# Lecture13-1 对象模型

## 1. 常见类型

### 介绍

类型：值和可应用于这些值的操作的集合

**强类型语言 Strongly typed language**：编译器和运行时系统检查没有违反类型系统规则的操作可以执行

- 编译时检查

```
Employee e = new Employee();
e.clear(); // ERROR Employee 类不存在该方法
```

- 运行时检查

```
e = null;
e.setSalary( 20000); // ERROR
```

### 类型查询

测试对象 `e` 是否是 `Shape` 类型的

```
if (e instanceof Shape) . . .
```

- 如果 `e` 是 `null`, 测试会返回 **false**（没有异常）

在类型转换之前

```
Shape s = (Shape) e;
```

### Type 和 Value

#### Java 数据类型 Type

- 基本数据类型
- `Class` 类型
- `Interface` 类型
- `Array` 类型
- `null` 类型（可以被赋值给任何引用类型）

`void` 不是一个类型

### Java 值类型 Value

- 基本数据类型的值
- 对 `Class` 类型对象的引用
- 对 `Array` 类型对象的引用
- null

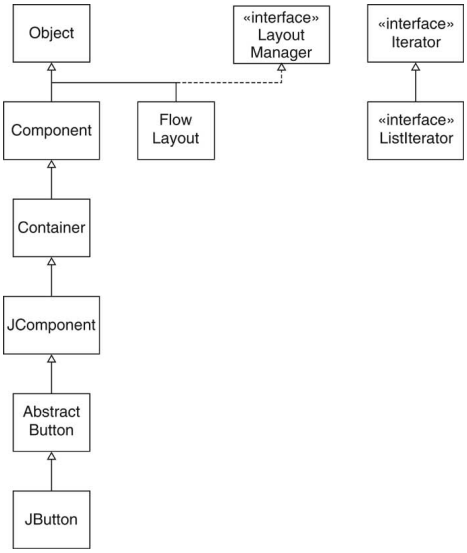
不可能拥有 `Interface` 类型的值

### 子类型关系

我们说 S 是 T 的子数据类型，当

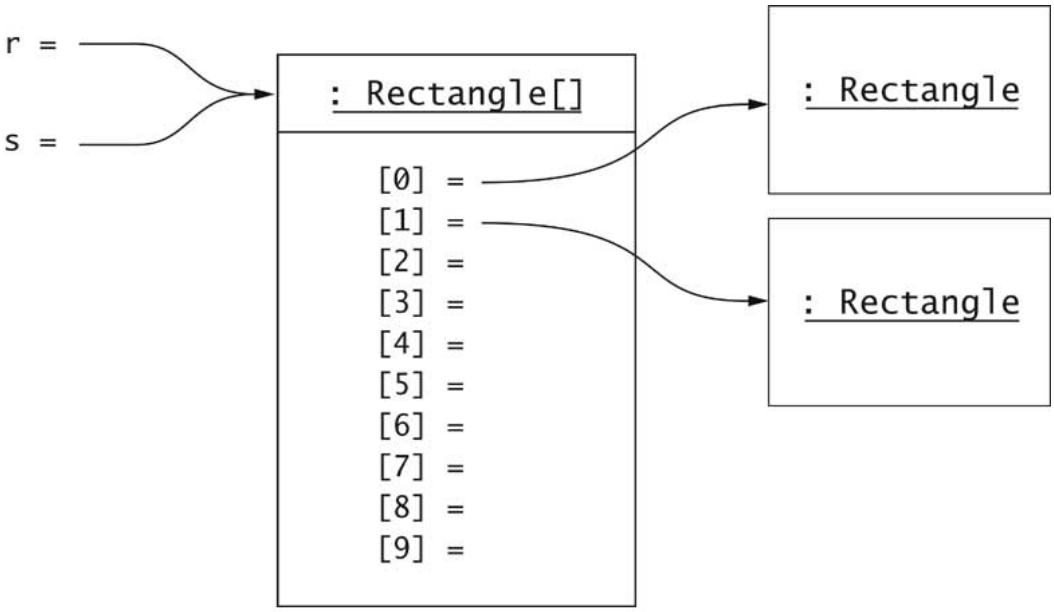
- S 和 T 有一样的数据类型
- S 和 T 都是 `Class` 类型的，且 T 是 S 的直接或非直接的父类
- S 是 `Class` 类型的，T 是 `Interface` 类型的，S 或者它的一个父类实现了 T
- S 和 T 都是 `Interface` 类型的，T 是 S 的直接或非直接的父接口
- S 和 T 都是 `Array` 类型的，S 的元素类型是 T 的元素类型的子类型
- S 不是基本数据类型，T 的类型是 `Object`
- S 是一个 `Array` 类型，T 是一个 `Cloneable` 或者 `Serializable` 类型的
- S 是 `null` 且 T 不是基本数据类型

