what is the difference between an abstract class and interface in c#

| **Aspect** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Definition** | A class that can have both abstract methods (no implementation) and concrete methods (with implementation). | A completely abstract type that can only have declarations for methods, properties, events, or indexers. |
| **Purpose** | To provide a common base class with shared functionality and enforce a contract for derived classes. | To define a contract that implementing classes must follow, without providing any implementation. |
| **Keyword** | Declared using the abstract keyword. | Declared using the interface keyword. |
| **Implementation** | Can include both abstract methods and methods with implementation. | Cannot include any method implementations (prior to C# 8.0). Starting with **C# 8.0**, default method implementations are supported in interfaces. |
| **Inheritance** | A class can inherit from only one abstract class (single inheritance). | A class can implement multiple interfaces (multiple inheritance). |
| **Access Modifiers** | Can have access modifiers (public, protected, etc.) for methods, properties, and fields. | All members are implicitly public and cannot have access modifiers. |
| **Fields** | Can contain fields (variables) and constants. | Cannot contain fields; only constants are allowed. |
| **Constructors** | Can have constructors. | Cannot have constructors. |
| **Use Case** | Use when a base class with common functionality is required to be shared among derived classes. | Use when multiple unrelated classes need to follow the same contract. |
| **Instance Creation** | Cannot create an instance of an abstract class. | Cannot create an instance of an interface. |
| **Compatibility** | A derived class must use the override keyword to implement abstract methods. | A class implementing an interface must implement all its members unless the interface provides default implementations (C# 8.0 and later). |

**Key Differences in Use Cases**

1. **When to Use an Abstract Class**:
   * If you need to provide **default behavior** or share common functionality among derived classes.
   * Example: A base Animal class with a Move() method that is common for all animals.
2. **When to Use an Interface**:
   * If you need to enforce a contract across multiple unrelated classes.
   * Example: A ILogging interface that different logging mechanisms (file logging, database logging) must implement.

**Combination of Both**

You can combine abstract classes and interfaces to leverage the strengths of both. For example:

* Use an abstract class to share common functionality.
* Use interfaces to enforce additional contracts.

interface IAnimal {

void Speak();

}

abstract class Animal {

public void Eat() {

Console.WriteLine("This animal eats food.");

}

}

class Dog : Animal, IAnimal {

public void Speak() {

Console.WriteLine("The dog barks.");

}

}

**Conclusion**

* **Abstract Class**: Use when you need a base class with shared functionality and a defined contract.
* **Interface**: Use when you need to define a contract for multiple unrelated classes to implement.

When to use Interface and when Abstract class in real applications?

**When to Use an Interface**

1. **Multiple Inheritance is Required**:
   * Use interfaces when a class needs to implement functionality from multiple sources since a class can implement multiple interfaces.
   * **Example**: A class Smartphone can implement both ICamera and IPhone interfaces.

interface ICamera {

void CapturePhoto();

}

interface IPhone {

void MakeCall(string number);

}

class Smartphone : ICamera, IPhone {

public void CapturePhoto() {

Console.WriteLine("Photo captured.");

}

public void MakeCall(string number) {

Console.WriteLine($"Calling {number}...");

}

}

**Unrelated Classes Need to Follow a Common Contract**:

* Use interfaces to define behavior for unrelated classes that do not share a common hierarchy.
* **Example**: A Logger system where FileLogger and DatabaseLogger implement a common ILogger interface.

interface ILogger {

void Log(string message);

}

class FileLogger : ILogger {

public void Log(string message) {

Console.WriteLine($"Logging to file: {message}");

}

}

class DatabaseLogger : ILogger {

public void Log(string message) {

Console.WriteLine($"Logging to database: {message}");

}

}

1. **Plug-and-Play Behavior**:
   * Use interfaces when you want to swap or replace components dynamically.
   * **Example**: Dependency injection in an application with different data access layers (ISqlDataAccess, IMongoDataAccess).
2. **Defining Cross-Cutting Concerns**:
   * Use interfaces for features like logging, authentication, or authorization that span multiple components.
   * **Example**: An IAuthorization interface implemented by various authentication mechanisms.

**Use an Interface when:**

1. **Defining a contract or capability**:
   * Interfaces specify a set of methods that a class must implement, without providing any implementation.
   * Ideal for defining behaviors or capabilities that can be shared by unrelated classes.
   * Example: Runnable, Comparable, or defining a PaymentProcessor interface for payment systems.
2. **Multiple inheritance of behavior is required**:
   * A class in many programming languages (like Java, C#) can implement multiple interfaces but can only inherit from one abstract class.
3. **Promoting decoupling**:
   * Interfaces are excellent for dependency injection and achieving loose coupling between components in your application.
4. **Future-proofing design**:
   * Using interfaces ensures flexibility to add multiple implementations without modifying existing ones.

