Interfaces

Part2 – More on Interfaces

Chapter 6, Core Java Volume I

Table of Contents

- Static Methods in Interfaces
- Default Methods in Interfaces
- Resolving Default Method Conflicts
- Callbacks
- Comparators
- Cloning

Static Methods in Interfaces

- Allowed since Java 8.
- Originally disallowed since it wasn't in the spirit of interfaces as abstract specifications.
- You can find many pairs of interface/companion class in the Java API: *Collection/Collections*, *Path/Paths*
- Paths has a factory method get to make a Path object.
- In Java 8, this would be better solved with a static method in the Path interface:

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

Default Methods in Interfaces

- Can 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
}
```

- More compelling example—event handler:

```
public interface MouseListener
{  // 5 methods
   void mouseClicked(MouseEvent event);
   void mousePressed(MouseEvent event);
   ...
}
```

```
public interface MouseListener
{  // 5 methods
   default void mouseClicked(MouseEvent event) {}
   default void mousePressed(MouseEvent event) {}
   ...
}
```

  - Among 5 methods, programmer usually overrides only one or two methods.

Default Methods in Interfaces

- A default method can call an abstract method:

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

  - Hence, a programmer implementing **Collection** Interface will not worry to implement **isEmpty**() method.

Default Methods in Interfaces

- Consider the `Collection` interface that has been in the JDK for many years.
- Suppose someone provided a class `Bag` implementing `Collection`.
- Later, another method is added to `Collection`. (This actually happened with the `stream` method in JDK 8.)
- If it's not a default method, then `Bag` no longer compiles—the change is not source compatible.
- The same change is binary compatible—the `Bag` class still works, as long as nobody calls the new method. (Otherwise, an `AbstractMethodError` occurs.)
- Making a new interface method a default method solves both problems.

Resolving Default Method Conflicts

- What happens when the exact same method is defined as a default method in one interface and again as a method of a superclass or another interface?
- Two simple rules:
  - *Interfaces clash rule*. If an interface provides a default method and another interface provides the same one (default or not), you must resolve the conflict.
  - *Superclasses win rule*. Concrete superclass methods mask default methods.

The "Interfaces Clash" Rule

- Consider two interfaces:

```
interface Person {  default String getName() { return "John Q. Public"; }; }
interface Named { default String getName() { return getClass().getName() + "_" + hashCode(); } }
```

- What happens if a class implements both interfaces?
- You need to implement the `getName()` method.
- If you like, you can call one or the other interface method:

```
class Student implements Person, Named
{
   public String getName() { return Person.super.getName(); }
   . . .
}
```

- Even if `Named.getName()` is abstract, you must provide `Student.getName()`.
- If both methods are abstract, you can provide an implementation or declare the class *abstract*.

The "Superclasses Win" Rule

- Assume that Student inherits Person class and implements Named interface:

```
class Student extends Person implements Named
{
    . . .
}
```

- Only the superclass method matters.
- The default method Named.getName() is ignored.
- This ensures compatibility with Java 7: If you add a default method to an interface, it has no impact on existing code.

Callbacks

- Callback: Action that should happen when an event occurs.
  - Example: Timer makes callback whenever a time interval has elapsed.
- Give the timer an object of a class that implements this interface:

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

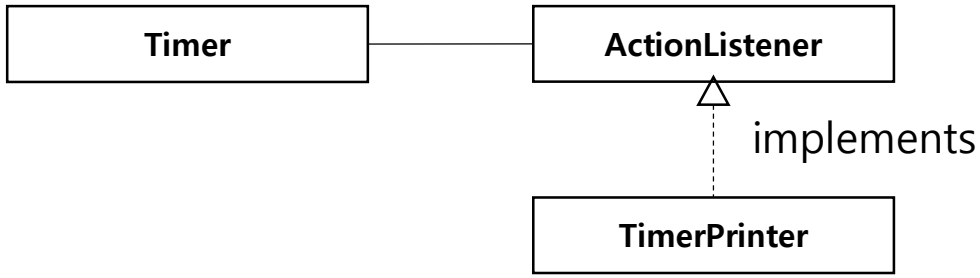
- The timer calls the actionPerformed method:

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

Callbacks

- Construct and install the object:

```
ActionListener listener = new TimePrinter();
Timer t = new Timer(10000, listener); // register a listener to a timer
t.start();
```



Timer *can provide a callback service to any object that implements* ActionListener

Listing 6.3 timer/TimerTest.java

