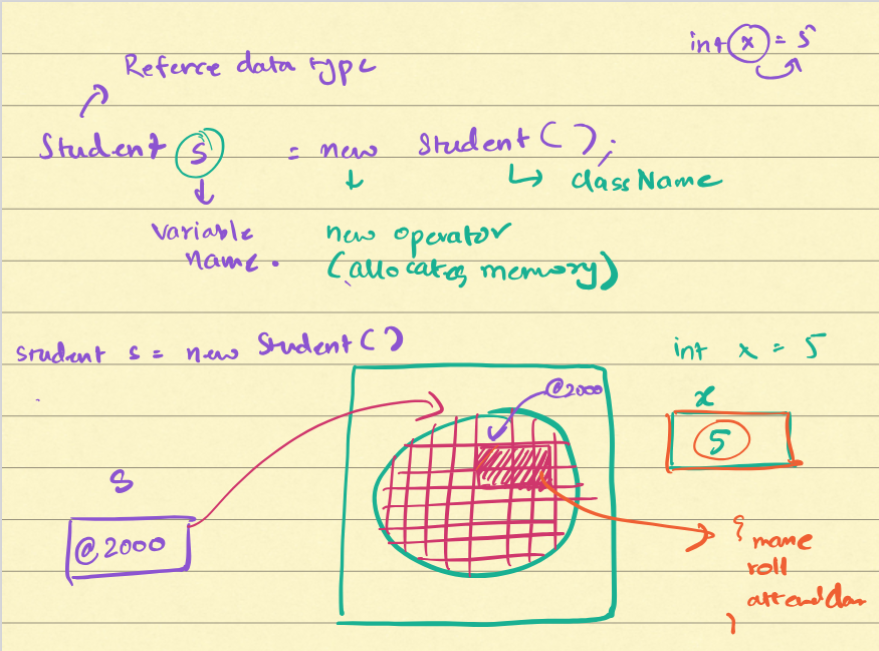
**What is Object-Oriented Programming (OOP)?**

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to structure and organize code. OOP is based on several core principles, including encapsulation, abstraction, inheritance, and polymorphism. The main goal of OOP is to increase the modularity and reusability of code.

**What is the difference between a class and an object?**

* **Class**: A class is a blueprint or template for creating objects. It defines the properties (attributes) and methods (functions) that the objects created from the class will have.
* **Object**: An object is an instance of a class. It is a concrete entity based on the class, with actual values assigned to the properties defined by the class.
* 
* 

**What is encapsulation, and how is it implemented in OOP?**

**Encapsulation** is the concept of bundling data (attributes) and methods (functions) that operate on the data into a single unit, typically a class. Encapsulation restricts direct access to some of an object's components, which is a means of preventing unintended interference and misuse of the data.

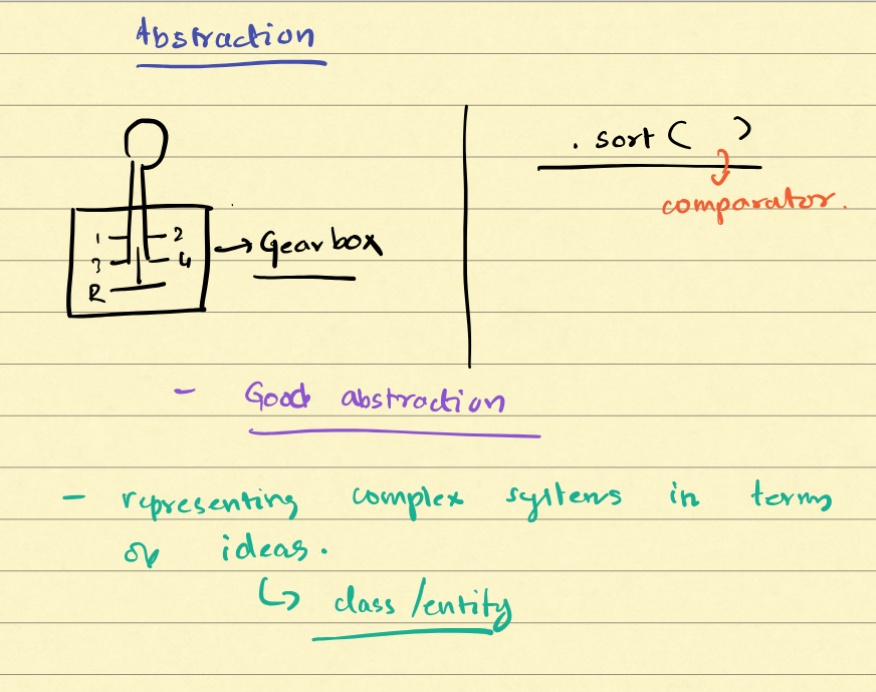
Encapsulation is implemented using access modifiers like private, protected, and public to control the visibility of class members.

