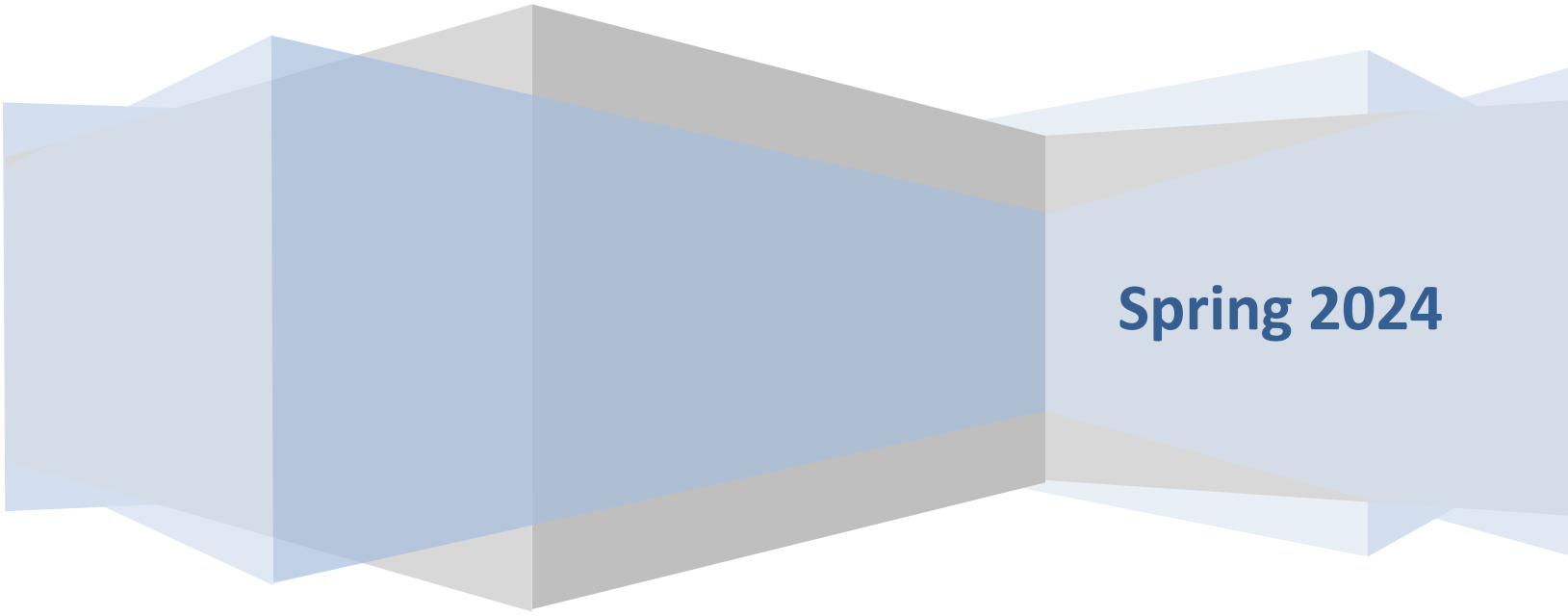
Northern University, Nowshera

# Interfaces in Java

## Week # 15 - Lecture 29 - 30



Spring 2024

# Learning Objectives

- **Interfaces**
- **Why interfaces?**
- **Implementing interfaces**
- **Default methods**
- **Extending interfaces**
- **Extending multiple interfaces**

# Interfaces

The word is probably familiar to you. For example, most computer programs and games have interfaces. In a broad sense, an interface is a kind of 'remote control' that connects two interacting parties. A simple example of an interface in everyday life is a TV remote control. It connects two object — a person and a TV — and performs different tasks: turn up or turn down the volume, switch channels, and turn on or turn off the TV.

One party (the person) needs to access the interface (press a button on the remote control) to cause the second party to perform the action.

For example, to make the TV change to the next channel. What's more, the user doesn't need to know how the TV is organized or how the channel changing process is implemented internally. The only thing the user has access to, is the *interface*. The main objective is to get the desired result.

What does this have to do with programming and Java?

Creating an interface is very similar to creating a regular class, but instead using the word class, we indicate the word interface. Let's look at the simplest Java interface, see how it works, and why we would need it:

```
public interface CanSwim {

     public void swim();
}
```

We've created a CanSwim interface. It's a bit like our remote control, but with one 'button': the swim() method. But how do we use this remote controller?

To do this, we need **to implement** a method, i.e. our remote control button. To use an interface, some classes in our program must implement its methods.