```
package timer;
import java.awt.*;
import java.awt.event.*;
import java.util.*;              //java.util.Timer
import javax.swing.*;
import javax.swing.Timer; // to resolve conflict with java.util.Time
r
public class TimerTest{
   public static void main(String[] args){
      ActionListener listener = new TimePrinter();
      // construct a timer
      //       that calls the listener once every 4 seconds
      Timer t = new Timer(4000, listener);
      t.start();
      JOptionPane.showMessageDialog(null, "Quit program?");
      System.exit(0);
   }// end of main()
} // end of TimerTest clas
```

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

Comparators

- You saw how **Arrays.sort** sorts an array of **Comparable** objects.
  - What if you want to sort the objects in a different way?
  - What if the objects belong to a class that doesn't implement Comparable?
- A second version of **Arrays.sort** accepts a comparator:

```
public interface Comparator<T>
{
   int compare(T first, T second);
}
```

- This comparator compares strings by length:

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

- Pass an instance to **Arrays.sort**:

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends, new LengthComparator());
```

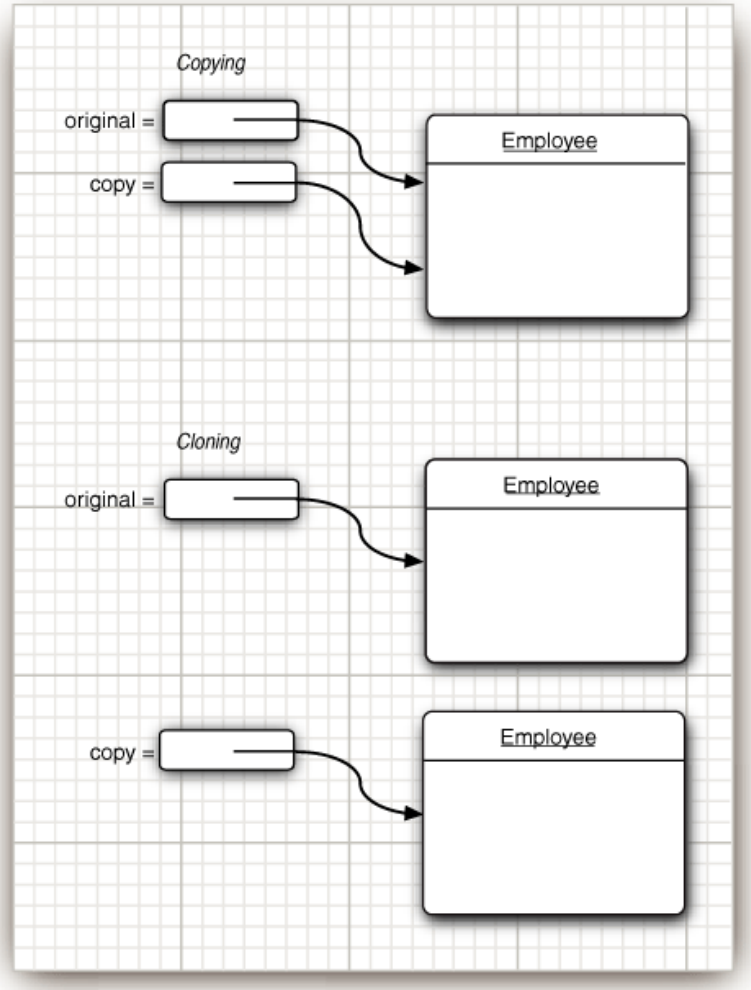
Cloning

- Recall what happens when you make a copy of an object variable:

```
Employee original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10); // oops--also changed original
```

- Remedy: Call the clone method.
- The Cloneable interface indicates that a class provides a safe clone method (*tagging interface*).
- If Employee is cloneable, then you can call

```
public class Employee implements Cloneable
{ … }
…
Employee copy = original.clone();
copy.raiseSalary(10); // OK--original unchanged
```



Cloning

- **Cloneable** is an interface without methods:

  ```
  public interface Cloneable {}
  ```

