

- 1 接口的概念
- 2 接口的属性
- 3 静态和私有方法
- 4 默认方法
- 5 接口与回调
- 6 Comparator 接口
- 7 对象克隆

# 1 接口的概念

接口不是类，而是对希望符合这个接口的类的一组需求。接口用来描述类应该做什么，而不指定它们具体应该如何做，由类具体实现接口中的方法。一个类可以实现一个或多个接口。

使用关键字 `interface` 定义接口。例如，`Comparable` 接口的代码如下：

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

接口中的所有方法都自动是 `public` 方法。因此，在接口中声明方法时，不必提供关键字 `public`。

接口中可以包含多个方法，也可以定义常量，但不能有实例字段。提供实例字段和方法实现的任务由实现接口的那个类来完成。

类实现接口时，要在类定义处使用关键字 `implements` 指定要实现的接口，并实现接口中的方法。例如，`Employee` 类实现 `Comparable` 接口的代码如下：

```

public class Employee implements Comparable<Employee>
{
    private String name;
    private double salary;

    public Employee(String name, double salary)
    {
        this.name = name;
        this.salary = salary;
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public void raiseSalary(double byPercent)
    {
        double raise = salary * byPercent / 100;
        salary += raise;
    }

    public int compareTo(Employee other) // 实现接口中的方法
    {
        return Double.compare(salary, other.salary);
    }
}

```

这里使用了 `Double` 类中的 `compare()` 方法，它的签名如下：

```

// 如果 x < y 返回一个负数，如果 x==y 返回0，如果 x > y 返回一个正整数
static int compare(double x, double y)

```

`Arrays` 类中的 `sort()` 方法可以对对象数组进行排序，但要求对象所属的类必须实现 `Comparable` 接口。这是因为不同的类有不同的比较方式，必须向 `sort()` 方法提供对象的比较方式。上面已经让 `Employee` 类实现了 `Comparable` 接口，就可以使用 `sort()` 方法对 `Employee` 数组进行排序，例如：

```

Employee staff = new Employee[3];
staff[0] = new Employee("Harry", 35000);
staff[1] = new Employee("Carl", 75000);
staff[2] = new Employee("Tony", 38000);
Arrays.sort(staff);

```

## 2 接口的属性

不能使用 `new` 运算符实例化一个接口，但可以声明接口的变量，接口变量必须引用实现了这个接口的类对象。例如：

```
Comparable x = new Employee("Harry", 35000);
```

可以使用 `instanceof` 检查一个对象是否实现了某个特定的接口。例如：

```
if (obj instanceof Comparable) { ... }
```

接口也可以像类一样继承。例如：

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

public interface Powered extends Moveable
{
    double milesPerGallon();
}
```

接口中可以包含常量，接口中的字段自动被设置为 `public static final`，声明字段时可以省略这些关键字。

接口与抽象类很类似，它们都有未实现的方法，由另外一个类提供抽象方法的实现，但它们并不相同。每个类只能有一个超类，但可以实现多个接口，这就为定义类的行为提供了极大的灵活性。

## 3 静态和私有方法

在 Java 8 中，允许在接口中增加静态方法。此前，通常将静态方法放在伴随类中，构成成对出现的接口和实用工具类，例如 `Collection/Collections` 或 `Path/Paths`。如果可以将静态方法写在接口中，就不需要提供伴随类了。

在 Java 9 中，接口中的方法可以是 `private`。接口中的私有方法只能在接口本身的方法中使用，所以它们用法很有限，只能作为接口中其他方法的辅助方法。

## 4 默认方法

可以为接口方法提供一个默认实现，必须用 `default` 修饰符标记这样的方法。例如：

```
public interface Comparable<T>
{
    default int compareTo(T other) { return 0; }
}
```

在实现这个接口的类中提供方法的具体实现后，默认实现就会被覆盖。

默认方法可以调用其他方法。例如：

```
public interface Collection
{
    int size();

    default boolean isEmpty() { return size() == 0; }
}
```

默认方法的一个重要用法是“接口演化”。假设类 C 实现了接口 F，后来接口 F 添加了一个新方法，如果新方法不是默认方法，类 C 中又没有实现这个新方法，就不能通过编译。如果将新方法实现为默认方法，就不需要修改类 C 的代码。

如果先在一个接口中将一个方法定义为默认方法，然后又在超类或另一个接口中定义同样的方法，会产生二义性。这种情况下的处理规则为：

1. 超类优先。如果超类提供了一个具体方法，接口中同名且有相同参数类型的默认方法会被忽略。
2. 接口冲突。如果一个接口提供了一个默认方法，另一个接口提供了一个同名且参数类型（不论是否是默认参数）相同的方法，必须覆盖这个方法解决冲突。

## 5 接口与回调

**回调**是一种常见的程序设计模式。在这种模式中，可以指定某个特定事件发生时应该采取的动作。

在 `javax.swing` 包中有一个 `Timer` 类，每经过一个时间间隔就发出一个通知。构造 `Timer` 对象时，需要设置一个时间间隔，并告诉定时器经过这个时间间隔时需要做什么。`Timer` 类的部分方法如下：

```
/* javax.swing.Timer */

// 构造一个定时器，每经过 interval 毫秒通知 listener 一次
Timer(int interval, ActionListener listener)

// 启动定时器。一旦启动，定时器将调用监听器的 actionPerformed() 方法
void start()