比如




- `Container` 是 `Component` 的子类型
- `JButton` 是 `Component` 的子类型
- `FlowLayout` 是 `LayoutManager` 的子类型
- `ListIterator` 是 `Iterator` 的子类型
- `Rectangle[]` 是 `Shape[]` 的子类型
- `int[]` 是 `Object` 的子类型
- `int` 不是 `long` 的子类型
- `long` 不是 `int` 的子类型
- `int[]` 不是 `Object[]` 的子类型

### ArrayStoreException




`Rectangle[]` 是 `Shape[]` 的子类型

可以把 `Rectangle[]` 的值赋值给 `Shape[]` 的变量

```
1   Rectangle[] r = new Rectangle[10];
2   Shapes[] s = r;
```

`r` 和 `s` 都引用了相同的数组，数组保存的是 `Rectangle` 类型的变量

如果你要对数组进行下面的操作

```
1   s[0] = new Polygon();
```

那么编译器会在运行时抛出 `ArrayStoreException` 异常，因为每一个数组都知道它内部元素的真实类型

## 包装类 Wrapper Classes

- 基本类型不是类
- 在需要对象时使用包装器
- 每种类型的包装器

```
1   Integer Short Long Byte
2   Character Float Double Boolean
```

- 自动装箱与自动拆箱

```java
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add( 13 );   // calls new Integer( 13 ) 自动装箱
int n = numbers.get( 0 );   // calls intValue(); 自动拆箱
```

## 枚举类 Enum

枚举类等于**同于具有固定数量实例**的类

```java
public class Size {
   private /* ! */ Size() {}
   public static final Size SMALL  = new Size();
   public static final Size MEDIUM = new Size();
   public static final Size LARGE  = new Size();
}
```

它比**整数常量**要安全

```java
public static final int SMALL = 1;
public static final int MEDIUM = 2;
public static final int LARGE = 3;
```

- Enum 类型是类，可以**添加方法，字段，构造函数**

## Class 类

- `getClass()` 方法获得对象所在的类
- 返回的对象的类型是 `Class` 类
- `Class` 对象描述一种数据类型

```java
Object e = new Rectangle();
Class c = e.getClass();
System.out.println( c.getName() ); // prints java.awt.Rectangle
```

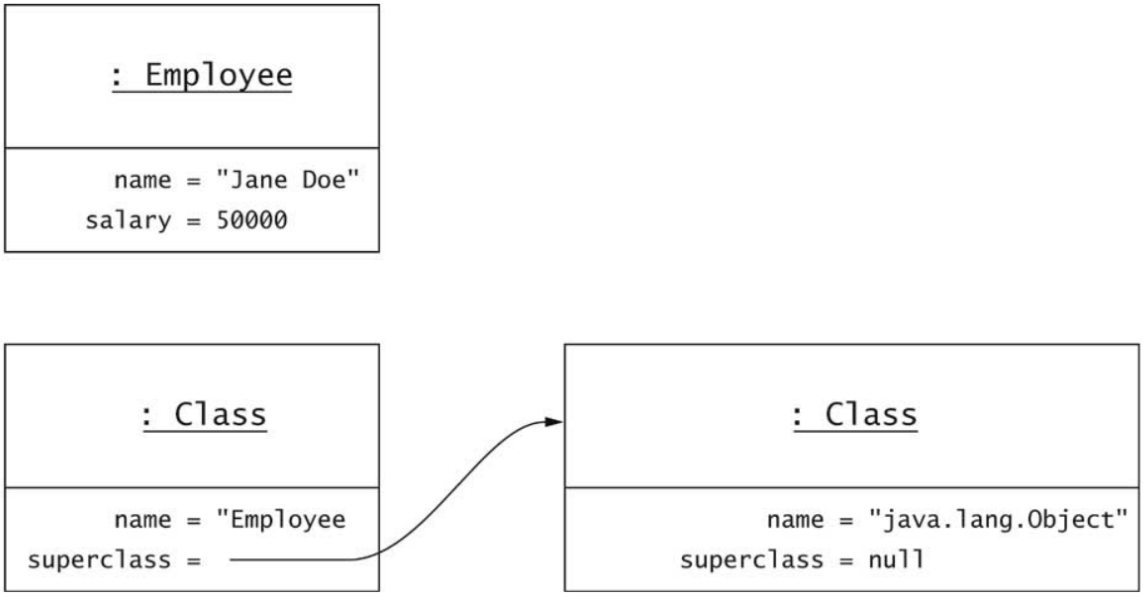
- `Class.forName()` 方法产生一个 `Class` 对象

```java
Class c = Class.forName( "java.awt.Rectangle" );
```

