JC2002 Java Programming

Lecture 12: Interfaces

Interface

- With ***interfaces***, unrelated classes can implement a set of common methods: people and systems can interact one with another in a standardized way via the interfaces
- Example: The controls on a radio serve as an interface between the user of radio the internal components of the radio
  - Offers a limited set of operations (e.g., change the station, adjust the volume, choose between AM and FM)
  - Different radios may implement the controls in different ways (e.g., using push buttons, dials, voice commands)
  - The interface specifies ***what*** operations a radio must permit users to control, but does not specify ***how*** the operations are performed

Interfaces in Java

- A Java interface describes a set of methods that can be called on an object
- An *interface declaration* begins with the keyword **`interface`** and typically contains only constants and `abstract` methods
  - All interface members *must* be `public`
  - Mandatory methods declared in an interface are implicitly `public abstract` methods
  - All fields are implicitly `public`, `static` and `final`
- An interface cannot be instantiated, so it does not define a constructor

Using interface in a class

- To use an interface, a concrete class must specify that it `implements` the interface and must declare each method in the interface with specified signature
- A class that does not implement all the methods of the interface is an abstract class and must be declared `abstract`.
  - Implementing an interface is like signing a contract with the compiler: *"I will declare all the methods specified by the interface or I will declare my class abstract"*

Example of using interface

```java
abstract class Animal {
    protected boolean hungry = true;
}
interface Feedable {
    public void feed();
}
class Cat extends Animal implements Feedable {
    public void feed() {
        hungry = false;
    }
}
public class InterfaceExample1 {
    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.feed();
        System.out.print("Is the cat hungry? ");
        System.out.println(cat.hungry ? "Yes" : "No");
    }
}
```

```
$ java InterfaceExample1
Is the cast hungry? No
```

New features of interfaces in Java

- From Java SE 8, interfaces also may contain `public` default methods with concrete default implementations that specify how operations are performed if not overridden
  - If a class implements such an interface, the class also receives the interface's `default` implementations (if any)
  - To declare a default method, place the keyword `default` before the method's return type and provide a concrete method implementation
- From JAVA SE 8 interfaces may contain static methods
- From JAVA SE 9 interfaces may also contain private methods, however, defining a protected method causes compilation error

Example of interface with default method

```java
abstract class Animal {
    protected boolean hungry = true;
}
interface Feedable {
    public default void feed() {
        System.out.println("No method for feeding!");
    }
}
class Cat extends Animal implements Feedable {
}
public class InterfaceExample2 {
    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.feed();
        System.out.print("Is the cat hungry? ");
        System.out.println(cat.hungry ? "Yes" : "No");
    }
}
```

```
$ java InterfaceExample2
No method for feeding!
Is the cat hungry? Yes
```

Using multiple interfaces

- Java does not allow subclasses to inherit from more than one superclass (multiple heritance); however, a class can inherit from one superclass, *and* implement as many interfaces as it needs
- To implement more than one, use a comma-separated list of interface names after keyword `implements` in the class declaration, as in:

```
public class Subclass extends Superclass implements
    FirstInterface, SecondInterface {
```

- The Java API contains a lot of interfaces, and many of the Java API methods take interface arguments and return interface values

When to use an interface

- An interface is often used when disparate classes (i.e., unrelated classes) need to share common methods and constants
    - Allows objects of unrelated classes to be processed *polymorphically* by responding to the *same* method calls
    - You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality
- An interface should be used in place of an `abstract` class when there is no default implementation to inherit
- Like `public abstract` classes, interfaces are typically `public`
    - A `public` interface must be declared in a file with the same name as the interface and the `.java` filename extension

Same method in multiple interfaces

- If a class implements two interfaces, both defining a default method with the same name, then the class *must* override that method and provide an implementation
- It is possible to call one of the interface default methods using the following syntax:

```
InterfaceName.super.method( );
```

