Lập trình hướng đối tượng

Biên soạn: Nguyễn Thị Tuyết Hải

Email: tuyethai@ptithcm.edu.vn

Chapter 6: Interfaces, Lambda Expressions, and Inner Classes

6.1 Interfaces

6.1.1 The Interface Concept

Interface: a way of describing what classes should do, without specifying how they should do it

- contain a set of requirements for the classes that want to conform to the interface
- **public methods**
- **public static final fields**

E.g

Arrays.sort(): sort an array of objects that must belong to classes which implement the Comparable interface.

```
// any class that implements the Comparable interface is required to have a compareTo method
  public interface Comparable {
        int compareTo(Object other);
  }
```

6.1 Interfaces

6.1.1 The Interface Concept

- Two steps to make a class implement an interface:
  1. declare that your class intends to implement the given interface.
  2. supply definitions for all methods in the interface.

```
public interface Comparable {
        int compareTo(Object other);
  }

public class Employee implements Comparable<Employee>{
       public int compareTo(Employee other){
            return Double.compare(salary, other.salary);
         }
}
```

Arrays.sort()

6.1 Interfaces

6.1.2 Properties of Interfaces

- Interfaces are not classes => cannot construct interface objects

```
Comparable x; // OK
x = new Comparable(. . .); // ERROR
```

- An interface variable must refer to **an object of a class that implements the interface**:

```
Comparable x = new Employee(. . .);
```

6.1 Interfaces

6.1.2 Properties of Interfaces

- Build hierarchies of interfaces

```
public interface Moveable {
    void move(double x, double y);
}

public interface Powered extends Moveable {
    double milesPerGallon(); // public
    double SPEED_LIMIT = 95; // public static final
}
```

6.1 Interfaces

6.1.2 Properties of Interfaces

Each class can have only **one superclass**, classes can implement **multiple interfaces**

```
class Employee implements Cloneable, Comparable
```

// Cloneable: make an exact copy of your class's objects

6.1 Interfaces

6.1.3 Interfaces and Abstract Classes

- Each class can **only extend** a single class, it can **implement many interfaces**:

```
class Employee extends Person implements Cloneable, Comparable // OK
```

6.1.4 Static Methods

- Possible to add static methods to interfaces

```
public interface Path {
    public static Path get(String first, String... more) {
        return FileSystems.getDefault().getPath(first, more);
    }
... }
```

6.1 Interfaces

6.1.5 Default Methods

- supply a default implementation for any interface method

```
public interface Comparable<T> {
     default int compareTo(T other) {
          return 0;
     } // By default, all elements are the same
}
```

- a default method can call other methods

```
public interface Collection {
     int size(); // An abstract method
     default boolean isEmpty(){
          return size() == 0;
     }
...}
```

6.1 Interfaces

6.1.6 Resolving Default Method Conflicts

What happens if **the exact same method** is defined as **a default method in one interface**, in another interface, or **a method of a superclass**?

- Superclasses win.
- Interfaces clash. If at least one interface provides an implementation, the programmer must resolve the ambiguity

```
interface Person {
  default String getName() {
    return getClass().getName() + "_PPP_" + hashCode();
  }
}

interface Named {
  default String getName() {
    return getClass().getName() + "_NNN_" + hashCode();
  }
}

class Student extends A implements Person, Named {
  @Override
  public String getName() {
    // return Person.super.getName();
    return Named.super.getName();
  }
}
```

6.2 Examples of Interfaces

6.2.1 Interfaces and Callbacks

- Callback pattern: specify the action that should occur whenever a particular event happens

E.g., you may want a **particular action** to occur when **a button is clicked** or a menu item is selected

Suppose that a part of your program contains **a clock**, you can ask to be notified every second so that you can update the clock face.

When constructing a timer,

- set the time interval
- tell it **what it should do** whenever **the time interval has elapsed**: give it **an object** (containing the function that the timer should call periodically) or lambda expression

6.2 Examples of Interfaces

```
class TimePrinter implements ActionListener
{
  public void actionPerformed(ActionEvent event)
  {
    System.out.println("At the tone, the time is " +
new Date());
    Toolkit.getDefaultToolkit().beep();
  }
}
```

```
public class TimerTest
{
  public static void main(String[] args)
  {
    ActionListener listener = new TimePrinter();

    // construct a timer that calls the listener
    // once every 10 seconds
    Timer t = new Timer(10000, listener);
    t.start();

    JOptionPane.showMessageDialog(null, "Quit program?");
    System.exit(0);
  }
}
```