Let's invent a class whose objects 'can swim'. For example, a Duck class fits:

```
public class Duck implements CanSwim {

    public void swim() {
        System.out.println("Duck, swim!");
    }

    public static void main(String[] args) {

        Duck duck = new Duck();
        duck.swim();
    }
```

What do we see here?

The Duck class is 'associated' with the CanSwim interface by the **implements** keyword. You may recall that we used a similar mechanism to associate two classes through inheritance, but in that case we used the word *extends*. For complete clarity, we can translate '**public class Duck implements CanSwim**' literally as: 'The public Duck class implements the CanSwim interface'. This means that a class associated with an interface must implement all of its methods. *Note:* our Duck class, just like the CanSwim interface, has a swim() method, and it contains some logic. This is a mandatory requirement. If we just write public class Duck implements CanSwim without creating a swim() method in the Duck class, the compiler will give us an error:

**Why? Why does this happen?** If we explain the error using the TV example, it would be like handing someone a TV remote control with a 'change channel' button that can't change channels. You could press the button as much as you like, but it won't work. The remote control doesn't change channels by itself: it only sends a signal to the TV, which implements the complex process of channel changing. And so it is with our duck: it must know how to swim so it can be called using the CanSwim interface.

If it doesn't know how, the CanSwim interface doesn't connect the two parties — the person and the program. The person won't be able to use the swim() method to make a Duck swim inside the program. Now you understand more clearly what interfaces are for.

**An interface describes the behavior** that classes implementing the interface must have. 'Behavior' is a collection of methods.

An interface is a **reference type** in Java. It is similar to class. It is a **collection of abstract methods**. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface **may also contain constants, default methods, static methods, and nested types**. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behavior of an object and an interface contains behaviors that a class implements.

Like a class, an interface can have methods and variables, but the **methods declared in interface are by default abstract** (only method signature, no body).

- Interfaces specify what a class must do (not how). It is the blueprint of the class.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then **class must be declared abstract**.

### However, an interface is different from a class in several ways, including:

- We cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

# Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface –

## Example

Following is an example of an interface.

```
import java.lang.*;

public interface NameOfInterface {

  // Any number of final, static fields

  // Any number of abstract method declarations\

}
```

Interfaces have the following properties –

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example

```
interface Animal {

  public void eat();

  public void travel();

}
```

# Why do we use interface?

Abstract classes may contain non-final variables, whereas variables in interface are final and static.

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

# Implementing Interfaces

When a class implements an interface, you can think of ***the class as signing a contract***, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements keyword** to implement an interface. The implements keyword appears in the class declaration following the **extends** portion of the declaration.

Example

```
public class MammalInt implements Animal {

  public void eat() {

    System.out.println("Mammal eats");

  }

  public void travel() {

    System.out.println("Mammal travels");

  }
```

```
  public int noOfLegs() {
    return 0;
  }
  public static void main(String args[]) {
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

This will produce the following result –

### Output

**Mammal eats**

**Mammal travels**

## Example: Vehicle interface

Let's consider the example of vehicles like bicycle, car, bike……, they have common functionalities. So, we make an interface and put all these common functionalities. And lets Bicylce, Bike, car ….etc implement all these functionalities in their own class in their own way.

```
import java.io.*;

interface Vehicle {

    // all are the abstract methods.
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}
```

```
class Bicycle implements Vehicle{

    int speed;
    int gear;

     // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed
              + " gear: " + gear);
    }
}

class Bike implements Vehicle {

    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }
```

```
    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed
             + " gear: " + gear);
    }

}
class MainClass {

    public static void main (String[] args) {

        // creating an inatance of Bicycle
        // doing some operations
        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.print("Bicycle present state:- ");
        bicycle.printStates();

        // creating instance of bike.
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);

        System.out.print("\nBike present state:- ");
        bike.printStates();
    }
}
```

**Output;**

```
Bicycle present state:- speed: 2 gear: 2
Bike present state:- speed: 1 gear: 1
```

**When overriding methods defined in interfaces, there are several rules to be followed –**

- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

**When implementing interfaces, there are several rules –**

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

# Another Example: Messenger interface

If we want to create several messengers, the easiest thing to do is to creating a Messenger interface. What does every messenger need? At a basic level, they must be able to receive and send messages.