- `.class` 后缀产生一个 `Class` 对象

```java
Class c = Rectangle.class; // java.awt prefix not needed
```

### Employee 对象和 Employee.class 对象




### 对 Class 类的类型查询

测试对象 `e` 是否是 `Rectangle` 类型的

```
1   if (e.getClass() == Rectangle.class) . . .
```

- 可以使用 `==`
- 每一个类都有一个**唯一**的 `Class` 对象
- 无法测试子数据类型，使用 `instanceof` 来测试子数据类型

```
1   if (e instanceof Rectangle) . .
```

### Array 类型的 Class 类

可以对一个 `Array` 类型使用 `getClass()` 方法

将返回一个描述 `Array` 类型的 `Class` 对象

```
1   double[] a = new double[10];
2   Class c = a.getClass();
3   if (c.isArray())
4      System.out.println( c.getComponentType() );
5         // prints double
```

`getName()` 类型对于 `Array` 类型会返回一个奇怪的名字

```
1   [D for double[])
2   [[java.lang.String; for String[][]
```

# 2. Object 类

所有的类都继承 `Object` 类，它提供了一些有用的方法

```
String toString ()
boolean equals (Object otherObject)
Object clone ()
int hashCode ()
```

## toString()

返回该对象的字符串表示形式，对于 debug 很常用

### 介绍

例如 `Rectangle.toString()` 返回类似于

```
java.awt.Rectangle[x=5,y=10,width=20,height=30]
```

`toString()` 可以被连接符使用，下面两者等价

```
aString + anObject
aString + anObject.toString()
```

`Object.toSring()` 打印类的名称和对象的地址

```
System.out.println( System.out ) // java.io.PrintStream@d2460bf
```

### 重写 toString() 方法

```
public class Employee {
   public String toString ()  {
      return getClass().getName()+"[name=" + name + ",salary=" + salary +"]";
   }
   ...
}
```

比如一个示例会返回

```
Employee[name=Harry Hacker,salary=35000]
```

如果有一个子类继承了 `Employee`

```
public class Manager extends Employee {
    public String toString () {
        return super.toString() +
             "[department=" + department + "]";
    }
    ...
}
```

注意，即使此时调用了 `super.toString()` 时候打印了 `getClass().getName()`，但是父类中的方法仍然返回的是实际类的名称

```
Manager[name=Dolly Dollar,salary=100000][department=Finance]
```

## equal()

- `equal()` 方法测试相同的**内容**
- `==` 测试相同的**地址**

如果要比较原始数据类型，可以使用 `==`，如果要比较其它对象，使用 `equals`

### 重写 equal() 方法

通常重写 `equal()` 的定义是比较所有的字段

```
public class Employee {
    protected String name;
    protected int salary;
    public boolean equals (Object otherObject) {
        if(otherObject == null) return false;
        if(!(otherObject instanceof Employee)) return false;
        if (getClass() != otherObject.getClass()) return false;
        // not complete--see below
        Employee other = (Employee)otherObject;
        return name.equals( other.name ) && salary == other.salary;
    }
    ...
}
```

- 必须要将 `Object` 类型的对象转换成子类型

### 子类型的重写 equal() 方法

如果在子类型中添加了字段 `department`，那么重写的 `euqals()` 方法为

```
public class Manager {
    private String department;
    public boolean equals (Object otherObject) {
        Manager other = (Manager)otherObject;
        return super.equals( other ) && department.equals( other.department );
    }
}
```

### 不是所有的 equal() 方法都很简单

如果两个集合的元素有相同的顺序，那么 `equals()` 返回 True

```
public boolean equals (Object o) {
   if (o == this) return true;
   if (!(o instanceof Set)) return false;
   Collection c = (Collection) o;
   if (c.size() != size()) return false;
   return containsAll( c );
}
```

