Lecture 9

Abstract class in Java

- A class which is declared with the **abstract** keyword is known as an **abstract** class in Java. It can have abstract and non-abstract methods (method with the body).
- Before learning the Java abstract class, let's understand the **abstraction** in Java first.

Abstraction

- Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

- Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

- Abstraction can be achieved with either **abstract classes** or **interfaces**.

Abstract Classes and Methods

- The **abstract** keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

- To access the abstract class, it must be inherited from another class.

Abstract Classes

- A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.
- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Abstract Method

- A method which is declared as abstract and does not have implementation is known as an abstract method.

  abstract void printStatus();
  //no method body and abstract

Abstract Method

- An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

- If there is an abstract method in a class, that class must be abstract.

- If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Why And When To Use Abstract Classes and Methods?

- To achieve security - hide certain details and only show the important details of an object.
- **Note:** Abstraction can also be achieved with Interfaces.

Interface

- Another way to achieve **abstraction** in Java, is with interfaces.

- An **interface** is a completely "**abstract class**" that is used to group related methods with empty bodies

- To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the **implements** keyword (instead of **extends**). The body of the interface method is provided by the "implement" class.

Interface

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

- The interface in Java is *a mechanism to achieve* ***abstraction***. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple **inheritance in Java**.

- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

- Java Interface also represents the IS-A relationship.

Why To Use Interfaces?

- To achieve security - hide certain details and only show the important details of an object (interface).
- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

When To Use Interfaces?

- Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces.

- **Note:** To implement multiple interfaces, separate them with a comma (see example below).

How to declare an interface?

- An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>{
    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

Internal addition by the compiler

- The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

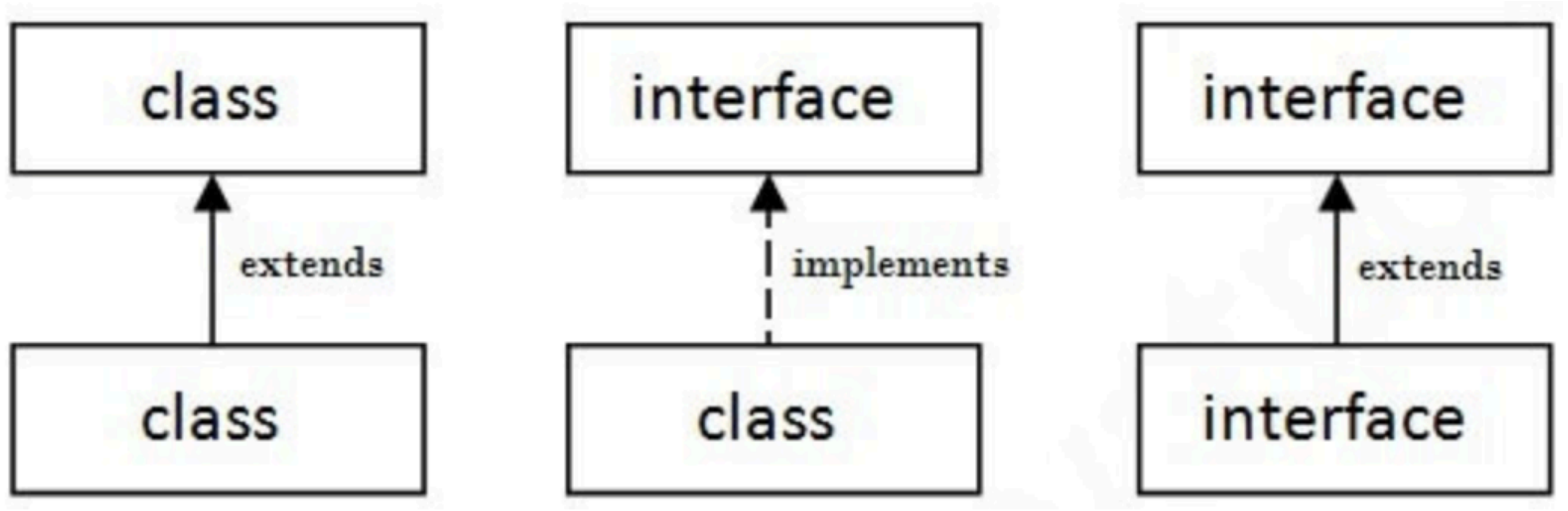
- In other words, Interface fields are public, static and final by default, and the methods are public and abstract.

Interfaces

- Like **abstract classes**, interfaces **cannot** be used to create objects.
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default **abstract** and **public**
- Interface attributes are by default **public**, **static** and **final**
- An interface cannot contain a constructor (as it cannot be used to create objects)

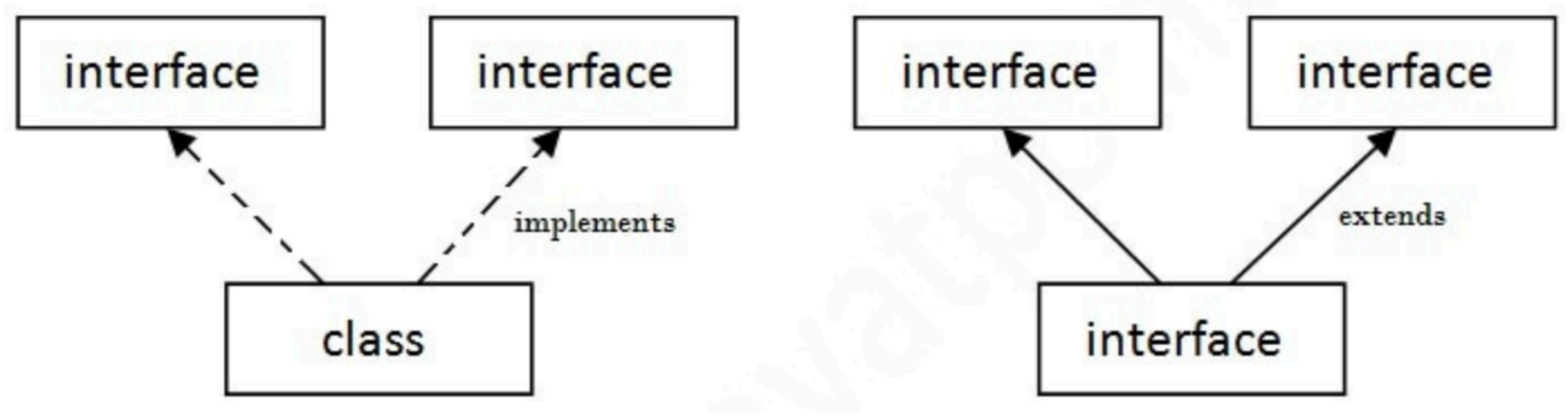
The relationship between classes and interfaces

- As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Multiple inheritance

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple inheritance

- As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity.

- However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

Multiple inheritance

- We can have method body in interface. But we need to make it default method.
- We can have static method in interface.