UML图

统一建模语言（Unified Modeling Language，UML）是用来设计软件的可视化建模语言。它的特点是简单、统一、图形化、能表达软件设计中的动态与静态信息。

UML 从目标系统的不同角度出发，定义了用例图、类图、对象图、状态图、活动图、时序图、协作图、构件图、部署图等 9 种图。

类图概述

类图(Class diagram)是显示了模型的静态结构，特别是模型中存在的类、类的内部结构以及它们与其他类的关系等。类图不显示暂时性的信息。类图是面向对象建模的主要组成部分。

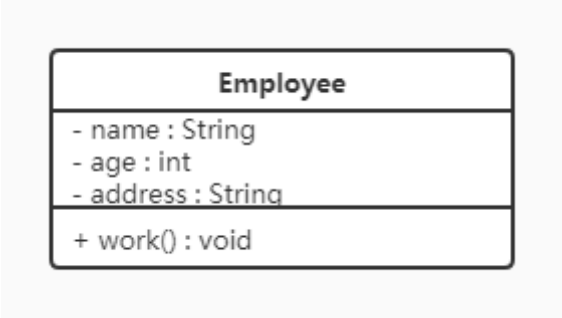
类图的作用

- 在软件工程中，类图是一种静态的结构图，描述了系统的类的集合，类的属性和类之间的关系，可以简化了人们对系统的理解；
- 类图是系统分析和设计阶段的重要产物，是系统编码和测试的重要模型。

类图表示法

类的表示方式

在UML类图中，类使用包含类名、属性(field) 和方法(method) 且带有分割线的矩形来表示，比如下图表示一个Employee类，它包含name,age和address这3个属性，以及work()方法。



属性/方法名称前加的加号和减号表示了这个属性/方法的可见性，UML类图中表示可见性的符号有三种：

- +：表示public
- -：表示private

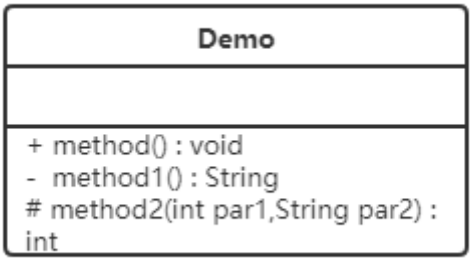
- #：表示protected

属性的完整表示方式是： **可见性 名称 ： 类型 [= 缺省值]**

方法的完整表示方式是： **可见性 名称(参数列表) [： 返回类型]**

> 注意：
> 1，中括号中的内容表示是可选的
> 2，也有将类型放在变量名前面，返回值类型放在方法名前面

举个栗子：



上图Demo类定义了三个方法：

- method()方法：修饰符为public，没有参数，没有返回值。
- method1()方法：修饰符为private，没有参数，返回值类型为String。
- method2()方法：修饰符为protected，接收两个参数，第一个参数类型为int，第二个参数类型为String，返回值类型是int。

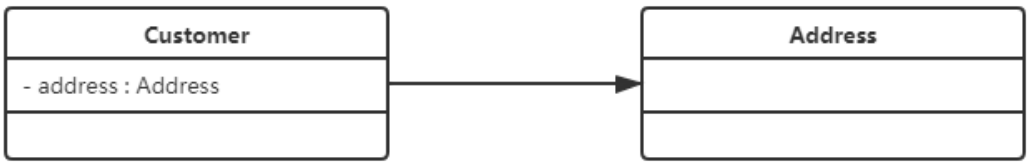
类与类之间关系的表示方式

关联关系

关联关系是对象之间的一种引用关系，用于表示一类对象与另一类对象之间的联系，如老师和学生、师傅和徒弟、丈夫和妻子等。关联关系是类与类之间最常用的一种关系，分为一般关联关系、聚合关系和组合关系。我们先介绍一般关联。

