Abstract Classes and Interfaces

In Java, both Abstract Classes and Interfaces are used to achieve abstraction and define the structure for other classes to implement. However, there are key differences between them in terms of functionality and use cases. Let's explore these differences and guidelines on when to use one over the other.

Differences between Abstract Class and Interface:

Feature	**Abstract Class**	**Interface**
Keyword Used	Declared using the `abstract` keyword.	Declared using the `interface` keyword.
Method Implementation	Can have both abstract (unimplemented) and concrete (implemented) methods.	Only contains abstract methods until Java 7; starting from Java 8, it can also have default and static methods with implementations.
Multiple Inheritance	A class can extend only one abstract class (single inheritance).	A class can implement multiple interfaces (multiple inheritance).
Fields (Variables)	Can have instance variables, which can be non-final.	Can only have `public`, `static`, and `final` variables (essentially constants).
Access Modifiers	Methods can have any access modifier (e.g., `public`, `protected`, `private`).	All methods are implicitly `public`, except `private` methods added from Java 9 onward for utility purposes.
Constructors	Can have constructors.	Cannot have constructors.
Inheritance Relationship	Represents an "is-a" relationship.	Represents an "ability" or "can-do" relationship.
Use Case	Best suited when there is shared behavior or common code among subclasses.	Best suited when you want to define a contract that different classes can implement.

When to Use an Interface vs. an Abstract Class:

1. **Use an Interface When:**
    - You need to define a contract or capability that can be shared across different classes, even if they are not related. For example, `Runnable` and `Comparable` are interfaces that define capabilities.
    - You require multiple inheritance of behavior. Since a class can implement multiple interfaces, this is ideal for scenarios where a class needs to inherit behavior from multiple sources.
    - You want to achieve loose coupling between the components. Interfaces define a clear contract, allowing different implementations without being tied to a specific class hierarchy.
2. **Use an Abstract Class When:**
    - You need to provide common functionality or shared code to related classes, along with some level of abstraction. An abstract class allows you to define both abstract methods (to be implemented by subclasses) and concrete methods (common behavior shared by all subclasses).
    - You want to maintain some state using instance variables, which can be inherited by subclasses.
    - Your classes are closely related and share a common base, for example, a `Shape` class that has subclasses like `Circle`, `Square`, etc.

Example Use Cases:

- **Abstract Class:**

```
abstract class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    abstract void makeSound(); // Abstract method
    void eat() { // Concrete method
        System.out.println(name + " is eating.");
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    @Override
    void makeSound() {
        System.out.println(name + " barks.");
    }
}
```

- **Interface:**

```
interface Flyable {
    void fly(); // Abstract method
}

interface Swimmable {
    void swim(); // Abstract method
}

class Duck implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("Duck is flying.");
    }

    @Override
    public void swim() {
        System.out.println("Duck is swimming.");
        }
    }
```

In the examples above:

- **Abstract Class (`Animal`)** is used to group related classes (like `Dog`) that share common behavior (`eat()` method) and have some abstract behavior (`makeSound()` method).
- **Interfaces (`Flyable`, `Swimmable`)** define abilities that can be shared across different classes (like `Duck`) without forcing those classes into a strict hierarchy.

Conclusion:

- **Use an Interface** when you want to define a role or capability that can be implemented by unrelated classes and to support multiple inheritance of behavior.
- **Use an Abstract Class** when you have related classes that should share common behavior while also enforcing some level of abstraction.