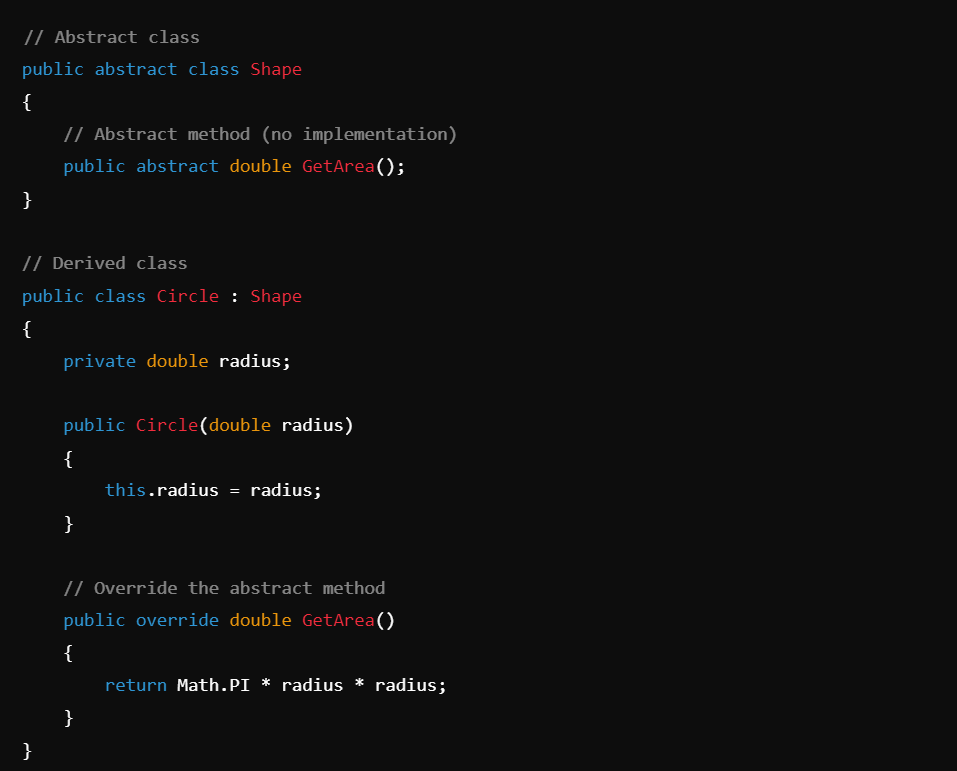


**What is abstraction in OOP, and why is it important?**

**Abstraction** is the concept of hiding the complex implementation details of a system and exposing only the necessary and relevant parts. It helps in reducing programming complexity and effort, making the code more maintainable and understandable.

Abstraction is typically achieved using abstract classes and interfaces in C#.

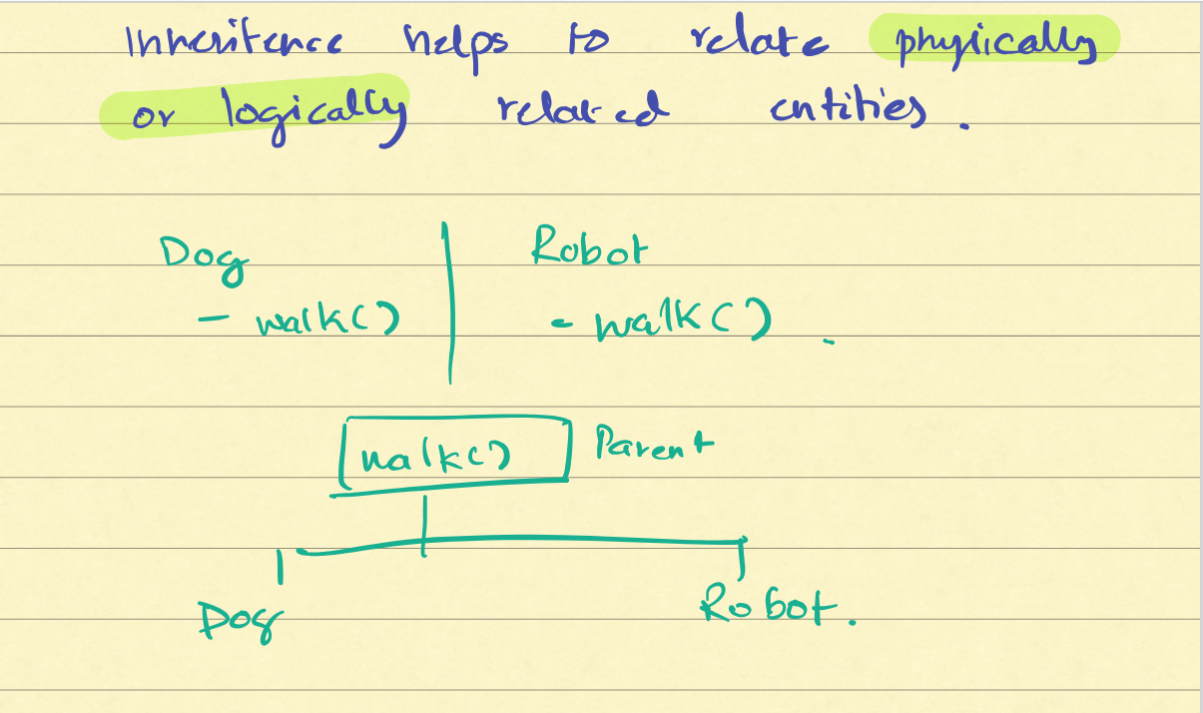






**What is inheritance in OOP, and what are its benefits?**

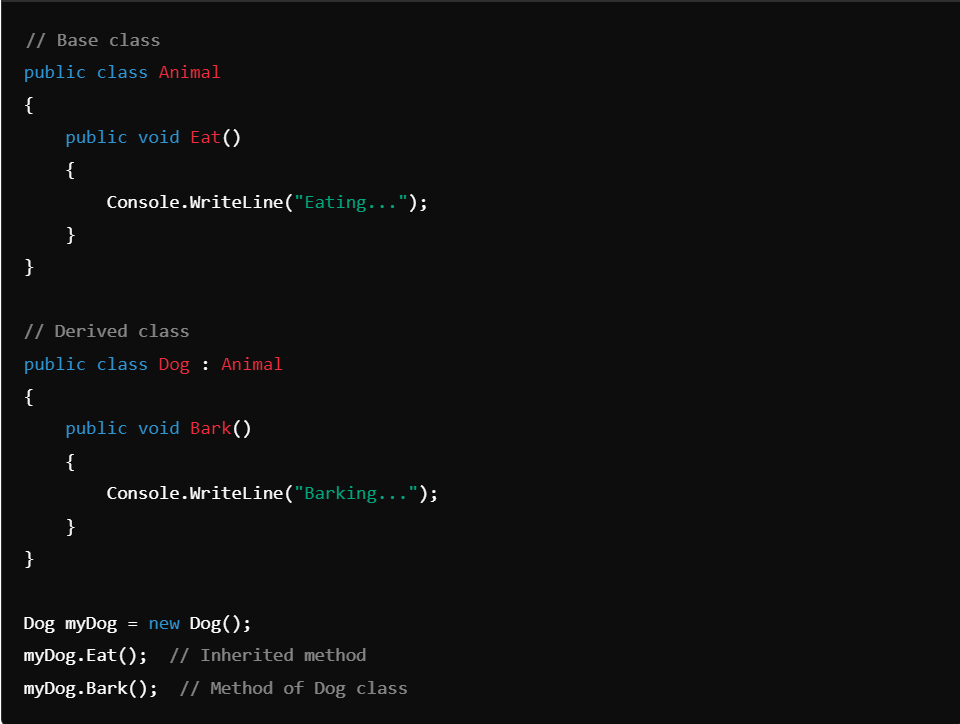
**Inheritance** is a fundamental principle of OOP that allows a new class (derived class) to inherit properties and methods from an existing class (base class). This promotes code reusability and establishes a relationship between the base and derived classes.



