1) What is an interface in Java?

Ans:

1. Contractual Obligation: Interfaces define a contract that classes must adhere to. Classes implementing an interface are obligated to provide implementations for all the abstract methods declared in the interface.

2. Abstraction and Modularity: Interfaces promote code abstraction by separating the specification of behaviour from its implementation. They allow for modularity and loose coupling by providing a clear and standardized way for classes to interact.

3. Polymorphism and Flexibility: Interfaces enable polymorphism in Java. A variable of an interface type can refer to any instance of a class that implements that interface, allowing for flexible and interchangeable behaviour.

4. Multiple Inheritance: Unlike classes, Java allows multiple inheritance of interfaces. A class can implement multiple interfaces, inheriting the abstract methods and default implementations from each interface. This supports code reuse and enhances flexibility in class design.

5. API Design and Compatibility: Interfaces are commonly used in API design to define a contract that clients must adhere to. By programming to interfaces, developers can write code that is compatible with multiple implementations, promoting interoperability and enabling easier swapping of implementations.

2) Which modifiers are allowed for methods in an interface? Explain with an example

Ans:
In an interface, methods can have the following modifiers:

1. `public`: Methods in an interface are implicitly `public`. This means they can be accessed and called from anywhere in the codebase.

2. `default`: Java 8 introduced the `default` modifier for methods in interfaces. Default methods provide a default implementation that can be used by implementing classes. They allow interfaces to evolve without breaking the compatibility of existing implementations.

3. `static`: Java 8 also introduced the `static` modifier for methods in interfaces. Static methods can be invoked directly on the interface itself, without the need for an instance of the implementing class. They are typically used to provide utility methods or helper functions related to the interface.

Example illustrating the use of these modifiers in an interface:

```
public interface MyInterface {

    // Public abstract method

    public void myMethod();



    // Default method

    default void myDefaultMethod() {

        System.out.println("Default method implementation");

    }



    // Static method

    static void myStaticMethod() {

        System.out.println("Static method implementation");

    }

}
```

3) What is the use of the interface in Java? Or, why do we use an interface in Java?

Ans:

Interfaces serve several purposes in Java, and they are used for the following reasons:

1. Defining Contracts: Interfaces are used to define a contract or a set of rules that classes must follow. By implementing an interface, a class agrees to fulfill the contractual obligations specified by the interface, ensuring a consistent behavior across multiple classes.

2. Achieving Abstraction: Interfaces promote code abstraction by separating the specification of behavior from its implementation. They allow programmers to focus on what needs to be done (the method signatures) rather than how it is done (the implementation details). This enhances code maintainability and modularity.

3. Enabling Polymorphism: Interfaces are instrumental in achieving polymorphism in Java. They provide a way to achieve multiple forms of behavior through a single interface type. This allows different classes implementing the same interface to be used interchangeably, enhancing flexibility and code reuse.

4. Supporting API Design: Interfaces are commonly used in API design to define a contract that clients must adhere to. By programming to interfaces rather than concrete classes, developers can write code that is compatible with multiple implementations. This promotes interoperability, modularity, and allows for easier swapping of implementations.

5. Facilitating Unit Testing: Interfaces facilitate unit testing by providing a clear separation between dependencies and the code being tested. By using interfaces as dependencies, test cases can easily mock or stub those dependencies to isolate and test specific behavior.

6. Future Compatibility and Evolution: Interfaces enable the evolution of code without breaking the compatibility of existing implementations. By adding default methods in newer versions of interfaces, existing implementations can continue to function without modification, providing forward compatibility.

4) What is the difference between abstract class and interface in Java?

Ans:

The differences between abstract class and interface are:

1. Method Implementation:

- Abstract Class: Abstract classes can have both abstract and non-abstract methods. Abstract methods have no implementation and must be overridden by the subclasses. Non-abstract methods can have implementations and can be directly called by the subclasses.

- Interface: Interfaces can only have abstract methods, which do not contain any implementation. The implementing classes are responsible for providing the implementation for all the methods defined in the interface.

2. Multiple Inheritance:

- Abstract Class: A class can extend only one abstract class, as Java does not support multiple inheritance of classes.

- Interface: A class can implement multiple interfaces, allowing for multiple inheritance of behavior. This allows a class to inherit and implement the methods from multiple interfaces.

3. Constructors:

- Abstract Class: Abstract classes can have constructors that are used for initializing the instance variables of the class. These constructors are called when a concrete subclass is instantiated.

- Interface: Interfaces cannot have constructors, as they do not define instance variables or maintain state.

4. Access Modifiers:

- Abstract Class: Abstract classes can have different access modifiers for their members (methods, variables). They can be public, protected, private, or default (package-private).

- Interface: All the methods in an interface are implicitly public. The variables in an interface are implicitly public, static, and final.

5. Extensibility:

- Abstract Class: Subclasses extend abstract classes using the `extends` keyword. They inherit both the implemented and non-implemented methods of the abstract class.

- Interface: Classes implement interfaces using the `implements` keyword. They must provide implementation for all the methods defined in the interface.

6. Usage:

- Abstract Class: Abstract classes are useful when creating a common base class that provides default behavior or state for its subclasses. They are well-suited for situations where inheritance represents an "is-a" relationship.

- Interface: Interfaces are used when defining a contract that classes must adhere to, promoting loose coupling and allowing classes to provide different behaviors through multiple interfaces. They are well-suited for situations where inheritance represents a "can-do" relationship.