// 停止定时器。一旦停止，定时器将不再调用监听器的 actionPerformed() 方法
void stop()
```

`ActionListener` 是 `java.awt.event` 包中的一个接口，这个接口的定义如下：

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

可以定义一个类实现 `ActionListener` 接口，向定时器传入这个类的对象。每经过指定的时间间隔，定时器就调用实现的 `actionPerformed()` 方法。例如：

```

1  import java.awt.*;
2  import java.awt.event.*;
3  import java.time.*;
4  import javax.swing.*;
5
6  public class TimerTest
7  {
8      public static void main(String[] args)
9      {
10         // 获取监听器
11         TimePrinter listener = new TimePrinter();
12
13         // 获取定时器, 每隔 1000 毫秒触发一次监听器
14         Timer timer = new Timer(1000, listener);
15
16         // 启动定时器
17         timer.start();
18
19         // 在屏幕上显示一个对话框, 用户点击 OK 按钮后程序停止
20         JOptionPane.showMessageDialog(null, "Quit program?");
21         System.exit(0);
22     }
23 }
24
25 class TimePrinter implements ActionListener
26 {
27     public void actionPerformed(ActionEvent event)
28     {
29         // 输出当前时间, 并发出一声铃响
30         System.out.println("At the tone, the time is " +
31                             Instant.ofEpochMilli(event.getWhen()));
32         Toolkit.getDefaultToolkit().beep();
33     }
34 }

```

上面的例子中用到的方法有：

```

/* javax.swing.JOptionPane */

// 显示一个包含一条提示信息和 OK 按钮的对话框
// 对话框位于 parent 组件的中央。如果 parent 为 null, 对话框将显示在屏幕中央
static void showMessageDialog(Component parent, Object message)

/* java.time.Instant */

// 使用从 1970 年 1 月 1 日 00:00:00 开始的毫秒数获得一个 Instant 实例
static Instant ofEpochMilli(long epochMilli)

/* java.awt.event.ActionEvent */

// 返回发生此事件的时间戳, 表示为从 1970 年 1 月 1 日 00:00:00 开始到现在的毫秒数
long getWhen()

/* java.awt.Toolkit */

// 获得默认的工具箱。工具箱包含有关 GUI 环境的信息
static Toolkit getDefaultToolkit()

// 发出一声铃响
void beep()

```

## 6 Comparator 接口

Arrays 类中的 sort() 方法有另一个版本, 通过 Comparator 接口指定排序方式:

```

/* java.util.Arrays */
static <T> void sort(T[] a, Comparator<? super T> c)
static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)

```

Comparator 接口的定义如下:

```

public interface Comparator<T>
{
    int compare(T o1, T o2);

    // 默认方法在此省略
}

```

例如, 要按长度比较字符串, 可以如下定义一个实现 Comparator<String> 的类:

```

1  import java.util.*;
2
3  public class ComparatorTest
4  {
5      public static void main(String[] args)
6      {
7          String[] friends = {"Peter", "Paul", "Mary"};
8          Arrays.sort(friends, new LengthComparator());
9          System.out.println(Arrays.toString(friends));
10     }
11 }
12
13 class LengthComparator implements Comparator<String>
14 {
15     public int compare(String o1, String o2)
16     {
17         return o1.length() - o2.length();
18     }
19 }

```

## 7 对象克隆

对象变量直接赋值只是让两个变量引用同一个对象，改变其中一个变量，另一个变量也会受到影响，这并不能达到克隆的目的。如果想让两个变量互不影响，就要使用 `clone()` 方法。例如：

```

Employee original = new Employee("John", 50000);
Employee copy = original.clone()

```

`clone()` 方法是 `Object` 类的一个 `protected` 方法。在使用 `clone()` 方法时，要求对象所属的类必须实现 `Cloneable` 接口，重新实现 `clone()` 方法，并指定 `public` 访问修饰符。例如：

```

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

    //...
}

```

`Cloneable` 接口是一个标记接口，它不包含任何方法，只是作为一个标记。如果在一个对象上调用 `clone()` 方法，但这个对象的类没有实现 `Cloneable` 接口，就会抛出 `CloneNotSupportedException` 异常。



上面的 `Employee` 类中实现的 `clone()` 方法使用它的默认操作，将整个对象逐个字段地进行拷贝。如果对象中所有的字段都是基本类型，拷贝不会有问题；但是如果有对象类型的字段，拷贝字段会得到引用同一个对象的另一个变量，相当于直接赋值，这样原对象和克隆对象仍然会共享部分信息。默认的克隆操作是“浅拷贝”，并没有克隆对象中引用的其他对象。如果原对象和浅克隆对象共享的子对象是不可变的，这种共享就是安全的；否则，必须重新定义 `clone()` 方法建立深拷贝，同时克隆所有子对象。例如，`Employee` 类的深拷贝实现如下：

```
class Employee implements Cloneable
{
    private String name; // 不可变类对象
    private double salary; // 基本类型字段
    private Date hireDay; // 可变类对象

    public Employee clone() throws CloneNotSupportedException
    {
        Employee cloned = (Employee) super.clone();
        cloned.hireDay = (Date) hireDay.clone(); // 可变类对象单独处理
        return cloned;
    }

    //...
}
```

所有数组类型都有一个 `public` 的 `clone()` 方法，可以用这个方法建立新数组，包含原数组所有元素的副本。例如：

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
int[] cloned = luckyNumbers.clone();
```