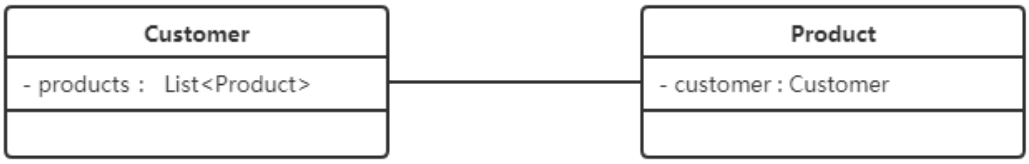
关联又可以分为单向关联，双向关联，自关联。

1，单向关联



在UML类图中单向关联用一个带箭头的实线表示。上图表示每个顾客都有一个地址，这通过让Customer类持有一个类型为Address的成员变量类实现。

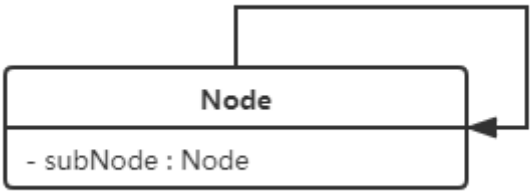
2，双向关联



从上图中我们很容易看出，所谓的双向关联就是双方各自持有对方类型的成员变量。

在UML类图中，双向关联用一个不带箭头的直线表示。上图中在Customer类中维护一个List<Product>，表示一个顾客可以购买多个商品；在Product类中维护一个Customer类型的成员变量表示这个产品被哪个顾客所购买。

3，自关联



自关联在UML类图中用一个带有箭头且指向自身的线表示。上图的意思就是Node类包含类型为Node的成员变量，也就是“自己包含自己”。

聚合关系

聚合关系是关联关系的一种，是强关联关系，是整体和部分之间的关系。

聚合关系也是通过成员对象来实现的，其中成员对象是整体对象的一部分，但是成员对象可以脱离整体对象而独立存在。例如，学校与老师的关系，学校包含老师，但如果学校停办了，老师依然存在。

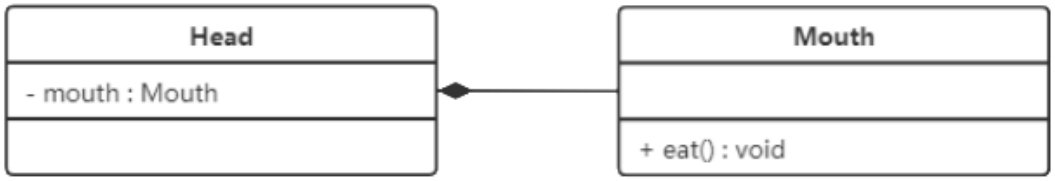
在 UML 类图中，聚合关系可以用带空心菱形的实线来表示，菱形指向整体。下图所示是大学和教师的关系图：

组合关系

组合表示类之间的整体与部分的关系，但它是一种更强烈的聚合关系。

在组合关系中，整体对象可以控制部分对象的生命周期，一旦整体对象不存在，部分对象也将不存在，部分对象不能脱离整体对象而存在。例如，头和嘴的关系，没有了头，嘴也就不存在了。

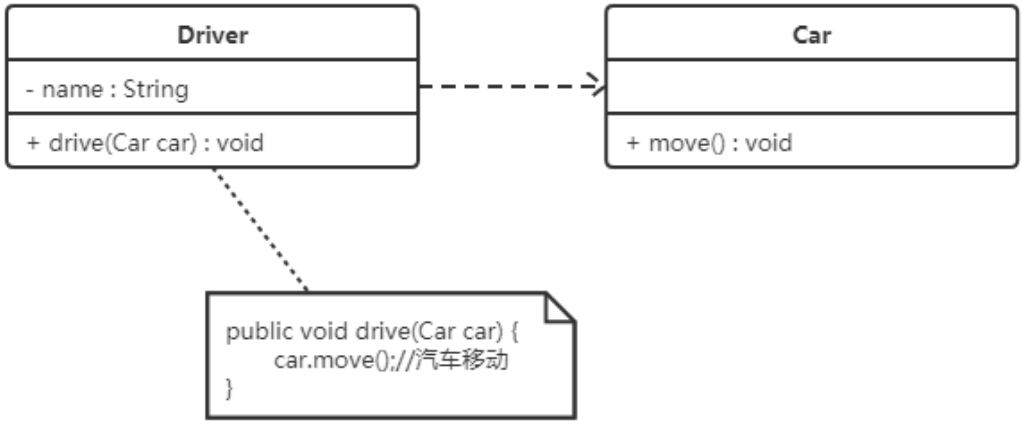
在 UML 类图中，组合关系用带实心菱形的实线来表示，菱形指向整体。下图所示是头和嘴的关系图：