6.2 Examples of Interfaces

6.2.2 The Comparator Interface

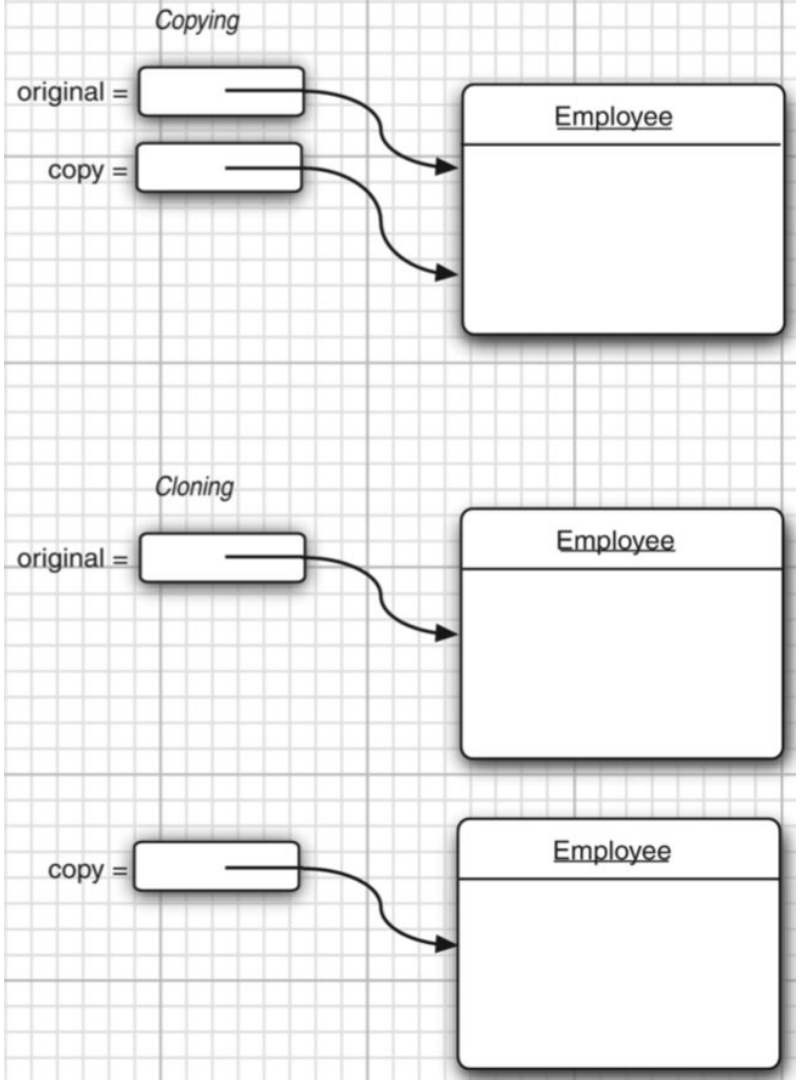
- compare strings by length

```
class LengthComparator implements Comparator<String> {
        public int compare(String first, String second) {
                return first.length() - second.length();
        }
}

public class TestMain {
        public static void main(String[] args) {
                String[] friends = { "Peter", "Paul", "Mary" };
                Arrays.sort(friends, new LengthComparator());
                for (String i: friends)
                        System.out.println(i);
                System.out.println(friends);
        }
}
```

6.2 Examples of Interfaces

6.2.3 Object Cloning

```
Employee original = new Employee("John Public",
50000);
Employee copy = original;
copy.raiseSalary(10);

Employee copy = original.clone();
copy.raiseSalary(10); // OK--original unchanged
```

