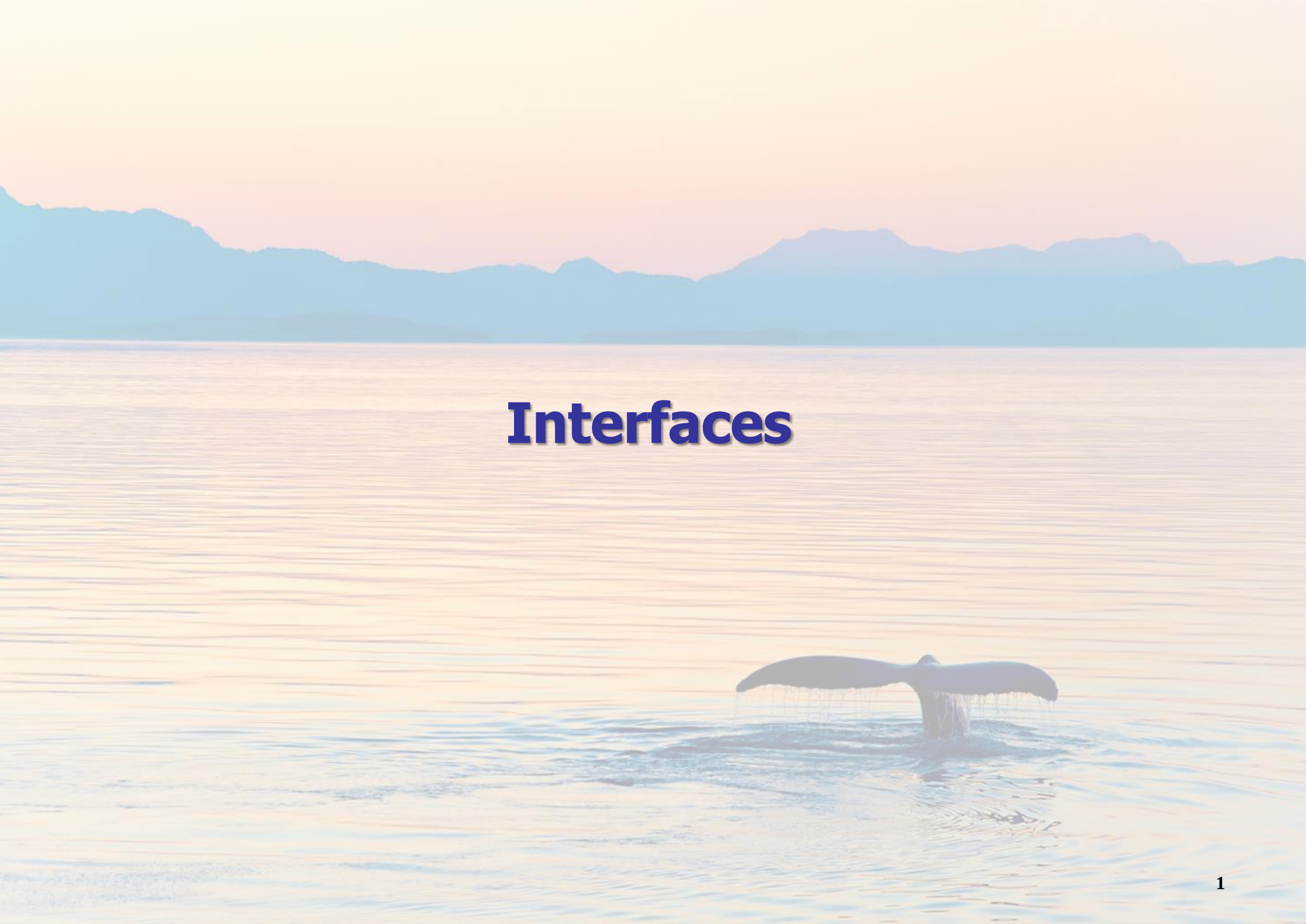
Interfaces



Contents

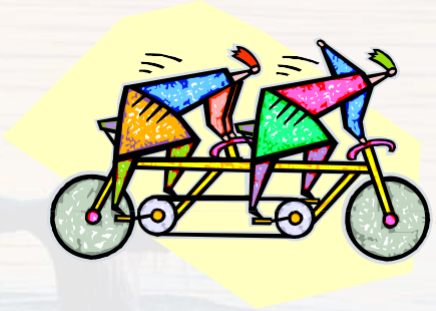
- What is an Interface?
- Interfaces in Java
- Interfaces and Multiple Inheritance
- A Sample Interface, Relatable
- Using an Interface as a Type
- Rewriting Interfaces
- Abstract Classes vs. Interfaces
- Summary of Interfaces