依赖关系

依赖关系是一种使用关系，它是对象之间耦合度最弱的一种关联方式，是临时性的关联。在代码中，某个类的方法通过局部变量、方法的参数或者对静态方法的调用来访问另一个类（被依赖类）中的某些方法来完成一些职责。

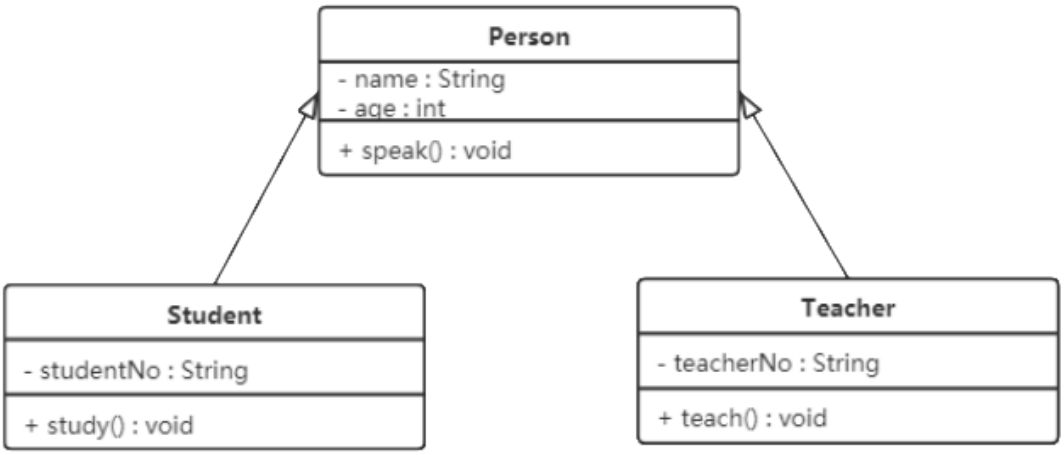
在 UML 类图中，依赖关系使用带箭头的虚线来表示，箭头从使用类指向被依赖的类。下图所示是司机和汽车的关系图，司机驾驶汽车：



继承关系

继承关系是对象之间耦合度最大的一种关系，表示一般与特殊的关系，是父类与子类之间的关系，是一种继承关系。

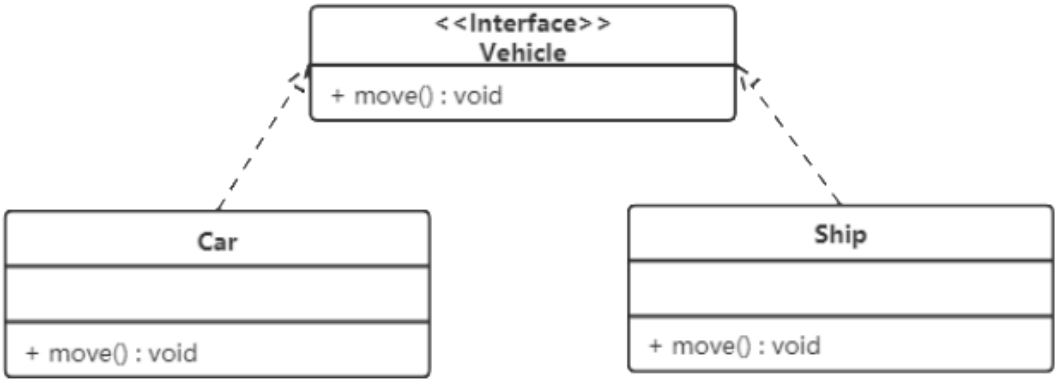
在 UML 类图中，泛化关系用带空心三角箭头的实线来表示，箭头从子类指向父类。在代码实现时，使用面向对象的继承机制来实现泛化关系。例如，Student 类和 Teacher 类都是 Person 类的子类，其类图如下图所示：