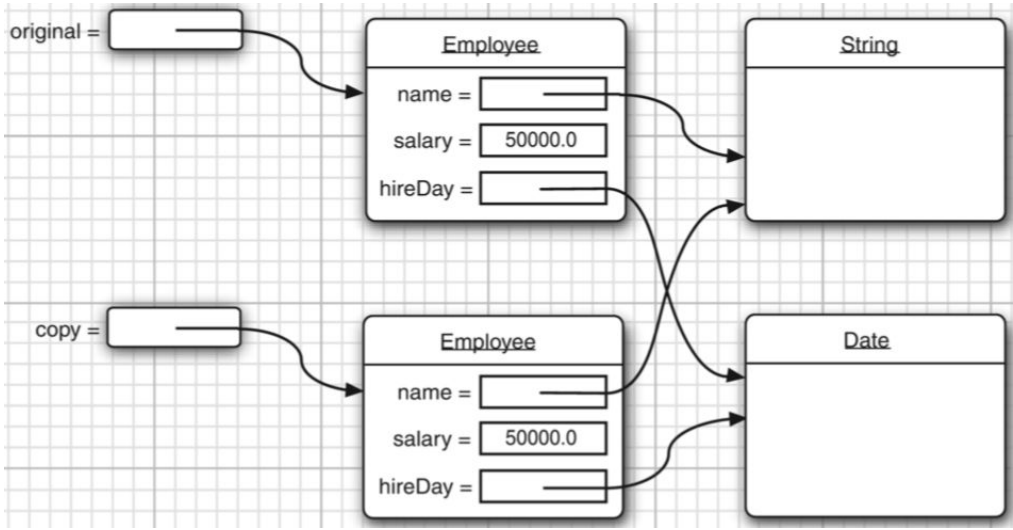


6.2 Examples of Interfaces

6.2.3 Object Cloning

Make only a field-by-field copy:

- A **shallow copy**: immutable types are fine, original and cloned variables refer to the same object.
- A deep copy: **redefine clone()** to make a deep copy that clones the sub objects as well.



6.2 Examples of Interfaces

6.2.3 Object Cloning

```
class Employee implements Cloneable {
    ...
    public Employee clone() throws CloneNotSupportedException {
        // call Object.clone()
        Employee cloned = (Employee) super.clone();

        // clone mutable fields
        cloned.hireDay = (Date) hireDay.clone();

        return cloned;
    }
}
```

Full code: v1ch06.clone

6.3 Lambda Expressions

6.3.1 Why Lambdas?

A lambda expression is **a block of code** that you can pass around so it can be executed later, once or multiple times.

```
class Worker implements ActionListener {
     public void actionPerformed(ActionEvent event) {
     // do some work
     }
}
class LengthComparator implements Comparator<String> {
     public int compare(String first, String second) {
          return first.length() - second.length();
     }
}
```

A block of code was passed to someone—a timer, or a sort method.
That code block was called at some later time.

6.3 Lambda Expressions

6.3.2 The Syntax of Lambda Expressions

parameters, the **->** arrow, and an **expression**

```
class LengthComparator implements Comparator<String> {
     public int compare(String first, String second) {
          return first.length() - second.length();
     }
}
```

```
(String first, String second) -> first.length() - second.length()
```

```
(String first, String second) -> {
     if (first.length() < second.length())
          return -1;
     else if (first.length() > second.length())
          return 1;
     else
          return 0;
}
```

6.3 Lambda Expressions

6.3.2 The Syntax of Lambda Expressions

parameters, the **->** arrow, and an **expression**

- No parameters

```
() -> { for (int i = 100; i >= 0; i--) System.out.println(i); }
```

- No parameter type

```
Comparator<String> comp
= (first, second) // Same as (String first, String second)
-> first.length() - second.length();
```

Full codes: v1ch06.lambda

```
Arrays.sort(planets, (first, second) -> first.length() - second.length());
Timer t = new Timer(1000, event -> System.out.println("The time is " + new Date()));
```

6.3 Lambda Expressions

6.3.3 Functional Interfaces

Many interfaces in Java encapsulate blocks of code, lambdas are compatible with them.

E.g.,

```
ArrayList:
        public boolean removeIf(Predicate<? super E> filter){...}

interface Predicate<T> {
        boolean test(T t);
        //….
}

public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(4); list.add(5); list.add(null);
        list.removeIf(e -> e == null);
        for (Integer i:list){
                System.out.println(i);
        }
}
```

6.3 Lambda Expressions

6.3.4 Method References

```
Timer t = new Timer(1000, event -> System.out.println(event));

Timer t = new Timer(1000, System.out::println);

(x, y) -> Math.pow(x, y)
Math::pow

(x, y) -> x.compareToIgnoreCase(y)
Arrays.sort(strings, String::compareToIgnoreCase)
```

6.3 Lambda Expressions

6.3.4 Method References

```
class Greeter {
    public void greet() {
        System.out.println("Hello, world!");
    }
}

class TimedGreeter extends Greeter {
    public void greet() {
        Timer t = new Timer(1000, super::greet);
        t.start();
    }
}
```

6.3 Lambda Expressions

6.3.6 Variable Scope

The body of a lambda expression has the same scope as a nested block.

```
Path first = Paths.get("/usr/bin");
Comparator<String> comp = (first, second) -> first.length() - second.length();
```

A lambda expression has three ingredients:

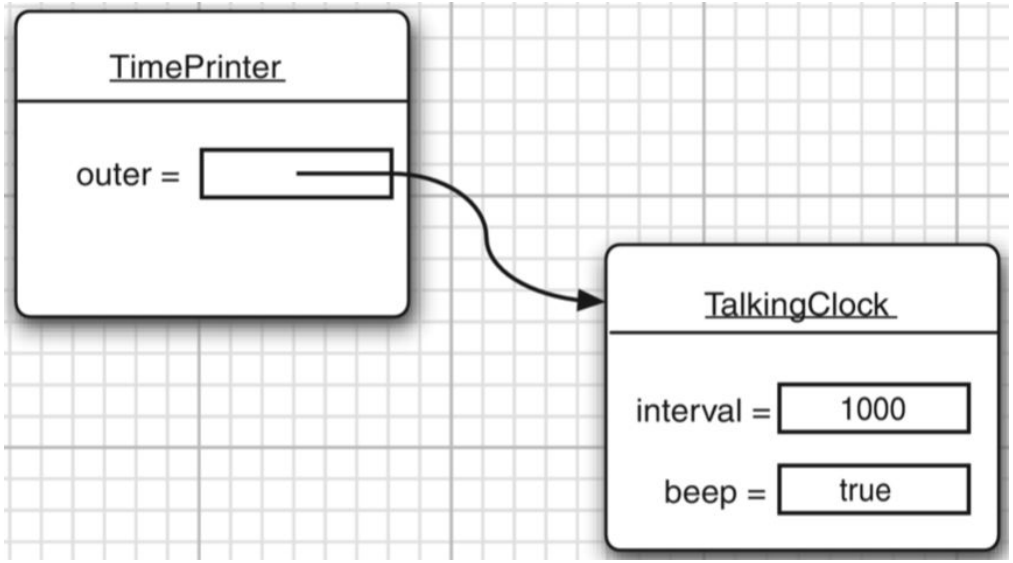
1. A block of code
2. Parameters
3. Free variables (captured variables):
   - effectively final: their values are never changed after their initialization;
   - lambda expression cannot change their values as well.

6.3 Lambda Expressions

```
public static void countDown(int start, int delay) {
     ActionListener listener = event -> {
          start--;
          System.out.println(start);
     };
     new Timer(delay, listener).start();
}

public static void repeat(String text, int count) {
     for (int i = 1; i <= count; i++) {
          ActionListener listener = event -> {
               System.out.println(i + ": " + text);
          };
          new Timer(1000, listener).start();
     }
}
```

6.4 Inner Classes

- An inner class is a class that is defined inside another class.
  - An inner class method **access** both its **own data fields** and those of the **outer object** creating it.
  - Inner classes can be **hidden** from other classes in the same package.
  - **Anonymous inner classes** are handy when you want to **define callbacks without writing a lot of code**.



6.4 Inner Classes

6.4.1 Use of an Inner Class to Access Object State

```
class TalkingClock{
   private int interval;
   private boolean beep;

   public TalkingClock(int interval, boolean beep){
      this.interval = interval;
      this.beep = beep;
   }

   public void start(){
      ActionListener listener = new TimePrinter();
      Timer t = new Timer(interval, listener);
      t.start();
   }

   class TimePrinter implements ActionListener{
      public void actionPerformed(ActionEvent event){
         System.out.println("At the tone, the time is " + new Date());
         if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }
}
```

6.4 Inner Classes

6.4.1 Use of an Inner Class to Access Object State

- The outer class reference:

```
public void actionPerformed (ActionEvent event) {
     if (TalkingClock.this.beep) //beep
          Toolkit.getDefaultToolkit().beep();
}
```

- The inner object constructor:

```
ActionListener listener = this.new TimePrinter(); // new
```

- Refer to an inner class:

```
TalkingClock jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

6.4 Inner Classes

6.4.3 Are Inner Classes Useful?

- Inner classes are **a phenomenon of the compiler**, virtual machine knows nothing about them.
- Inner classes have more access privileges (access private field).
- They are hidden from other class in the same package.

6.4 Inner Classes

TalkingClock, TimePrinter2 are the same package.

