Interfaces, Subclasses, Polymorphisms

Mitsu Ogihara

Department of Computer Science
University of Miami

Table of Contents

Outline

Interface

- An **interface** is a template for a Java class
- An interface may define static variables and methods
- An interface should not define instance variables and could not be instantiated
- An interface may declare methods but only their headers – the body should be substituted with a single ';'
  They are called **abstract methods**

A Format of an Interface

Let us say we are defining an interface by the name of FooInt.
The file name should be `FooInt.java` and the file takes the following format:

```
1  public interface FooInt {
2    // constants
3    public static TYPE NAME = DATA;
4    ...
5    public static TYPE NAME = DATA;
6    // static methods
7    public static TYPE NAME( PARAMTERS ) {
8      ...
9    }
10   ...
11   public static TYPE NAME( PARAMTERS ) {
12     ...
13   }
14   // abstract methods
15   public TYPE NAME( PARAMTERS );
16   ...
17   public TYPE NAME( PARAMTERS );
18 }
```

The interface declaration

A Format of an Interface

Let us say we are defining an interface by the name of FooInt.
The file name should be `FooInt.java` and the file takes the following format:

```
1  public interface FooInt {
2    // constants
3    public static TYPE NAME = DATA;
4    ...
5    public static TYPE NAME = DATA;
6    // static methods
7    public static TYPE NAME( PARAMTERS ) {
8      ...
9    }
10   ...
11   public static TYPE NAME( PARAMTERS ) {
12     ...
13   }
14   // abstract methods
15   public TYPE NAME( PARAMTERS );
16   ...
17   public TYPE NAME( PARAMTERS );
18 }
```

Constants

A Format of an Interface

Let us say we are defining an interface by the name of FooInt.
The file name should be `FooInt.java` and the file takes the following format:

```
1  public interface FooInt {
2    // constants
3    public static TYPE NAME = DATA;
4    ...
5    public static TYPE NAME = DATA;
6    // static methods
7    public static TYPE NAME( PARAMTERS ) {
8      ...
9    }
10   ...
11   public static TYPE NAME( PARAMTERS ) {
12     ...
13   }
14   // abstract methods
15   public TYPE NAME( PARAMTERS );
16   ...
17   public TYPE NAME( PARAMTERS );
18 }
```

Static methods

A Format of an Interface

Let us say we are defining an interface by the name of FooInt.
The file name should be `FooInt.java` and the file takes the following format:

```
1  public interface FooInt {
2    // constants
3    public static TYPE NAME = DATA;
4    ...
5    public static TYPE NAME = DATA;
6    // static methods
7    public static TYPE NAME( PARAMTERS ) {
8      ...
9    }
10   ...
11   public static TYPE NAME( PARAMTERS ) {
12     ...
13   }
14   // abstract methods
15   public TYPE NAME( PARAMTERS );
16   ...
17   public TYPE NAME( PARAMTERS );
18 }
```

Abstract instance methods

How to Work with an Interface

- A class can be written to realize the ideas in an interface – such a class is said to **implement** the interface

How to Work with an Interface

- A class can be written to realize the ideas in an interface – such a class is said to **implement** the interface
- In a class that **implements an interface**, the class must have to define all the abstract methods declared in the interface

How to Work with an Interface

- A class can be written to realize the ideas in an interface – such a class is said to **implement** the interface
- In a class that **implements an interface**, the class must have to define all the abstract methods declared in the interface
- In a class that **implements an interface**, the static variables and methods are available without attaching the name of interface and a period as the prefix
  This concept is called **inheritance**

Outline

frametitle

The problem of recording information about a person: the name as a String and the active/inactive status as a boolean

We want to be able to

1. obtaining the name of the person,
2. examining whether the person is active,
3. activating the person if the person is not active, and
4. deactivating the person if the person is active.

frametitle

The problem of recording information about a person: the name as a String and the active/inactive status as a boolean

We want to be able to

1. obtaining the name of the person,
2. examining whether the person is active,
3. activating the person if the person is not active, and
4. deactivating the person if the person is active.