### Object.equal() 方法

```
public class Object {
   public boolean equals (Object obj) {
      return this == obj;
   }
   ...
}
```

- 如果你不想继承这样的行为，重写 `equals()` 方法

### equal() 方法的要求

- 自反的：`x.euqals(x) = true`
- 对称的：`x.equals(y)` 当且仅当 `y.equals(x)`
- 传递的：如果 `x.equals(y)`，`y.equals(z)`，那么 `x.equals(z)`
- 空：`x.equals(null) = false`

一个完美的 `equals()` 方法从下面三个测试开始

```java
public boolean equals (Object otherObject) {
   if (this == otherObject) return true;
   if (otherObject == null) return false;
   if (getClass() != otherObject.getClass()) return false;
   ...
}
```

## hashCode()

- `hashCode()` 方法用在 HashMap，HashSet 里面
- 计算出一个对象的哈希值 `int` 类型
- 例如：`String` 类型的 hashCode

```java
int h = 0;
for (int i = 0; i < s.length(); i++)
   h = 31 * h + s.charAt(i);
```

### 注意

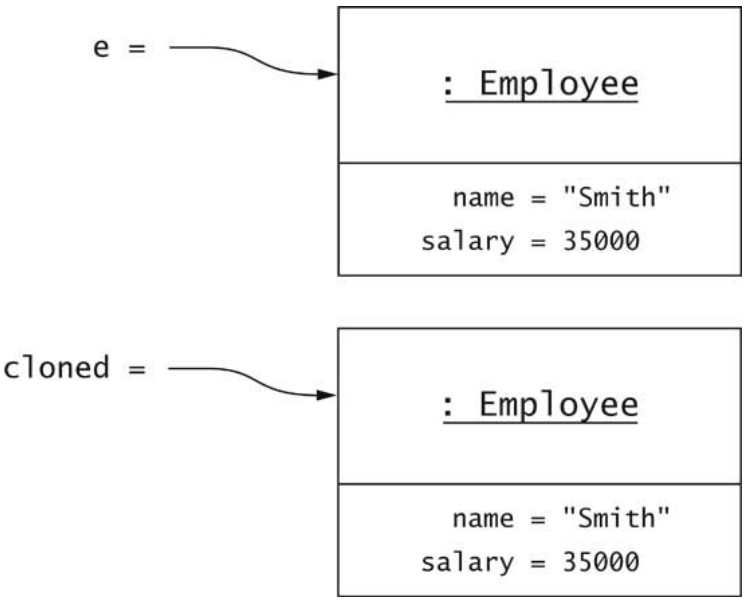
- `hashCode()` 必须与 `equals()` 方法兼容，如果 `x.equals(y)` 那么 `x.hashCode() == y.hashCode()`
- `Object.hashCode()` 哈希了内存地址
  - 如果 `equals()` 方法重写了，那么 `Object.hashCode()` 方法与其不兼容

### 重写 hashCode() 方法

```java
public class Employee {
   public int hashCode ( {
      return name.hashCode() + (new Double( salary )).hashCode();
   }
   ...
}
```

# 3. 浅拷贝与深拷贝

- 赋值 copy = e 做浅拷贝
- Clone 完成深拷贝




```
Employee cloned = (Employee)e.clone();
```

## clone()

- `Object.clone()` 创造一个新的对象，并且拷贝所有的字段，注意它是 `protected` 的方法
- 子类必须重新定义 `clone()` 方法，来让声明它为 `public` 的

```
public class Employee {
   public Object clone () {
      return super.clone(); // not complete
   }
   ...
}
```