实现关系

实现关系是接口与实现类之间的关系。在这种关系中，类实现了接口，类中的操作实现了接口中所声明的所有的抽象操作。

在 UML 类图中，实现关系使用带空心三角箭头的虚线来表示，箭头从实现类指向接口。例如，汽车和船实现了交通工具，其类图如图 9 所示。



类

1.1 如何定义类

类的定义格式如下:

```
修饰符 class 类名 {
    // 1.成员变量（属性）
    // 2.成员方法 (行为)
    // 3.构造方法 （初始化类的对象数据的）
}
```

例如:

```
public class Student {
    // 1.成员变量
    public String name ;
    public char sex ; // '男'  '女'
    public int age;
}
```

1.2 如何通过类创建对象

```
类名 对象名称 = new 类名();
```

例如:

```
Student stu = new Student();
```

1.3 封装

1.3.1 封装的步骤

1.使用 **private** 关键字来修饰成员变量。

2.使用 **public** 修饰getter和setter方法。

1.3.2 封装的步骤实现

1. private修饰成员变量

```
public class Student {
    private String name;
    private int age;
}
```

2. public修饰getter和setter方法

```
public class Student {
    private String name;
    private int age;

    public void setName(String n) {
        name = n;
    }

    public String getName() {
        return name;
    }

    public void setAge(int a) {
        if (a > 0 && a <200) {
            age = a;
        } else {
            System.out.println("年龄非法！");
        }
    }

    public int getAge() {
        return age;
    }
}
```

1.4 构造方法

1.4.1 构造方法的作用

在创建对象的时候，给成员变量进行初始化。

初始化即赋值的意思。

1.4.2 构造方法的格式

```
修饰符 类名(形参列表) {
    // 构造体代码，执行代码
}
```

1.4.3 构造方法的应用

首先定义一个学生类，代码如下：

```
public class Student {
    // 1.成员变量
    public String name;
    public int age;

    // 2.构造方法
    public Student() {
        System.out.println("无参数构造方法被调用");
    }
}
```

接下来通过调用构造方法得到两个学生对象。

```
public class CreateStu02 {
    public static void main(String[] args) {
        // 创建一个学生对象
        // 类名 变量名称 = new 类名();
        Student s1 = new Student();
        // 使用对象访问成员变量，赋值
        s1.name = "张三";
        s1.age = 20 ;

        // 使用对象访问成员变量 输出值
        System.out.println(s1.name);
        System.out.println(s1.age);

        Student s2 = new Student();
        // 使用对象访问成员变量 赋值
        s2.name = "李四";
        s2.age = 18 ;
        System.out.println(s2.name);
        System.out.println(s2.age);
    }
}
```

1.5 this关键字的作用

1.5.1 this关键字的作用

this代表所在类的当前对象的引用（地址值），即代表当前对象。

1.5.2 this关键字的应用

1.5.2.1 用于普通的gettter与setter方法

this出现在实例方法中，谁调用这个方法（哪个对象调用这个方法），this就代表谁（this就代表哪个对象）。

```
public class Student {
    private String name;
    private int age;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setAge(int age) {
        if (age > 0 && age < 200) {
            this.age = age;
        } else {
            System.out.println("年龄非法！");
        }
    }

    public int getAge() {
        return age;
    }
}
```

1.5.2.2 用于构造方法中

this出现在构造方法中，代表构造方法正在初始化的那个对象。

```java
public class Student {
    private String name;
    private int age;

    // 无参数构造方法
    public Student() {}

    // 有参数构造方法
    public Student(String name,int age) {
        this.name = name;
        this.age = age;
    }
}
```

static关键字

2.1 概述

以前我们定义过如下类：

```java
public class Student {
    // 成员变量
    public String name;
    public char sex; // '男'  '女'
    public int age;

    // 无参数构造方法
    public Student() {

    }

    // 有参数构造方法
    public Student(String  a) {

    }
}
```

我们已经知道面向对象中，存在类和对象的概念，我们在类中定义了一些成员变量，例如name,age,sex ,结果发现这些成员变量，每个对象都存在（因为每个对象都可以访问）。

而像name ,age , sex确实是每个学生对象都应该有的属性，应该属于每个对象。

所以Java中成员（**变量和方法**）等是存在所属性的，Java是通过static关键字来区分的。
static关键字在Java开发非常的重要，对于理解面向对象非常关键。