Not Physically and Logically correct.

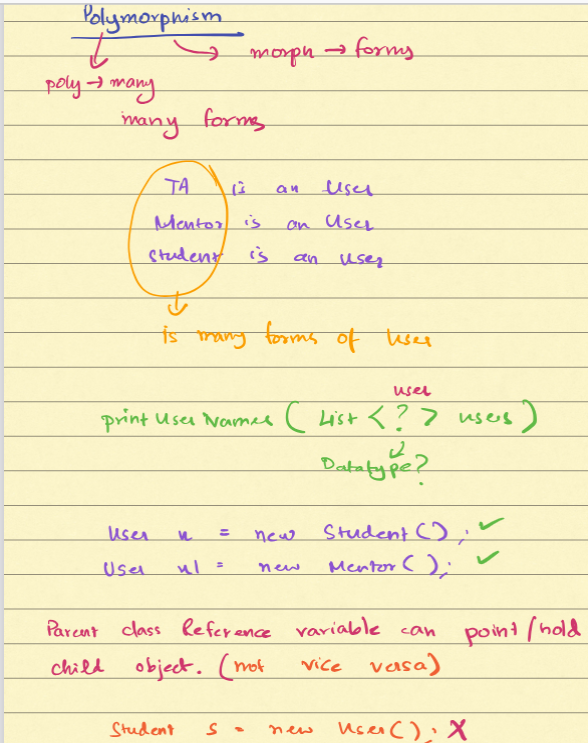
**Benefits of Inheritance:**

* **Code Reusability**: Inherit common functionality from base classes, reducing code duplication.
* **Extensibility**: Extend base class functionality in derived classes without modifying the base class.
* **Maintainability**: Easier to manage and update code since common functionality is centralized.



**What is polymorphism in OOP, and what are its types?**

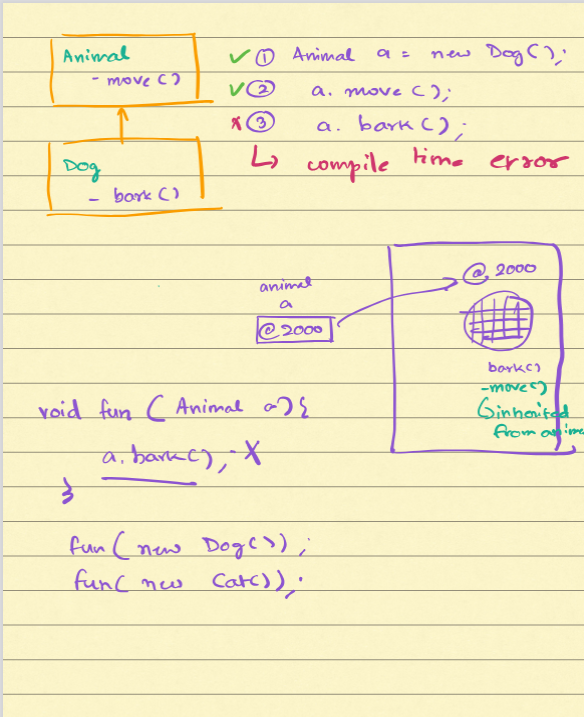
**Polymorphism** allows objects from different classes to be treated as if they belong to the same class. It lets you use a single interface to perform different actions, with the exact action depending on the specific object being used.



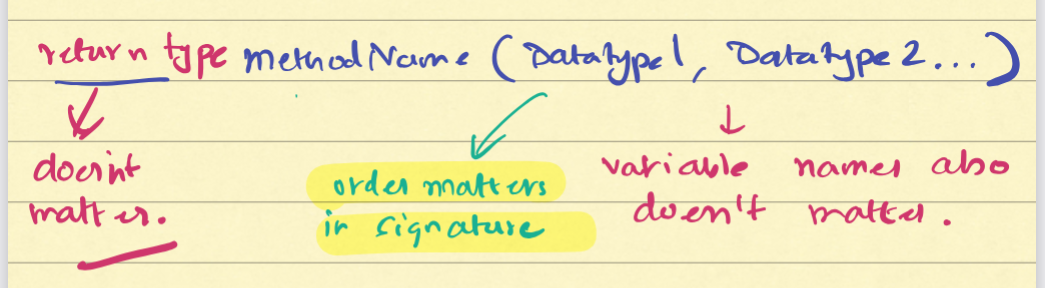
Consider a class named User with different roles:

* **TA (Teaching Assistant)** is a User.
* **Mentor** is a User.
* **Student** is a User.

These different roles (TA, Mentor, Student) are various forms of the base class User. Each role can perform certain actions that are specific to it but also share common behaviors defined in the User class.