### Cloneable 接口

`Object.clone()` 对克隆有很强的约束

它只会克隆那些实现了 `Cloneable` 接口的对象

```
public interface Cloneable { }
```

- 这个接口没有任何的方法
- 它只是一个标记接口，在 `Object.clone()` 中会去测试

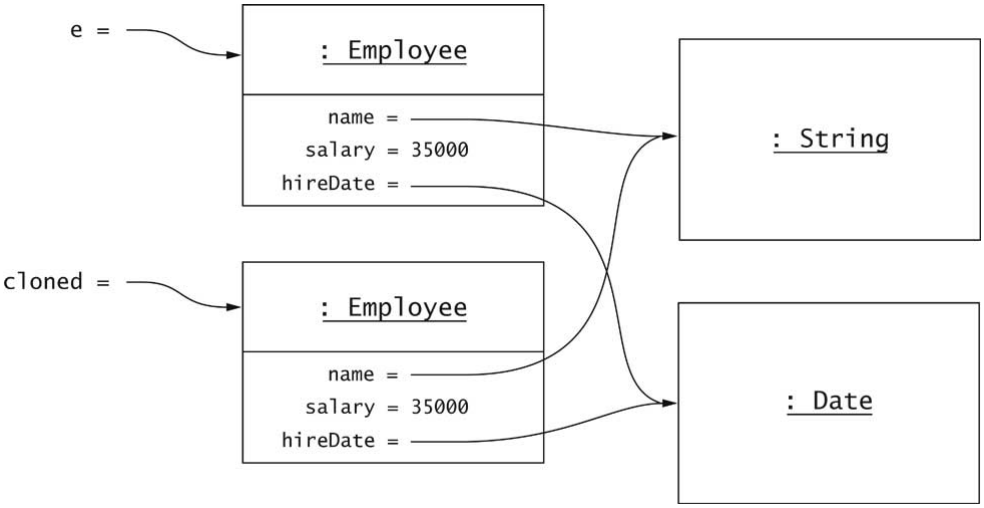
```
if x implements Cloneable
```

- `Object.clone` 抛出 `CloneNotSupportedException`，是一个**检查异常 checked exception**

### clone() 方法

```
1  public class Employee implements Cloneable {
2     public Object clone () {
3        try {
4           return super.clone();
5        } catch (CloneNotSupportedException e) {
6           return null; // won't happen
7        }
8     }
9     ...
10 }
```