Example of interface with default method

```java
interface Pianist {
    default void play() { System.out.println(“Bling blong"); }
}
interface Violinist {
    default void play() { System.out.println(“Viih vooh"); }
}
class Musician implements Pianist, Violinist {
    public void play() {
        Pianist.super.play();
    }
}
public class InterfaceExampleMusician {
    public static void main(String[] args) {
        new Musician().play();
    }
}
```

```
$ java InterfaceExampleMusician
Bling blong
```

Extending interfaces

- Like classes, interfaces can be extended
  - Extended interface inherits all the methods from the superinterface
- An interface can extend more than one superinterfaces
- A class that implements such an interface must implement the abstract methods defined directly by the interface and all the abstract methods inherited from all the superinterfaces

1495 UNIVERSITY OF ABERDEEN

Example of extended interfaces

```java
interface Scalable  {  void scale(double scaler); }
interface Rotatable {  void rotate();             }
interface Transformable extends Scalable, Rotatable {}
class Rectangle implements Transformable {
    public double w, h;
    public Rectangle(double w, double h) { this.w = w; this.h = h; }
    public void scale(double scaler) { this.w *= scaler; this.h *= scaler; }
    public void rotate() {
        double temp = this.w; this.w = this.h; this.h = temp; }
}
public class InterfaceExampleTransformable {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(10.0,5.0);
        rect.scale(0.5);
        System.out.printf("New dimensions: %f,%f\n", rect.w, rect.h);
    }
}
```

```
$ java InterfaceExampleTransformable
New dimensions: 5.000000,2.500000
```

Functional interfaces

- As of Java SE 8, any interface containing only one `abstract` method is known as a *functional interface*—also called SAM (Single Abstract Method) interfaces
- Optional annotation **@FunctionalInterface** can be used
- Example functional interfaces defined in Java API:
  - **`Comparator`** (Chapter 16 in Deitel book) — implement this interface to define a method to compare two objects of given type to determine if the first object is less than, equal to or greater than the second
  - **`Runnable`** (Chapter 23 in Deitel book) — implement this interface to define a task that runs in parallel with other parts of your program

1495 UNIVERSITY OF ABERDEEN

Example of functional interface

```java
@FunctionalInterface
interface Talkable {
    void talk(String msg);
}
public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        Talkable person = new Talkable() {
            public void talk(String msg) {
                System.out.println(msg);
            }
        };
        person.talk("Hello world!");
    }
}
```

```
$ java FunctionalInterfaceExample
Hello world!
```

Lambda expressions

- *Lambda expression* is a new feature introduced in Java SE 8, allowing to represent the single method of a functional interface
- Format of lambda expression: **`(argument list) -> { body }`**
  - Argument list can be empty **`()`** or contain one or more arguments
  - Body contains the implementation of the method
- Lambda expressions are used in *functional programming*
  - We will revisit lambda expressions later in this course in more detail

Example of lambda expression (1)

```java
@FunctionalInterface
interface Talkable {
    void talk(String msg);
}
public class LambdaExample1 {
    public static void main(String[] args) {
        Talkable person = (msg) -> {System.out.println(msg);};
        person.talk("Hello world!");
    }
}
```

```
$ java LambdaExample1
Hello world!
```

Example of lambda expression (2)

```java
@FunctionalInterface
interface Talkable {
    void talk(String msg);
}
public class LambdaExample2 {
    public static void main(String[] args) {
        Talkable person = (msg) -> {System.out.println(msg);};
        person.talk("Hello world!");
        Talkable quietPerson =
            (msg) -> {System.out.println("Shh!");};
        quietPerson.talk("Hello world!");
    }
}
```

```
$ java LambdaExample2
Hello world!
Shh!
```

Summary

- Abstract classes are classes including methods without concrete implementation
  - Abstract methods used as a “placeholder” for concrete implementations in subclasses of an abstract class
  - Helps to keep definition and implementation of functionality separate
- Interfaces define a set of common functionalities, like abstract classes
  - Interface is a kind of “agreement” on what your class can do
  - Java does not support multiple inheritance, but similar effect can be achieved by implementing multiple interfaces
  - Functional interface is a type of interface that contains exactly one abstract class

Questions, comments?