```
class TalkingClock {
      private int interval;
      private boolean beep;
      …
      public void start() {
            ActionListener listener = new TimePrinter2(this);
            Timer t = new Timer(interval, listener);
            t.start();
      }
}

class TimePrinter2 implements ActionListener {
      private TalkingClock outer;
      public TimePrinter2(TalkingClock clock) {
            outer = clock;
      }
      public void actionPerformed(ActionEvent event) {
            System.out.println("At the tone, the time is " + new Date());
            if (outer.beep)
                  Toolkit.getDefaultToolkit().beep();
      }
}
```

6.4 Inner Classes

6.4.4 Local Inner Classes

- The class is defined **locally in a single method**.
- Their scope is always restricted to the block in which they are declared.

6.4 Inner Classes

6.4.4 Local Inner Classes

```
class TalkingClockLocal {
        private int interval;
        private boolean beep;
        public TalkingClockLocal(int interval, boolean beep) {
                this.interval = interval;
                this.beep = beep;
        }
        public void start() {
                class TimePrinterLocal implements ActionListener {
                        public void actionPerformed(ActionEvent event) {
                                System.out.println("At the tone, the time is " + new Date());
                                if (beep)
                                        Toolkit.getDefaultToolkit().beep();
                        }
                }
                ActionListener listener = new TimePrinterLocal();
                Timer t = new Timer(interval, listener);
                t.start();
        }
}
```

6.4 Inner Classes

6.4.5 Accessing Variables from Outer Methods

- Local classes can access
  - the fields of their outer classes
  - local variables

6.4 Inner Classes

6.4.5 Accessing Variables from Outer Methods

```
class TalkingClockLocalVar {
        public TalkingClockLocalVar() {
        }
        public void start(int interval, boolean beep) {
                class TimePrinterLocalVar implements ActionListener {
                        public void actionPerformed(ActionEvent event) {
                                System.out.println("At the tone, the time is " + new Date());
                                if (beep) // local variable
                                        Toolkit.getDefaultToolkit().beep();
                        }
                }
                ActionListener listener = new TimePrinterLocalVar();
                Timer t = new Timer(interval, listener);
                t.start();
        }
}
```

6.4 Inner Classes

6.4.6 Anonymous Inner Classes

```
new SuperType(construction parameters) {
    inner class methods and data
}
```

SuperType: class, interface

6.4 Inner Classes

6.4.6 Anonymous Inner Classes

```
class TalkingClockLocalAnony {
        public TalkingClockLocalAnony() {
        }

        public void start(int interval, boolean beep) {
                ActionListener listener = new ActionListener() {
                        public void actionPerformed(ActionEvent event) {
                                System.out.println("At the tone, the time is " + new Date());
                                if (beep)
                                        Toolkit.getDefaultToolkit().beep();
                        }
                };
                Timer t = new Timer(interval, listener);
                t.start();
        }
}
```

6.4 Inner Classes

6.4.6 Anonymous Inner Classes

- Anonymous inner class
- Lambda expression

```
class TalkingClockLocalAnony {
    public TalkingClockLocalAnony() {
    }
    // start: anonymous inner class
    public void start(int interval, boolean beep) {
        ActionListener listener = new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                System.out.println("At the tone, the time is " + new Date());
                if (beep)
                    Toolkit.getDefaultToolkit().beep();
            }
        };
        Timer t = new Timer(interval, listener);
        t.start();
    }
    // start: lambda expression
    public void start(int interval, boolean beep) {
        Timer t = new Timer(interval, event -> {
            System.out.println("At the tone, the time is " + new Date());
            if (beep)
                Toolkit.getDefaultToolkit().beep();
        });
        t.start();
    }
}
```

6.4 Inner Classes

6.4.7 Static Inner Classes

- if you don’t want the inner class to have a reference to the outer class object => use static inner class
- only inner classes can be declared static

Full code: v1ch06.staticInnerClass

6.5 Proxies

6.5.1 When to Use Proxies

- A proxy is used to create new classes at runtime, these classes implement a given set of interfaces.
- Proxies can be used for many purposes, such as:
  - Routing method calls to remote servers
  - Associating user interface events with actions in a running program
  - Tracing method calls for debugging purposes
- The proxy class has the following methods:
  - All methods required by the specified interfaces;
  - All methods defined in the Object class (toString, equals, and so on).

6.5 Proxies

6.5.2 Creating Proxy Objects

To create a proxy object, use the newProxyInstance method of the *Proxy* class

- A class loader: different class loaders, null as the default class loader
- An array of Class objects, one for each interface to be implemented.
- An invocation handler: an object of any class that implements the InvocationHandler interface which has a single method:
  - `Object invoke(Object proxy, Method method, Object[] args)`

```
Object proxy = Proxy.newProxyInstance(null, new Class[] {Comparable.class},
handler);
```

6.5 Proxies

6.5.3 Properties of Proxy Classes

- All proxy classes **extend** the class **Proxy** that has **only one instance field** - the invocation handler
- Any **additional data** must be stored in the **invocation handler**.
- All proxy classes override the **toString**, **equals**, and **hashCode** methods of the Object class.
- There is only one proxy class for **a particular class loader and ordered set of interfaces**.

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

- A proxy class is always **public** and **final**.