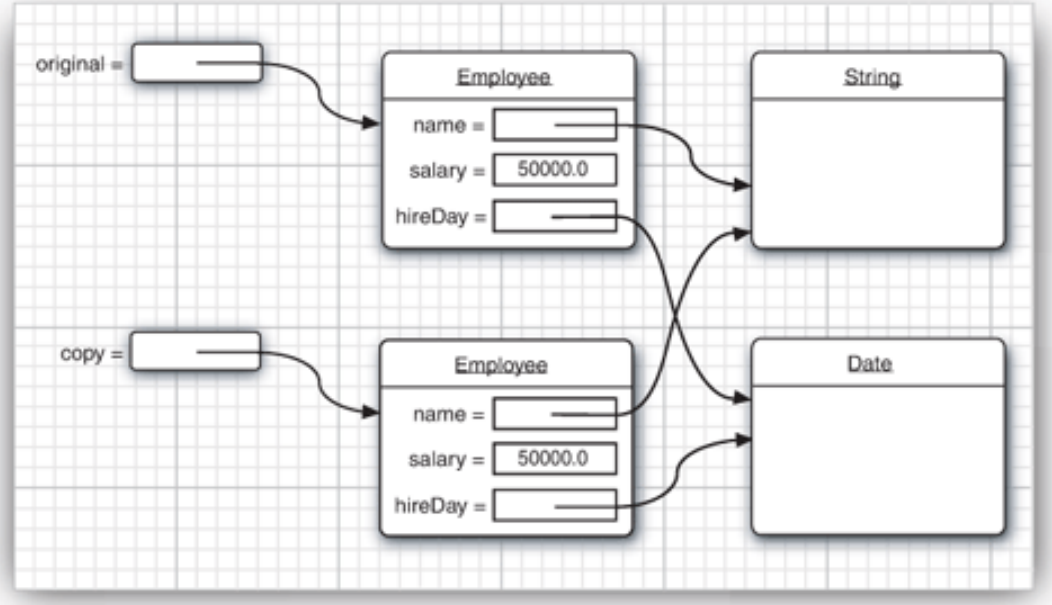
- The **clone** method is a *protected* method of the **Object** class.
- For every class, we need to decide whether
  1. The default clone method is good enough (*shallow copy*)
  2. The default clone method can be patched up by calling clone on the mutable sub-objects (*deep copy*)
  3. clone should not be attempted(default)
- In options 1 and 2, a class must
  1. Implement the "**Cloneable**" interface; and
  2. Re-define the **clone**() method of "**Object**" class with the **public** access modifier

Shallow Copies

- Object.clone() makes a "*shallow*" copy: a new object with the same fields.
- If subobjects are immutable, shallow copy is safe.
- But, it is bad if one of the fields is a reference to a mutable object:

- Even if the default implementation of clone is adequate, you still need to implement the Cloneable interface and redefine clone() to be public.

```
class Employee implements Cloneable
{
  public Emploee clone()
      throws CloneNotSupportedException
  {
     return (Employee) super.clone();
  }
}
```



Deep Copies

- You must implement a deep copy and clone any mutable fields:

```
class Employee implements Cloneable
{
   . . .
   public Employee clone() throws CloneNotSupportedException
   {
      Employee cloned = (Employee) super.clone();
      cloned.hireDay = (Date) hireDay.clone();
      return cloned;
   }
}
```

  - Why does the **clone** methed clone the **name** field?

- Less than 5% of the classes in the Java API are cloneable.

Listing 6.4 Clone/CloneTest.java

```
package clone;  // this program demonstrates cloning
public class CloneTest
{
   public static void main(String[] args)
   {
      try
      {
         Employee original = new Employee("John Q. Public", 50000);
         original.setHireDay(2000, 1, 1);
         Employee copy = original.clone();
         copy.raiseSalary(10);
         copy.setHireDay(2002, 12, 31);
         System.out.println("original=" + original);
         System.out.println("copy=" + copy);
      }
      catch (CloneNotSupportedException e)
      {
         e.printStackTrace();
      }
   }
}
```

Listing 6.4 Clone/CloneTest.java

```
package clone;
import java.util.Date;
import java.util.GregorianCalendar;
public class Employee implements Cloneable
{
 private String name;
 private double salary;
 private Date hireDay;  // mutable object
 public Employee(String name, double salary)
 {
      this.name = name;
      this.salary = salary;
      hireDay = new Date();
 }
public Employee clone() throws
        CloneNotSupportedException
{
   // call Object.clone()
  Employee cloned = (Employee) super.clone();
      // clone mutable fields
  cloned.hireDay = (Date) hireDay.clone();
     return cloned;
}
    // Set the hire day to a given date.
public void setHireDay(int year, int month, int day)
{
  Date newHireDay = new GregorianCalendar
   (year, month -        1, day).getTime();
    // Example of instance field mutation
     hireDay.setTime(newHireDay.getTime());
}
public void raiseSalary(double byPercent)
{
   double raise = salary * byPercent / 100;
   salary += raise;
 }

public String toString()
{
  return "Employee[name=" + name + ",salary="
   + salary + ",hireDay=" + hireDay + "]";
}
} //end of Employee
```