We can think of the class implementing an interface `PersonInt`

PersonInt.java

```
1  public interface PersonInt {
2    ////////////////// CONSTANTS ////////////////////////////
3    public static final String ACTIVE_STRING = "active";
4    public static final String INACTIVE_STRING = "inactive";
5    ////////////////// ACCESSORS ////////////////////////////
6    /**
7     * @return name
8     */
9    public String getName();
10   /**
11    * @return whether the person is active
12    */
13   public boolean isActive();
14   ////////////////// MUTATORS ////////////////////////////
15   /**
16    * deactivate
17    */
18   public void deactivate();
19   /**
20    * activate
21    */
22   public void activate();
23 }
```

The interface declaration

PersonInt.java

```
1  public interface PersonInt {
2    ////////////////////// CONSTANTS ////////////////////////////////
3    public static final String ACTIVE_STRING = "active";
4    public static final String INACTIVE_STRING = "inactive";
5    ////////////////////// ACCESSORS ////////////////////////////////
6    /**
7     * @return name
8     */
9    public String getName();
10   /**
11    * @return whether the person is active
12    */
13   public boolean isActive();
14   ////////////////////// MUTATORS /////////////////////////////////
15   /**
16    * deactivate
17    */
18   public void deactivate();
19   /**
20    * activate
21    */
22   public void activate();
23 }
```

Constants; these are available to any classes implementing the interface
They have the attribute of `final` and so they cannot be modified

PersonInt.java

```
1  public interface PersonInt {
2    ////////////////// CONSTANTS ////////////////////////////
3    public static final String ACTIVE_STRING = "active";
4    public static final String INACTIVE_STRING = "inactive";
5    ////////////////// ACCESSORS ////////////////////////////
6    /**
7     * @return name
8     */
9    public String getName();
10   /**
11    * @return whether the person is active
12    */
13   public boolean isActive();
14   ////////////////// MUTATORS ////////////////////////////
15   /**
16    * deactivate
17    */
18   public void deactivate();
19   /**
20    * activate
21    */
22   public void activate();
23 }
```

Abstract instance methods - accessors

PersonInt.java

```
1  public interface PersonInt {
2    ////////////////////// CONSTANTS ////////////////////////////////
3    public static final String ACTIVE_STRING = "active";
4    public static final String INACTIVE_STRING = "inactive";
5    ////////////////////// ACCESSORS ////////////////////////////////
6    /**
7     * @return name
8     */
9    public String getName();
10   /**
11    * @return whether the person is active
12    */
13   public boolean isActive();
14   ////////////////////// MUTATORS /////////////////////////////////
15   /**
16    * deactivate
17    */
18   public void deactivate();
19   /**
20    * activate
21    */
22   public void activate();
23 }
```

javadoc

Implementation Number 1: Class PersonReg

- This implementation uses a boolean to record the active/inactive status
- Two constructors are designed:
  - One that takes one `String` as the paramter:
    the String is the name and the default status is assigned
  - One that takes two `String` objects as the paramters: the first String is the name and the second String is the status, which is supposed to be either `ACTIVE_STRING` or `INACTIVE_STRING` as defined in the interface `PersonInt`

Implementation Number: Code

```
1  public class PersonReg implements PersonInt {
2    ////////////////////// CONSTANTS ////////////////////////////////
3    static final boolean DEFAULT_STATUS = false;
4    ////////////////////// INSTANCE VARIABLES ///////////////////////
5    private String name;
6    private boolean status;
7    ////////////////////// CONSTRUCTORS /////////////////////////////
8    /** constructor
9     * @param name the name
10    */
11   public PersonReg( String name ) {
12     this.name = name;
13     status = DEFAULT_STATUS;
14   }
15   /** constructor
16    * @param name  the name
17    * @param status  the status String
18    */
19   public PersonReg( String name, String statusString ) {
20     this.name = name;
21     status = statusString.equals( ACTIVE_STRING );
22   }
```