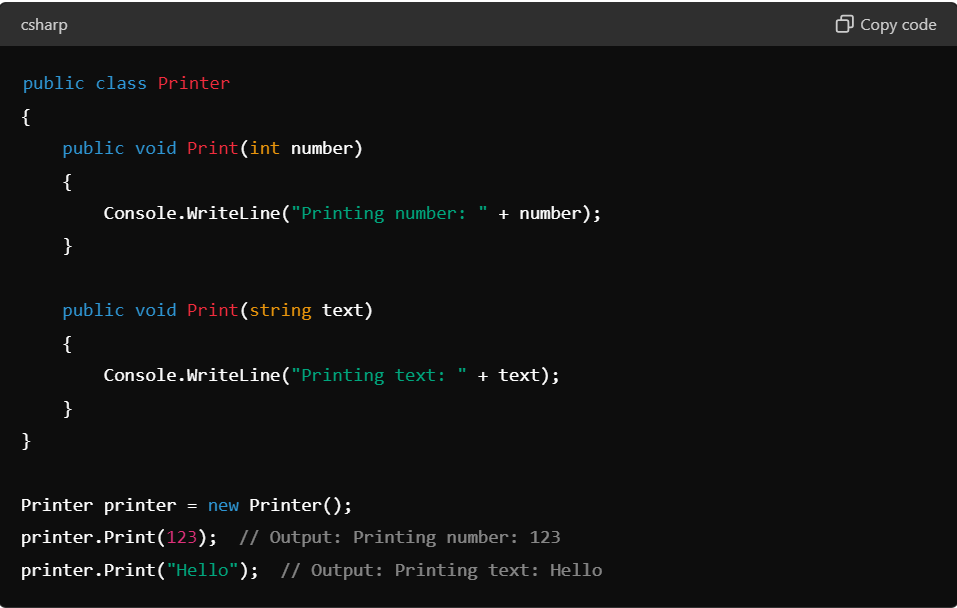
**Types of Polymorphism:**

* **Compile-Time Polymorphism (Static Binding)**: Achieved through method overloading and operator overloading.  
  
* **Run-Time Polymorphism (Dynamic Binding)**: Achieved through method overriding, using virtual methods and inheritance.

**What is the difference between method overloading and method overriding?**

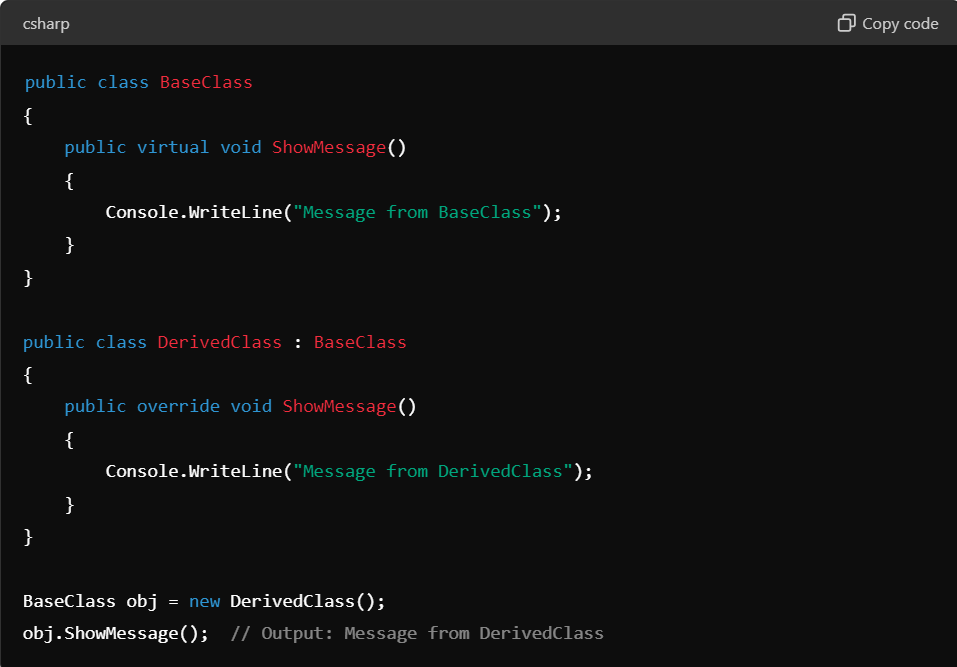
**Method Overloading:**

* **Definition**: Multiple methods with the same name but different parameters within the same class.
* **Binding**: Compile-time.
* **Use Case**: Provide different ways to perform a similar operation.
* **Example**:



**Method Overriding:**

* **Definition**: A derived class provides a specific implementation of a method already defined in its base class.
* **Binding**: Run-time.
* **Use Case**: Change or extend the base class functionality.
* **Example**:

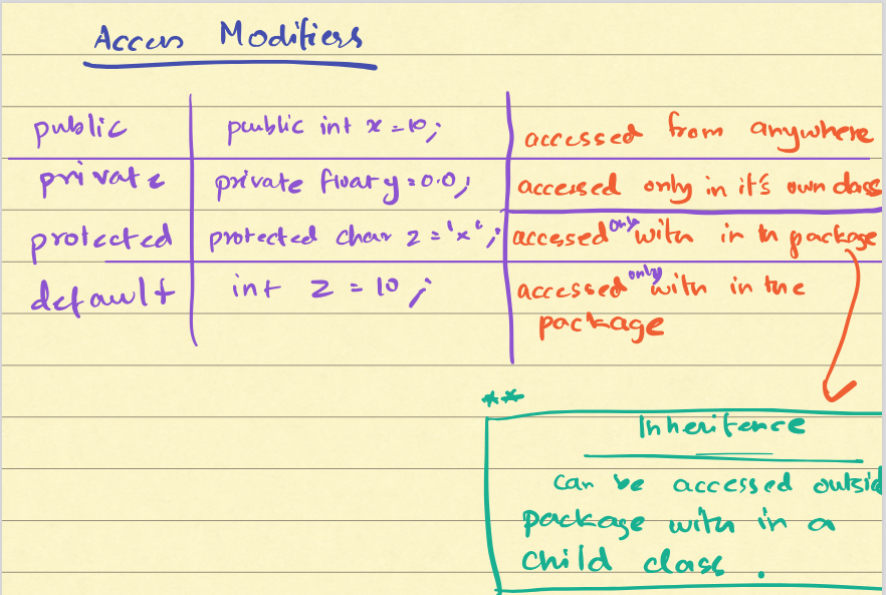


**What are access modifiers in OOP, and why are they important?**

**Access modifiers** are keywords used to define the accessibility of classes, methods, and other members in OOP. They help in encapsulating the data and methods, restricting access to them as needed. This ensures data integrity and security, preventing unintended interference and misuse.