A Hierarchy of Bicycle Classes

Bicycle

RoadBike

MountainBike



TandemBike

Possible Implementation of a Bicycle

```
class Bicycle {
    private int cadence, speed, gear;           // three fields  represent the object state
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;                        // the Bicycle class has one constructor
        cadence = startCadence;
        speed = startSpeed;
    }
    public void changeCadence(int newValue) {       // methods define interactions of
        cadence = newValue;                           // the object with the outside world
    }
    public void changeGear(int newValue){
        gear = newValue;
    }
    public void speedUp(int increment){
        speed = speed + increment;
    }
    public void applyBrakes(int decrement){
        speed = speed - decrement;
    }
    public void printStates() {
        System.out.println("cadence:"+cadence+" speed:"+speed+" gear:"+gear);
    }
}
```

Possible Implementation of a MountainBike

```
public class MountainBike extends Bicycle {
        // the MountainBike subclass adds one exra field
    private int seatHeight;
        // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }
        // the MountainBike subclass has one extra method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

- All fields (cadence, speed and gear) are inherited from the Bicycle class
- All methods are inherited from the Bicycle class

What is an Interface?

- In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

```
interface BicycleInterface {
        void changeCadence(int newValue);
        void changeGear(int newValue);
        void speedUp(int increment);
        void applyBrakes(int decrement);
}
```

What is an Interface?

- To implement this interface, the name of your class would change (to ACMEBicycle, for example), and you'd use the implements keyword in the class declaration (see the next slide).
- Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler.
- If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

What is an Interface?

```
class ACMEBicycle implements BicycleInterface {
      private int cadence, speed, gear;     // three fields  represent the object state
      public ACMEBicycle(int startCadence, int startSpeed, int startGear) {
          gear = startGear;                        // the ACMEBicycle class has one constructor
          cadence = startCadence;
          speed = startSpeed;
      }
      void changeCadence(int newValue) {     // methods define interactions of
           cadence = newValue;                    // the object with the outside world
       }
       void changeGear(int newValue) {
           gear = newValue;
       }
       void speedUp(int increment) {
           speed = speed + increment;
       }
       void applyBrakes(int decrement) {
           speed = speed - decrement;
       }
       void printStates() {
            System.out.println("cadence:"+cadence+" speed:"+speed+" gear:"+gear);
       }
}
```

Interfaces in Java

- In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only*
  - constants,
  - method signatures, and
  - nested types.
- There are no method bodies.
- Interfaces cannot be instantiated——they can only be *implemented* by classes or *extended* by other interfaces.

Interfaces in Java

- Imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator.
  - Automobile manufacturers write Java software that operates the automobile—stop, start, accelerate, turn left, and so forth.
  - Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning Satellite) position data and wireless transmission of traffic conditions and use that information to drive the car.
  - An example of the definition of this interface is on the next slide.

Interfaces in Java

- Defining an interface is similar to creating a new class:

```
public interface OperateCar {

  // constant declarations, if any

  // method signatures
  int turn(Direction direction,       // An enum with values RIGHT,
           LEFT double radius, double startSpeed, double endSpeed);
  int changeLanes(Direction direction, double startSpeed, double endSpeed);
  int signalTurn(Direction direction, boolean signalOn);
  int getRadarFront(double distanceToCar, double speedOfCar);
  int getRadarRear(double distanceToCar, double speedOfCar);
      ......
  // more method signatures
}
```

Interfaces in Java

- To use an interface, you write a class that *implements* the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface.

```
public class OperateBMW760i implements OperateCar {
// the OperateCar method signatures, with implementation, for example:

int signalTurn(Direction direction, boolean signalOn) {
   //code to turn BMW's LEFT turn indicator lights on
   //code to turn BMW's LEFT turn indicator lights off
   //code to turn BMW's RIGHT turn indicator lights on
   //code to turn BMW's RIGHT turn indicator lights off
}

// other members, as needed -- for example, helper classes
// not visible to clients of the interface
}
```

Interfaces in Java

- In the robotic car example above, it is the automobile manufacturers who will implement the interface.
  - BMW's implementation will be substantially different from that of Toyota, of course, but both manufacturers will adhere to the same interface.
- The guidance manufacturers, who are the clients of the interface, will build systems that use GPS data on a car's location, digital street maps, and traffic data to drive the car. In so doing, the guidance systems will invoke the interface methods:
  - turn, change lanes, brake, accelerate, and so forth.

Interfaces and Multiple Inheritance

- Interfaces are not part of the class hierarchy, although they work in combination with classes.
- Java does not permit multiple inheritance but interfaces provide an alternative.
- In Java, a class can inherit from only one class but it can implement more than one interface.
- Therefore, objects can have multiple types:
  - The type of their own class and the types of all the interfaces that they implement.
  - InterfaceA interfaceName = new ClassB();
    or new ClassC();
    or new ClassD();
  - InterfaceY interfaceNameY = new ClassX();
    or InterfaceZ interfaceNameZ
    or InterfaceK interfaceNameK

An Example with Multiple Inheritance

- An interface declaration consists of modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body.