The header; states **"implements PersonInt"**

Implementation Number: Code

```
1  public class PersonReg implements PersonInt {
2    ////////////////////// CONSTANTS ////////////////////////////////
3    static final boolean DEFAULT_STATUS = false;
4    ////////////////////// INSTANCE VARIABLES ///////////////////////
5    private String name;
6    private boolean status;
7    ////////////////////// CONSTRUCTORS /////////////////////////////
8    /** constructor
9     * @param name the name
10    */
11   public PersonReg( String name ) {
12     this.name = name;
13     status = DEFAULT_STATUS;
14   }
15   /** constructor
16    * @param name  the name
17    * @param status  the status String
18    */
19   public PersonReg( String name, String statusString ) {
20     this.name = name;
21     status = statusString.equals( ACTIVE_STRING );
22   }
```

Constant used as private

Implementation Number: Code

```
1  public class PersonReg implements PersonInt {
2    ////////////////////// CONSTANTS ////////////////////////////////
3    static final boolean DEFAULT_STATUS = false;
4    ////////////////////// INSTANCE VARIABLES ///////////////////////
5    private String name;
6    private boolean status;
7    ////////////////////// CONSTRUCTORS /////////////////////////////
8    /** constructor
9     * @param name the name
10    */
11   public PersonReg( String name ) {
12     this.name = name;
13     status = DEFAULT_STATUS;
14   }
15   /** constructor
16    * @param name  the name
17    * @param status  the status String
18    */
19   public PersonReg( String name, String statusString ) {
20     this.name = name;
21     status = statusString.equals( ACTIVE_STRING );
22   }
```

Instance variables; they are private

Implementation Number: Code

```
1  public class PersonReg implements PersonInt {
2    ////////////////////// CONSTANTS ////////////////////////////////
3    static final boolean DEFAULT_STATUS = false;
4    ////////////////////// INSTANCE VARIABLES ///////////////////////
5    private String name;
6    private boolean status;
7    ////////////////////// CONSTRUCTORS /////////////////////////////
8    /** constructor
9     * @param name the name
10    */
11   public PersonReg( String name ) {
12     this.name = name;
13     status = DEFAULT_STATUS;
14   }
15   /** constructor
16    * @param name  the name
17    * @param status  the status String
18    */
19   public PersonReg( String name, String statusString ) {
20     this.name = name;
21     status = statusString.equals( ACTIVE_STRING );
22   }
```

Constructor 1: the parameter is the name

Implementation Number: Code

```
1  public class PersonReg implements PersonInt {
2    ////////////////////// CONSTANTS ////////////////////////////////
3    static final boolean DEFAULT_STATUS = false;
4    ////////////////////// INSTANCE VARIABLES ///////////////////////
5    private String name;
6    private boolean status;
7    ////////////////////// CONSTRUCTORS /////////////////////////////
8    /** constructor
9     * @param name the name
10    */
11   public PersonReg( String name ) {
12     this.name = name;
13     status = DEFAULT_STATUS;
14   }
15   /** constructor
16    * @param name  the name
17    * @param status  the status String
18    */
19   public PersonReg( String name, String statusString ) {
20     this.name = name;
21     status = statusString.equals( ACTIVE_STRING );
22   }
```

Constructor 2: the second `String` is compared with `ACTIVE_STRING` in `PersonInt` to determine the status stored

Implementation Number: Methods

```
23    ////////////////// ACCESSORS //////////////////////////////
24    public String getName() {
25      return name;
26    }
27    public boolean isActive() {
28      return status;
29    }
30    ////////////////// MUTATORS ///////////////////////////////
31    public void deactivate() {
32      status = false;
33    }
34    public void activate() {
35      status = true;
36    }
37  }
```

The body for `getName()`

Implementation Number: Methods

```
23    ////////////////// ACCESSORS ///////////////////////////////
24    public String getName() {
25      return name;
26    }
27    public boolean isActive() {
28      return status;
29    }
30    ////////////////// MUTATORS ////////////////////////////////
31    public void deactivate() {
32      status = false;
33    }
34    public void activate() {
35      status = true;
36    }
37  }
```