```
public interface Messenger{
     public void sendMessage();
     public void getMessage();
}
```

Now we can simply create our messenger classes that implement the corresponding interface. The compiler itself will 'force' us to implement them in our classes.

### Telegram:

```
public class Telegram implements Messenger {

    public void sendMessage() {

        System.out.println("Sending a Telegram message!");
    }

     public void getMessage() {
         System.out.println("Receiving a Telegram message!");
     }
}
```

### WhatsApp:

```
public class WhatsApp implements Messenger {

    public void sendMessage() {

        System.out.println("Sending a WhatsApp message!");
    }

     public void getMessage() {
         System.out.println("Reading a WhatsApp message!");
     }
}
```

### Viber:

```
public class Viber implements Messenger {

    public void sendMessage() {

        System.out.println("Sending a Viber message!");
    }

     public void getMessage() {
         System.out.println("Receiving a Viber message!");
     }
}
```

**What advantages does this provide?** The most important of them is **loose coupling**. Imagine that we're designing a program that will collect client data. The Client class definitely needs a

field to indicate which specific messenger the client is using. Without interfaces, this would look weird:

```
public class Client {

    private WhatsApp whatsApp;
    private Telegram telegram;
    private Viber viber;
}
```

**But why do we need interfaces for this?** That's a good question — and the right question! **Can't we achieve the same result using ordinary inheritance**?

The Messenger class as the parent, and Viber, Telegram, and WhatsApp as the children. Indeed, that is possible.

But there's one snag. As you already know, **Java has no multiple inheritance**. But there is support for multiple interfaces. **A class can implement as many interfaces as you want**.

Now the Telegram class can easily **implement both interfaces**! Accordingly.

```
public class Telegram implements Application, Messenger {

    // ...methods
}
```

# Default methods

An interesting addition appeared in Java 8 — **default methods**.

For example, your interface has 10 methods. 9 of them have different implementations in different classes, but one is implemented the same for all. Previously, before Java 8, interface methods had no implementation whatsoever: the compiler immediately gave an error.

Now you can do something like this:

```
public interface CanSwim {
```

```
    public default void swim() {
        System.out.println("Swim!");
    }

    public void eat();

    public void run();
}
```

Using the **default** keyword, we've created an interface method with a default implementation. We need to provide our own implementation for two other methods — eat() and run() — in all classes that implement CanSwim. We don't need to do this with the swim() method: the implementation will be the same in every class.

# Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

Example

```
// Filename: Sports.java
public interface Sports {
  public void setHomeTeam(String name);
  public void setVisitingTeam(String name);
}
// Filename: Football.java
public interface Football extends Sports {
  public void homeTeamScored(int points);
```

```
  public void visitingTeamScored(int points);
  public void endOfQuarter(int quarter);
}
public interface Hockey extends Sports {
  public void homeGoalScored();
  public void visitingGoalScored();
  public void endOfPeriod(int period);
  public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but

- it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods.
- Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

# Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface. The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as –

Example

```
public interface Hockey extends Sports, Event
```

## Understanding concept of interfaces

1- **Simplest Answer:** Suppose your parents gives you a list of items to purchase, that list is an Interface that you will implement at time of purchasing. Means before implementing anything, you list out what you have to done that will be interface.

2- **Real World:** You have an Animal class. It is an abstract class, because you cannot instantiate a generic "Animal," but it provides base functionality.
You have many derived classes of Animal. You have HomoSapiens, Platypus, Penguin (which extends Bird, another subclass of Animal), Giraffe, Housefly, etc. Each of these are concrete classes that may be instantiated (of course, there are several levels of abstract classes between these and Animal (like Chordata, etc.)
Now you want to make something fly. What can fly? Birds and Houseflys (among others), so these classes should provide similar functionality, even though they are widely spaced on our inheritance tree.
**Solution:** Make them use interfaces. Bird and Housefly cannot both implement the Flyer interface, so whenever we want something to fly, we can use a Flyer object, not caring whether it's a Bird or a Housefly. Likewise, Penguins and Playtpuses can implement the Swimmer interface

3- **Business examples**: I have a persistance engine that will work against any data sourcer (XML, ASCII (delimited and fixed-length), various JDBC sources (Oracle, SQL, ODBC, etc.) I created a base, abstract class to provide common functionality in this persistance, but instantiate the appropriate "Port" (subclass) when persisting my objects. (This makes development of new "Ports" much easier, since most of the work is done in the superclasses; especially the various JDBC ones; since I not only do persistance but other things [like table generation], I have to provide the various differences for each database.)
The best business examples of Interfaces are the Collections. I can work with a java.util.List without caring how it is implemented; having the List as an abstract class does not make sense because there are fundamental differences in how an ArrayList works as opposed to a LinkedList. Likewise, Map and Set. And if I am just working with a group of objects and don't care if it's a List, Map, or Set, I can just use the Collection interface.
Hope that this helps