**Use an Abstract Class when:**

1. **Shared state or default behavior**:
   * Abstract classes can have both abstract methods and concrete (implemented) methods.
   * Useful when you want to provide default behavior for subclasses.
2. **Defining a base class with shared properties**:
   * If classes share common state or logic, an abstract class provides a centralized place to define them.
   * Example: A Shape abstract class with shared properties like color and area().
3. **Extending a single inheritance hierarchy**:
   * Use abstract classes when the design requires strict hierarchical relationships (is-a relationships).
4. **Adding or updating behaviour without breaking implementations**:
   * Since interfaces require all implementing classes to provide method definitions, adding methods later can be problematic. Abstract classes allow adding non-abstract methods without affecting subclasses.
5. **Comparison:**

| **Feature** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| **Purpose** | Define a contract (what to do) | Define a base class (how and what to do) |
| **Multiple Inheritance** | Yes | No |
| **Fields** | Constants only | Instance and static fields allowed |
| **Default Methods** | Supported (since Java 8) | Supported |
| **Constructor** | Not allowed | Allowed |

**Real-world Examples:**

* **Interface**:
  + Payment gateways (PayPal, Stripe) implementing a PaymentProcessor interface.
  + Vehicles (Car, Bike, Truck) implementing a Movable interface.
* **Abstract Class**:
  + Abstract Vehicle class with shared attributes like speed, fuelType, and concrete methods like startEngine().

By assessing the design needs of your application, you can determine which approach better suits your use case.

**Use an Interface when:**

1. **Contract without Implementation:**
   * You want to define a contract or a blueprint for classes to implement but without providing any shared functionality.
   * Example: You might have an interface like Flyable for classes like Bird and Airplane to implement.
2. **Multiple Inheritance:**
   * The class needs to implement multiple behaviors from different sources. In many languages (e.g., Java), a class can implement multiple interfaces but can only inherit from one class.
   * Example: A Drone class might implement both Flyable and RemoteControllable.
3. **API Design for Flexibility:**
   * You want the implementing classes to have full control over their implementation.
   * Example: When creating a library or plugin system, interfaces define what the users of your library should implement.
4. **Loose Coupling:**
   * Interfaces ensure that the code remains loosely coupled. This is useful in dependency injection and for achieving polymorphism.
   * Example: A PaymentProcessor interface could have implementations like PayPalProcessor or StripeProcessor.

**Use an Abstract Class when:**

1. **Shared Code/Logic:**
   * You want to define common functionality or state that should be shared among all subclasses.
   * Example: An abstract class Vehicle might have a common property like speed and a method startEngine().
2. **Need for Partial Implementation:**
   * Some methods have a default implementation, but you also want subclasses to provide their specific implementations for others.
   * Example: An abstract class Shape might have a default implementation for getName() but leave calculateArea() abstract.
3. **Encapsulation of State:**
   * Abstract classes can have fields and fully implemented methods, allowing you to encapsulate state and behavior together.
   * Example: A GameCharacter abstract class might have properties like health and methods like takeDamage().
4. **Easier Maintenance:**
   * When there’s a possibility that shared functionality will grow or change in the future, an abstract class is better as it avoids repeating the same code across multiple classes.
5. **Single Inheritance Model:**
   * If a class hierarchy naturally forms a tree-like structure, abstract classes are more suitable.

**Key Points to Consider:**

| **Feature** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| **Multiple inheritance** | Yes (implements multiple) | No (extends only one) |
| **Default implementations** | Limited (e.g., default methods in Java 8+) | Yes (fully supported) |
| **State/Fields** | Not allowed | Allowed |
| **Flexibility in implementation** | High | Moderate |
| **Backward compatibility** | Harder to maintain | Easier to maintain |

Why to create Interfaces in real applications?

**1. Defining Contracts or Blueprints**

* An interface defines a **contract** that specifies what methods a class must implement without dictating how they should be implemented.
* This ensures consistency across different classes that share common behavior.

**2. Enabling Multiple Inheritance**

* In many languages like Java, a class can only inherit from one superclass but can implement multiple interfaces.
* This allows a class to **inherit multiple behaviors**.

**3. Supporting Dependency Injection**

* Interfaces allow you to implement **dependency injection** by decoupling the code from specific implementations.
* This makes the codebase more modular and easier to test or extend

**4. Achieving Loose Coupling**

* Interfaces decouple the implementation from the client code, making the system more flexible and easier to modify or replace parts of the codebase without affecting the rest.

**5. Facilitating Polymorphism**

* Interfaces support **polymorphism**, allowing you to treat different classes with the same behavior uniformly.

**6. Enhancing Testability**

* Using interfaces makes unit testing easier because you can provide mock implementations for testing purposes.

**7. Encouraging Extensibility**

* Interfaces make it easier to extend systems by allowing new implementations to be added without modifying existing code.

**8. Standardizing Cross-Team Collaboration**

* In large projects with multiple teams, interfaces serve as **clear contracts** for collaboration. Each team knows what methods need to be implemented without worrying about the internal details.

**10. Future-Proofing Code**

* Using interfaces ensures that your application can adapt to future changes by adding new implementations without affecting existing code.
* This aligns with the **Open/Closed Principle** of SOLID design principles: "Open for extension, closed for modification."

Can we define body of Interfaces methods c# ?

Yes, in **C#**, starting with **C# 8.0**, you can define the **body** of methods in interfaces using **default implementations**. This allows you to provide a method implementation within the interface itself, which implementing classes can either use as-is or override.

**How Default Implementations Work in Interfaces**

1. **Default Implementations**:
   * Methods in an interface can have a body, meaning they provide a default behavior.
   * Implementing classes are not required to override these methods unless they want custom behavior.

**Static Methods in Interfaces**:

* Interfaces can have **static methods** starting from C# 8.0.
* These methods belong to the interface itself and are not related to implementing classes.

**Properties and Other Members**:

* Interfaces can also define **default implementations for properties**.