The body for `isActive()`

Implementation Number: Methods

```
23    ////////////////// ACCESSORS ///////////////////////////////
24    public String getName() {
25      return name;
26    }
27    public boolean isActive() {
28      return status;
29    }
30    ////////////////// MUTATORS ////////////////////////////////
31    public void deactivate() {
32      status = false;
33    }
34    public void activate() {
35      status = true;
36    }
37  }
```

The body for `deactivate()`

Implementation Number: Methods

```
23    ////////////////// ACCESSORS //////////////////////////////
24    public String getName() {
25      return name;
26    }
27    public boolean isActive() {
28      return status;
29    }
30    ////////////////// MUTATORS ///////////////////////////////
31    public void deactivate() {
32      status = false;
33    }
34    public void activate() {
35      status = true;
36    }
37  }
```

The body for `aactivate()`

Table of Contents

Extending `Person`

- Suppose we want to write an application in which we maintain a record of students with their name, active/inactive status, classification (freshman, sophomore, etc.), and GPA.
- In `PersonInt` and `PersonReg` we designed an interface and two implementations of the interface.
- Can we recycle the code we developed?

Extending `Person`

- Suppose we want to write an application in which we maintain a record of students with their name, active/inactive status, classification (freshman, sophomore, etc.), and GPA.
- In `PersonInt` and `PersonReg` we designed an interface and two implementations of the interface.
- Can we recycle the code we developed?
- It is possible to design a new class by extending either `PersonReg`

Subclass and Superclass

- If a class `Student` *extends* a class `PersonReg`, we say that **Student is a subclass of PersonReg** and **PersonReg is a superclass of Student**
- We define this formally by declaration
  `class Student extends PersonReg`

Rules about Subclasses

Suppose class `Student` extends class `PersonReg`, that is, class `Student` is a subclass of class `PersonReg`

Rules about Subclasses

Suppose class `Student` extends class `PersonReg`, that is, class `Student` is a subclass of class `PersonReg`

- Class `Student` inherits from class `PersonReg`
  - All non-private instance variables
  - All non-private class variables
  - All non-private static methods
  - All non-private instance methods

Rules about Subclasses

Suppose class `Student` extends class `PersonReg`, that is, class `Student` is a subclass of class `PersonReg`

- Class `Student` inherits from class `PersonReg`
  - All non-private instance variables
  - All non-private class variables
  - All non-private static methods
  - All non-private instance methods
- Class `Student` must have constructors

Rules about Subclasses

Suppose class `Student` extends class `PersonReg`, that is, class `Student` is a subclass of class `PersonReg`

- Class `Student` inherits from class `PersonReg`
  - All non-private instance variables
  - All non-private class variables
  - All non-private static methods
  - All non-private instance methods
- Class `Student` must have constructors
- Class `Student` may redefine the methods inherited from class `PersonReg` (called **overriding**)
- Class `Student` may assign new values to the class variables inherited from class `PersonReg` (unless they have the attribute of **final**, which means only once a value can be assigned)

Class Student

```
1  public class Student extends PersonReg {
2    private int classification;
3    private double gpa;
4    //// constructor
5    public Student( String name, String statusString,
6        int classification, double gpa ) {
7      super( name, statusString );
8      setClassification( classification );
9      setGpa( gpa );
10   }
11   public Student( String name, int classification, double gap ) {
12     super( name );
13     setClassification( classification );
14     setGpa( gpa );
15   }
```

The class declaration

Class `Student`

```
1  public class Student extends PersonReg {
2    private int classification;
3    private double gpa;
4    //// constructor
5    public Student( String name, String statusString,
6        int classification, double gpa ) {
7      super( name, statusString );
8      setClassification( classification );
9      setGpa( gpa );
10   }
11   public Student( String name, int classification, double gap ) {
12     super( name );
13     setClassification( classification );
14     setGpa( gpa );
15   }
```

Private instance variable defined in `Student`

Class `Student`

```
1  public class Student extends PersonReg {
2    private int classification;
3    private double gpa;
4    //// constructor
5    public Student( String name, String statusString,
6        int classification, double gpa ) {
7      super( name, statusString );
8      setClassification( classification );
9      setGpa( gpa );
10   }
11   public Student( String name, int classification, double gap ) {
12     super( name );
13     setClassification( classification );
14     setGpa( gpa );
15   }
```