```
public interface GroupedInterface extends Interface1,
                                          Interface2, Interface3 {
   // constant declarations
   double E = 2.718282;   // base of natural logarithms

   // method signatures
   void doSomething (int i, double x);
   int doSomethingElse(String s);
}
```

A Sample Interface, Relatable

- An interface that defines how to compare the size of objects.
- If you want to be able to compare the size of similar objects, no matter what they are, the class that instantiates them should implement Relatable.

```
public interface Relatable {

   // this (object calling isLargerThan) and
   // other must be instances of the same class
   // returns 1, 0, -1 if this is greater
   // than, equal to, or less than other
   public int isLargerThan(Relatable other);

}
```

A Sample Interface, Relatable

- Any class can implement Relatable if there is some way to compare the relative "size" of objects instantiated from the class.
  - For strings, it could be a number of characters;
  - For books, it could be a number of pages;
  - For students, it could be weight; and so forth;
  - For planar geometric objects, it could be area;
  - For three-dimensional geometric objects, it could be volume.
  - All such classes can implement the isLargerThan() method (see the RectanglePlus class that follows).

Implementing the Relatable Interface

```
public class RectanglePlus implements Relatable {
    private int width = 0;
    private int height = 0;
    private Point origin;
    public RectanglePlus() {    // constructor
            origin = new Point(0, 0);
     }
     public int getArea() {// a method for computing the area of the rectangle
            return width * height;
    }
    public int isLargerThan(Relatable other) {    // a method to implement Relatable
            RectanglePlus otherRect = (RectanglePlus)other;
            if (this.getArea() < otherRect.getArea()) return -1;
            else if (this.getArea() > otherRect.getArea())  return 1;
            else return 0;
     }
}
```

Using an Interface as a Type

- When you define a new interface, you are defining a new reference data type.
  - You can use interface names anywhere you can use any other data type name.
- If you define a reference variable whose type is an interface, any object you assign to it *must* be an instance of a class that implements the interface.
  - InterfaceA interfaceName = new ClassB();

Using an Interface as a Type

- Here is a method for finding the largest object in a pair of objects, for *any* objects that are instantiated from a class that implements Relatable.

```
public Object findLargest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
        //   By casting object1 to a Relatable type, it can invoke the
        //   isLargerThan method
   if ( (obj1).isLargerThan(obj2)) > 0)
      return object1;
   else
      return object2;
}
```

- This method works for any "relatable" objects. When they implement Relatable, they can be of both their own class (or superclass) type and a Relatable type.

Rewriting Interfaces

- Consider an interface that you have developed called DoIt:

```
public interface DoIt {
   void doSomething(int i, double x); int
   doSomethingElse(String s);
}
```

- Suppose that, at a later time, you want to add a third method to DoIt, so that the interface now becomes:

```
public interface DoIt {
   void doSomething(int i, double x); int
   doSomethingElse(String s);
   boolean didItWork(int i, double x, String s);

}
```

Comments on the Previous Slide

- If you make this change, all classes that implement the old DoIt interface will break because they don't implement the interface anymore.
- Programmers relying on this interface will protest loudly. Try to anticipate all uses for your interface and to specify it completely from the beginning.
- You may need to create more interfaces later. For example, you could create a DoItPlus interface that extends DoIt, see the next slide.

Rewriting Interfaces

- Now users of your code can choose to continue to use the old interface or to upgrade to the new interface

```
public interface DoItPlus extends DoIt {

    boolean didItWork(int i, double x, String s);

}
```

Abstract Classes vs. Interfaces

- Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation.
- If an abstract class contains *only* abstract method declarations, it should be declared as an interface instead.

Abstract Classes vs. Interfaces

- A class that implements an interface must implement *all* of the interface's methods. It is possible, however, to define a class that does not implement all of the interface methods, provided that the class is declared to be abstract.

```
abstract class X implements Y {
    // implements all but one method of Y
}

class Z extends X {
    // implements the remaining method in Y
}
```

- In this case, class X must be abstract because it does not fully implement Y, but class Z does, in fact, implement Y.

Summary of Interfaces

- An interface defines a protocol of communication between two objects. An interface declaration contains signatures, but no implementations, for a set of methods, and might also contain constant definitions.
- A class that implements an interface must implement all the methods declared in the interface.
- An interface name can be used anywhere a type can be used.