**Importance:**

* **Encapsulation**: Protects the internal state of an object.
* **Security**: Restricts access to sensitive data.
* **Control**: Defines the level of access control over class members.
* **Maintainability**: Makes the code more maintainable by specifying access levels.

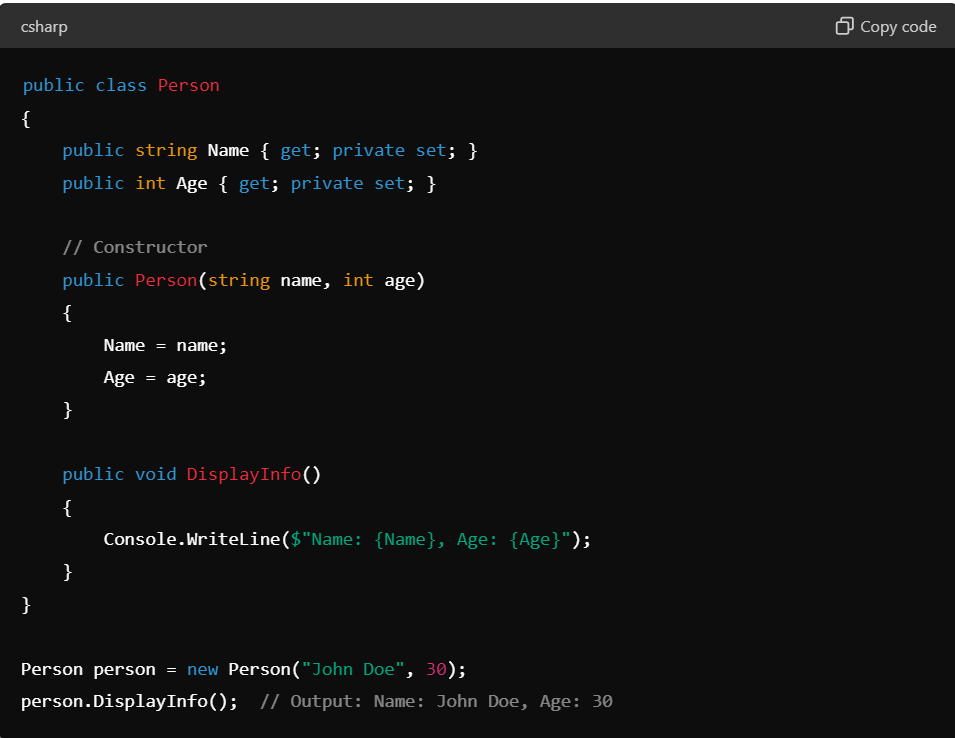


**What are constructors and what is the role of constructors in class design?**

**Constructors** are special methods in a class that are called when an instance of the class is created. They are used to initialize objects and set default values for the properties of the class.

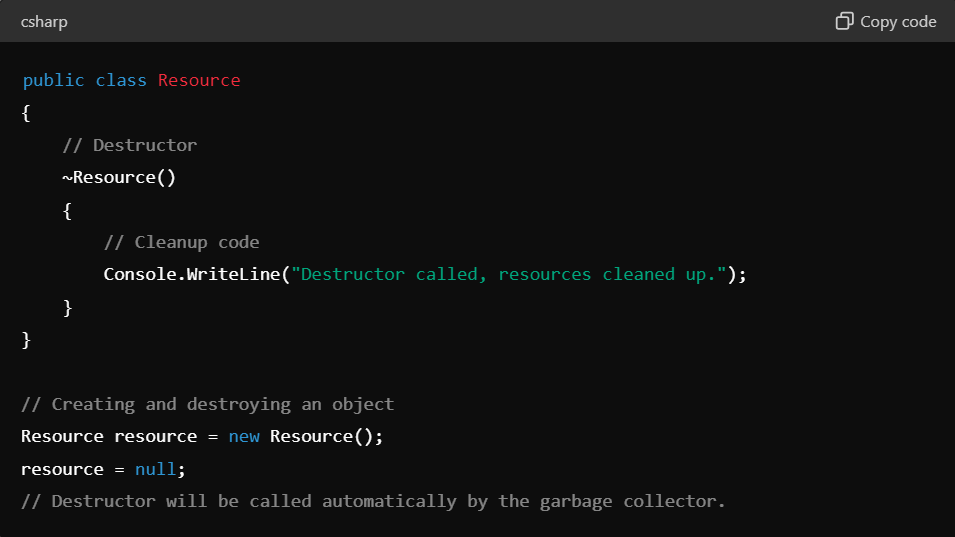
**Role of Constructors:**

* **Initialization**: Initialize object state.
* **Default Values**: Set default values for data members.
* **Dependency Injection**: Inject dependencies at the time of object creation.
* **Object Creation Control**: Control how objects are created and initialized.



**What is a destructor in OOP?**

**Destructor** is a special method that is called automatically when an object is destroyed. In C#, destructors are used to perform cleanup operations before the object is reclaimed by garbage collection.



**Explain the difference between public, private, protected, and internal access modifiers.**

* **Public**: The member is accessible from any other code.
* **Private**: The member is accessible only within the same class.
* **Protected**: The member is accessible within the same class and by derived class instances.
* **Internal**: The member is accessible within the same assembly, but not from another assembly.