Constructor with full information:
`name`, `statusString`, `class` (0 - 3), and `gpa`
`super(name, statusString)` executes the constructor for the superclass `PersonReg`
the constructor then calls the modifiers for classification and gpa

Class Student

```
1  public class Student extends PersonReg {
2    private int classification;
3    private double gpa;
4    //// constructor
5    public Student( String name, String statusString,
6        int classification, double gpa ) {
7      super( name, statusString );
8      setClassification( classification );
9      setGpa( gpa );
10   }
11   public Student( String name, int classification, double gap ) {
12     super( name );
13     setClassification( classification );
14     setGpa( gpa );
15   }
```

The one-parameter constructor of the superclass

Class `Student`: Accessors

```
16    ///// accessors
17    public int getClassification() {
18      return classification;
19    }
20    public double getGpa() {
21      return gpa;
22    }
23    ///// modifiers
24    public void setClassification( int c ) {
25      classification = c;
26    }
27    public void setGpa( double gpa ) {
28      this.gpa = gpa;
29    }
30  }
```

Get classification

Class `Student`: Accessors

```
16    ///// accessors
17    public int getClassification() {
18      return classification;
19    }
20    public double getGpa() {
21      return gpa;
22    }
23    ///// modifiers
24    public void setClassification( int c ) {
25      classification = c;
26    }
27    public void setGpa( double gpa ) {
28      this.gpa = gpa;
29    }
30  }
```

Get gpa

Class `Student`: Accessors

```
16    ///// accessors
17    public int getClassification() {
18      return classification;
19    }
20    public double getGpa() {
21      return gpa;
22    }
23    ///// modifiers
24    public void setClassification( int c ) {
25      classification = c;
26    }
27    public void setGpa( double gpa ) {
28      this.gpa = gpa;
29    }
30  }
```

Set classification

Class `Student`: Accessors

```
16    ///// accessors
17    public int getClassification() {
18      return classification;
19    }
20    public double getGpa() {
21      return gpa;
22    }
23    ///// modifiers
24    public void setClassification( int c ) {
25      classification = c;
26    }
27    public void setGpa( double gpa ) {
28      this.gpa = gpa;
29    }
30  }
```

set gpa

Polymorphism

- An object in a subclass can be treated as an object in superclass
- This is done by attaching the name of the superclass name in front the object name
  `(SUPER-CLASS-NAME)object-name`
- When executing a method from the superclass, we need an additional pair of parentheses
  `((SUPER-CLASS-NAME)object-name).METHOD-NAME(...)`
- The concept that a single object can be treated as an object from multiple classes is called **polymorphism**

Polymorphism

- An object in a subclass can be treated as an object in superclass
- This is done by attaching the name of the superclass name in front the object name
  `(SUPER-CLASS-NAME)object-name`
- When executing a method from the superclass, we need an additional pair of parentheses
  `((SUPER-CLASS-NAME)object-name).METHOD-NAME(...)`
- The concept that a single object can be treated as an object from multiple classes is called **polymorphism** For example, every object class in Java is a subclass of class `Object`

Polymorphism

- An object in a subclass can be treated as an object in superclass
- This is done by attaching the name of the superclass name in front the object name
  `(SUPER-CLASS-NAME)object-name`
- When executing a method from the superclass, we need an additional pair of parentheses
  `((SUPER-CLASS-NAME)object-name).METHOD-NAME(...)`
- The concept that a single object can be treated as an object from multiple classes is called **polymorphism** For example, every object class in Java is a subclass of class `Object`
- You can use
  `OBJECT-NAME instanceof CLASS-NAME`
  to test whether `OBJECT-NAME` can be treated as a `CLASS-NAME` object

Class Object

- Every object class in Java is a subclass of class `Object`!
- For each primitive data type, its object class version exists they are:
  - `Character`,
  - `Boolean`,
  - `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`

The End