## 浅拷贝

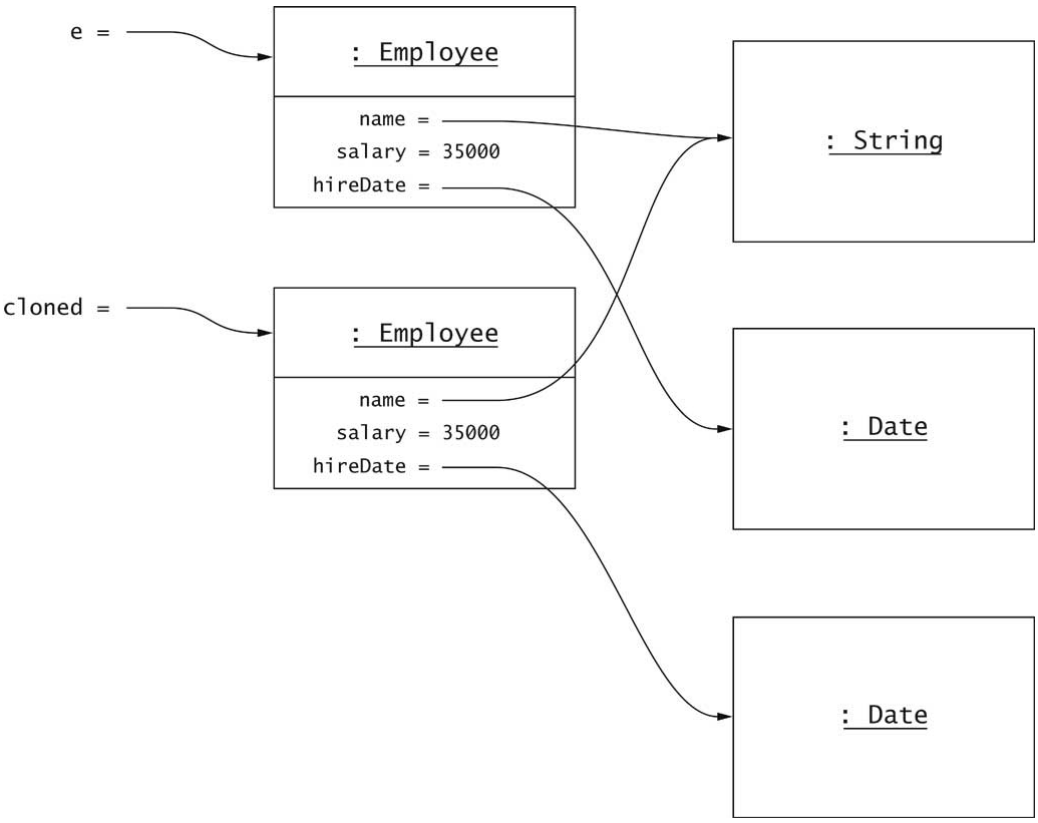



`clone()` 是一种浅拷贝

- 实例字段不会被克隆

## 深拷贝

- 对于不可变字段来说，浅拷贝不是问题
- 但是必须深拷贝**可变字段**




```
1   public class Employee implements Cloneable {
2      public Object clone () {
3         try {
4            Employee cloned = (Employee)super.clone();
5            cloned.hireDate = (Date)hiredate.clone();
6            return cloned;
7         } catch (CloneNotSupportedException e) {
8            return null; // won't happen
9         }
10     }
11     ...
12  }
```

## 克隆和继承

- `Object.clone()` 方法有很多约束
  - `clone()` 方法是 `protected` 的
  - `clone()` 只克隆实现了 `Cloneable` 的对象
  - `clone()` 会抛出检查的异常

规则：如果你继承了一个定义了 `clone()` 的类，那么重新定义 `clone`

## 序列化

### 实现 Serialziable 接口

定义一个类，实现了 `Serializable` 接口

```
class MyObject implements Serializable{
    private static final long serialVersionUID = -5809782578272943999L; // 序列化ID
    \\...
}
```

显式定义序列化 id 的用处

- 在某些场合，**希望类的不同版本对序列化兼容**，因此需要确保类的不同版本具有相同的 serialVersionUID
- 在某些场合，**不希望类的不同版本对序列化兼容**，因此需要确保类的不同版本具有不同的 serialVersionUID

### 序列化

创建 `ObjectOutputStream` 序列化

```
ObjectOutputStream out = new ObjectOutputStream(
   new FileOutputStream( "obj.dat" )
);
out.writeObject(obj);
out.close();
```

### 反序列化

```
ObjectInputStream in = new ObjectInputStream(
   new FileInputStream( "obj.dat" )
);
MyObject obj = (MyObject) in.readObject();
```

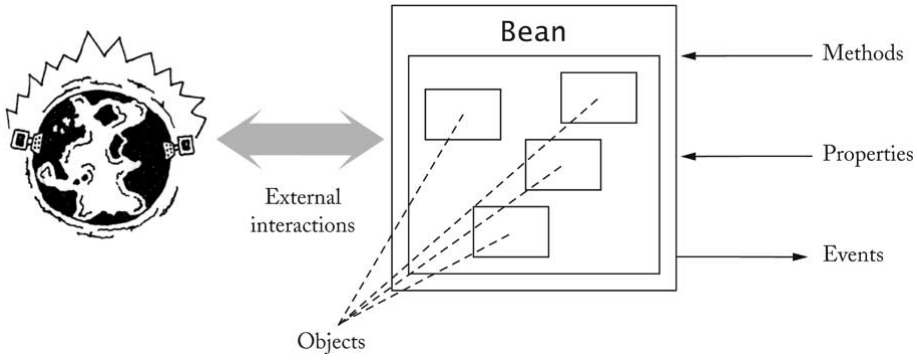
### 利用序列化进行完全深拷贝

```
    public static <T> T deepClone(T src) throws RuntimeException {
        ByteArrayOutputStream memoryBuffer = new ByteArrayOutputStream();
        ObjectOutputStream out = null;
        ObjectInputStream in = null;
        T dist = null;
        try {
            out = new ObjectOutputStream(memoryBuffer);
            out.writeObject(src);
            out.flush();
```

```
            in = new ObjectInputStream(new
    ByteArrayInputStream(memoryBuffer.toByteArray()));
            dist = (T) in.readObject();
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            if (out != null) {
                try {
                    out.close();
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
            if (in != null)
                try {
                    in.close();
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
        }
        return dist;
    }
```

# 4. Java Beans

## 介绍




- Java 组件模型，包含
  - 方法 method
  - 属性 property
  - 事件 event

## Bean 的特点

- 属性 = 可以 get 或 set 的值
- 大多数属性都是可以 get 和可以 set 的
- 也可以只能 get 或者只能 set
- 属性 Property 与实例字段 instance field 的不同
- Setter 可以设置字段，然后调用 repaint
- Getter 可以向数据库查询

## Java 命名约定

- property = 一对方法

```
public X getPropertyName();
public void setPropertyName(X newValue);
```

- boolean 属性

```
public boolean isPropertyName();
```

例如

- `getColor()`
- `setColor()`
- `isColor()`