**What is an Interface?**

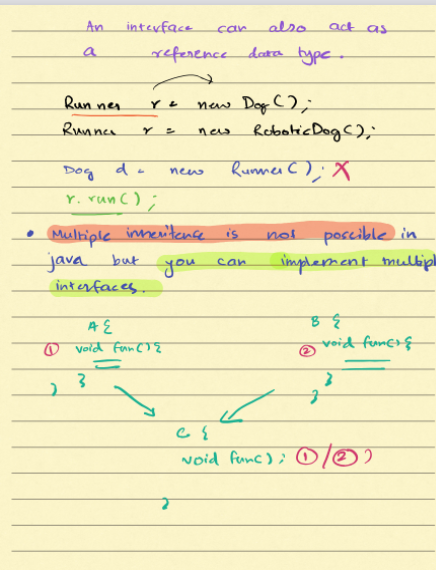
An **interface** in object-oriented programming (OOP) is a contract that defines a set of methods that a class must implement. Interfaces do not contain any implementation; they only specify what methods should be available. Interfaces are used to achieve abstraction and multiple inheritance in languages like C#.

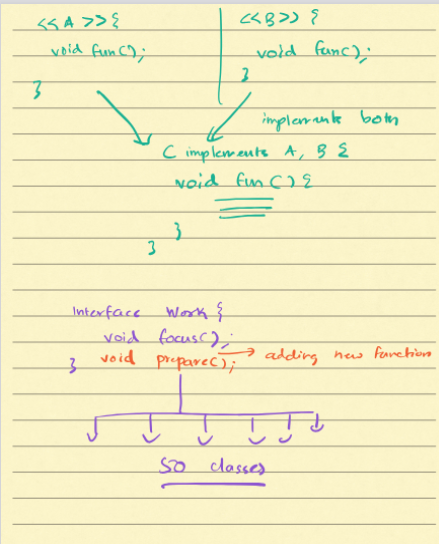
**Features of an Interface**

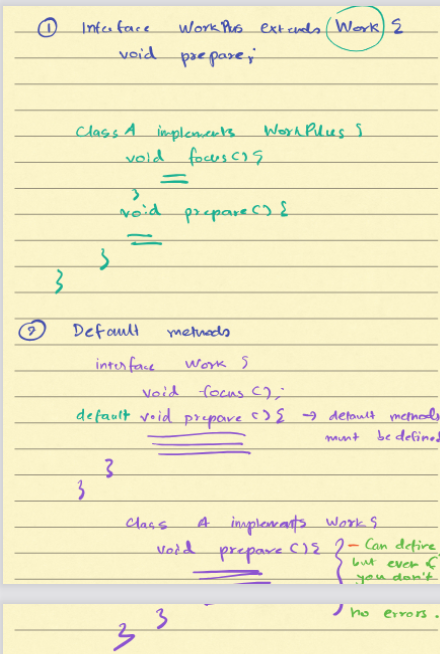
1. **No Implementation**: Interfaces only declare methods but do not provide any implementation.
2. **Multiple Inheritance**: A class can implement multiple interfaces, allowing for more flexible and modular designs.
3. **Pure Abstraction**: Interfaces define a contract for what a class can do, not how it does it.
4. **Access Modifiers**: Members of an interface are implicitly public, and they cannot have any access modifiers.  
     
   **Purpose of an Interface**
5. **Define Contracts**: Ensure that implementing classes provide specific functionality.
6. **Achieve Abstraction**: Provide a way to use objects without knowing their concrete types.
7. **Enable Multiple Inheritance**: Allow classes to inherit behavior from multiple sources.
8. **Promote Loose Coupling**: Decouple the code that uses the objects from the code that implements the objects.  
     
   **Usage of Interfaces**

Interfaces are used to define capabilities that can be shared across different classes, promoting code reusability and flexibility. They are particularly useful in scenarios where multiple classes share common behavior but have different implementations.









Default method in Interface

**When to Use Interfaces**

1. **Shared Functionality**: When multiple classes share common methods but have different implementations.
2. **Multiple Inheritance**: When you need to inherit behavior from multiple sources.
3. **Loose Coupling**: When you want to decouple code that uses objects from code that implements objects.
4. **Define Contracts**: When you need to define a contract that multiple classes must follow.

**Summary**

* **Interface**: A contract that defines methods and properties without implementation.
* **Multiple Inheritance**: Allows a class to implement multiple interfaces.
* **Pure Abstraction**: Provides a way to use objects without knowing their concrete types.
* **Purpose**: Defines contracts, achieves abstraction, promotes loose coupling, and enables multiple inheritance.

Interfaces are a powerful feature in OOP that help in designing flexible, modular, and maintainable systems by providing a way to define and enforce a common interface across multiple classes.

**What is an abstract class?**

An **abstract class** in object-oriented programming is a class that cannot be instantiated directly. It is designed to serve as a base class from which other classes can derive. Abstract classes are meant to represent generic concepts or base implementations, providing a common interface and some shared code to be used by derived classes.

**Features of an Abstract Class**

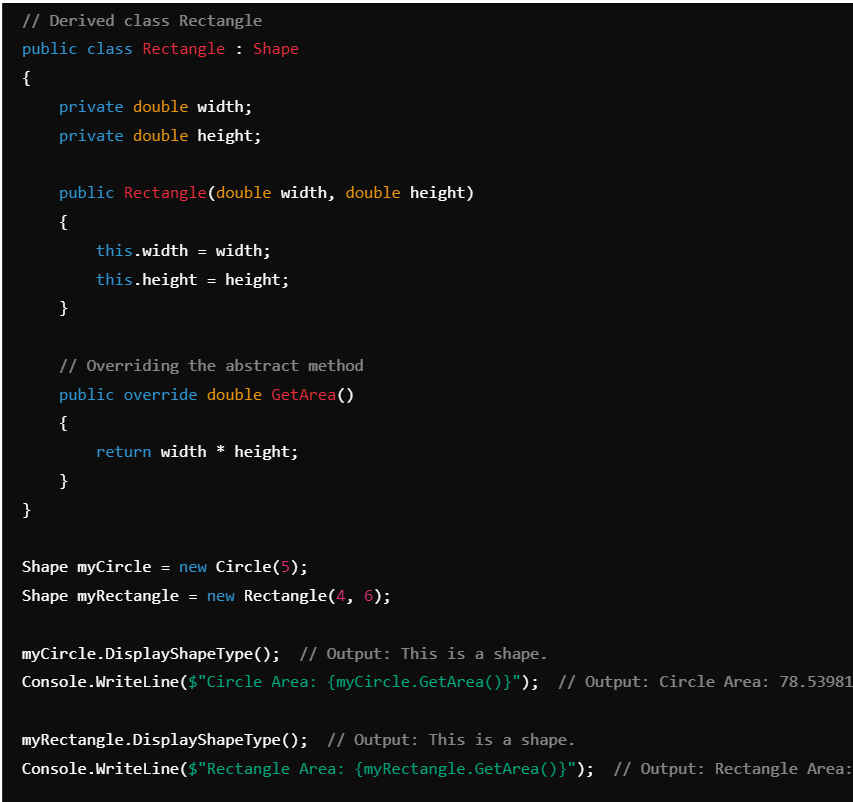
1. **Cannot be Instantiated**: An abstract class cannot be instantiated directly. You must derive a subclass and instantiate that subclass.
2. **Can Contain Abstract Methods**: These methods have no implementation in the abstract class and must be overridden in derived classes.
3. **Can Contain Non-Abstract Methods**: These methods have an implementation in the abstract class and can be used by derived classes.
4. **Can Contain Fields and Properties**: Like any other class, abstract classes can have fields, properties, and even constructors.

**Purpose of an Abstract Class**

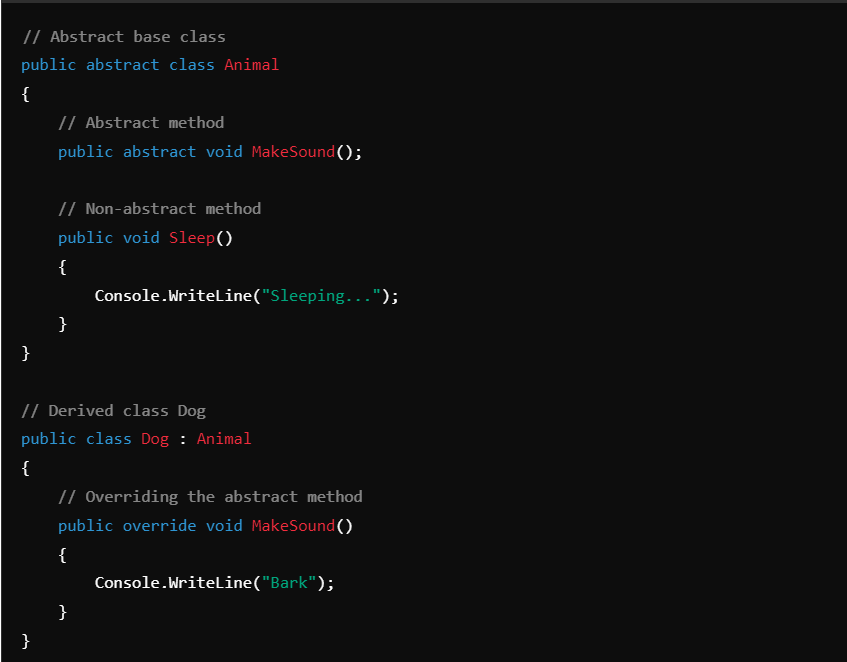
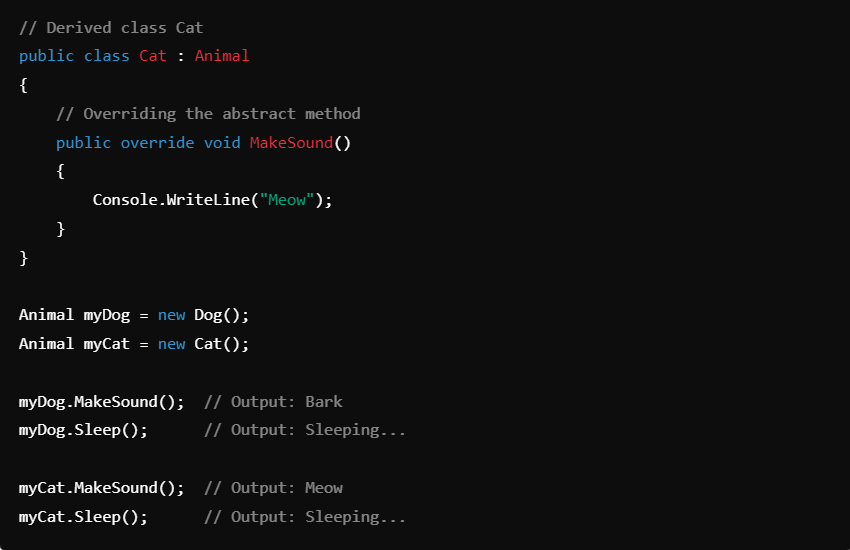
1. **Provide a Common Interface**: Abstract classes define a common interface for all derived classes.
2. **Reuse Code**: Abstract classes can include methods with implementation, allowing derived classes to reuse this code.
3. **Enforce Consistency**: By requiring derived classes to implement certain methods, abstract classes enforce consistency across different implementations.  
     
   **Usage of Abstract Classes**

Abstract classes are used when you have a base class that should not be instantiated on its own but instead provides a base set of functionality and interface for derived classes. This is particularly useful in scenarios where multiple classes share common behavior and interface but implement some specific behaviors differently.  
  
**Example 1: Shapes**





In this example, Shape is an abstract class with an abstract method GetArea() and a non-abstract method DisplayShapeType(). The Circle and Rectangle classes inherit from Shape and provide specific implementations of the GetArea() method.  
  
**Example 2: Animal Sounds**

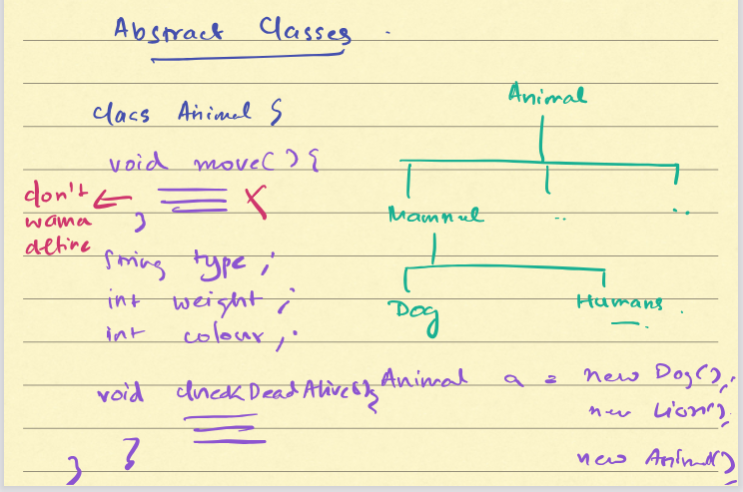
 

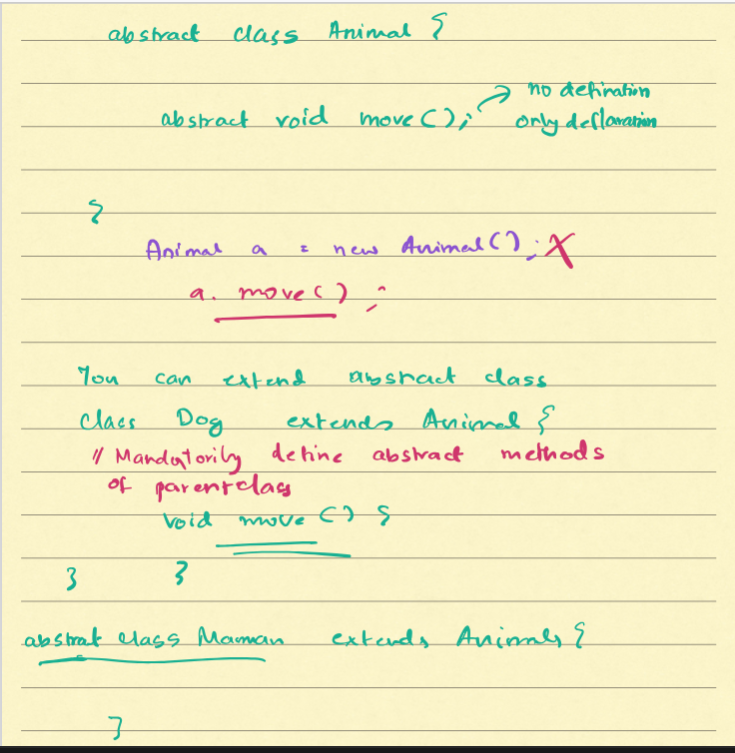
In this example, Animal is an abstract class with an abstract method MakeSound() and a non-abstract method Sleep(). The Dog and Cat classes inherit from Animal and provide specific implementations of the MakeSound() method.  
  
**When to Use Abstract Classes**

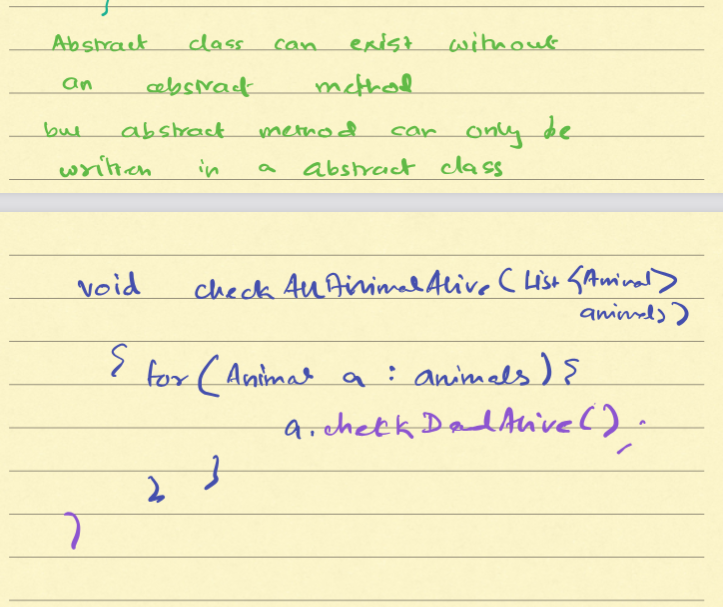
1. **Shared Code**: When multiple classes share common methods or properties but also have unique implementations.
2. **Base Class**: When you want to define a base class that should not be instantiated on its own but provides a common interface for derived classes.
3. **Enforce Implementation**: When you want to ensure that certain methods are implemented by all derived classes.  
     
   **Summary.**

* **Abstract Class**: Cannot be instantiated, serves as a base class.
* **Abstract Methods**: Defined in abstract classes, must be implemented in derived classes.
* **Non-Abstract Methods**: Can be included in abstract classes and used by derived classes.
* **Purpose**: Provides a common interface, enforces consistency, and allows code reuse.

Abstract classes play a crucial role in designing robust and maintainable object-oriented systems by providing a structured way to define and enforce a common interface and shared behavior across different classes.







**Difference Between Abstract Class and Interface**