关于 `static` 关键字的使用，它可以用来修饰的成员变量和成员方法，被static修饰的成员是**属于类**的是放在静态区中，没有static修饰的成员变量和方法则是**属于对象**的。我们上面案例中的成员变量都是没有static修饰的，所以属于每个对象。

2.2 定义格式和使用

static是静态的意思。 static可以修饰成员变量或者修饰方法。

2.2.1 静态变量及其访问

有static修饰成员变量，说明这个成员变量是属于类的，这个成员变量称为**类变量**或者**静态成员变量**。 直接用 类名访问即可。因为类只有一个，所以静态成员变量在内存区域中也只存在一份。所有的对象都可以共享这个变量。

如何使用呢

例如现在我们需要定义传智全部的学生类，那么这些学生类的对象的学校属性应该都是“传智”，这个时候我们可以把这个属性定义成static修饰的静态成员变量。

定义格式

```
修饰符 static 数据类型 变量名 = 初始值;
```

举例

```
public class Student {
    public static String schoolName = "传智播客";  // 属于类，只有一份。
    // .....
}
```

静态成员变量的访问:

格式：类名.静态变量

```
public static void  main(String[] args){
    System.out.println(Student.schoolName); // 传智播客
    Student.schoolName = "黑马程序员";
    System.out.println(Student.schoolName); // 黑马程序员
}
```

2.2.2 实例变量及其访问

无static修饰的成员变量属于每个对象的， 这个成员变量叫**实例变量**，之前我们写成员变量就是实例成员变量。

需要注意的是：实例成员变量属于每个对象，必须创建类的对象才可以访问。

格式：对象.实例成员变量

2.2.3 静态方法及其访问

有static修饰成员方法，说明这个成员方法是属于类的，这个成员方法称为**类方法或者**静态方法**。 直接用 类名访问即可。因为类只有一个，所以静态方法在内存区域中也只存在一份。所有的对象都可以共享这个方法。

与静态成员变量一样，静态方法也是直接通过**类名.方法名称**即可访问。

举例

```
public class Student{
    public static String schoolName = "传智播客";  // 属于类，只有一份。
    // .....
    public static void study(){
        System.out.println("我们都在黑马程序员学习");
    }
}
```

静态成员变量的访问:

格式：类名.静态方法

```
public static void  main(String[] args){
    Student.study();
}
```

2.2.4 实例方法及其访问

无static修饰的成员方法属于每个对象的，这个成员方法也叫做**实例方法。**

需要注意的是：实例方法是属于每个对象，必须创建类的对象才可以访问。

格式：对象.实例方法

示例:

```
public class Student {
    // 实例变量
    private String name ;
    // 2.方法：行为
    // 无 static修饰，实例方法。属于每个对象，必须创建对象调用
    public void run(){
        System.out.println("学生可以跑步");
    }
    // 无 static修饰，实例方法
    public  void sleep(){
        System.out.println("学生睡觉");
    }
    public static void study(){
```

```
14
15      }
16  }
```

```
1  public static void main(String[] args){
2      // 创建对象
3      Student stu = new Student ;
4      stu.name = "徐干";
5      // Student.sleep();// 报错，必须用对象访问。
6      stu.sleep();
7      stu.run();
8  }
```

2.3 小结

1.当 `static` 修饰成员变量或者成员方法时，该变量称为**静态变量**，该方法称为**静态方法。**该类的每个对象都**共享**同一个类的静态变量和静态方法。任何对象都可以更改该静态变量的值或者访问静态方法。但是不推荐这种方式去访问。因为静态变量或者静态方法直接通过类名访问即可，完全没有必要用对象去访问。

2.无static修饰的成员变量或者成员方法，称为**实例变量，实例方法**，实例变量和实例方法必须创建类的对象，然后通过对象来访问。

3.static修饰的成员属于类，会存储在静态区，是随着类的加载而加载的，且只加载一次，所以只有一份，节省内存。存储于一块固定的内存区域（静态区），所以，可以直接被类名调用。它优先于对象存在，所以，可以被所有对象共享。

4.无static修饰的成员，是属于对象，对象有多少个，他们就会出现多少份。所以必须由对象调用。

继承

3.1 概述

3.1.1 引入

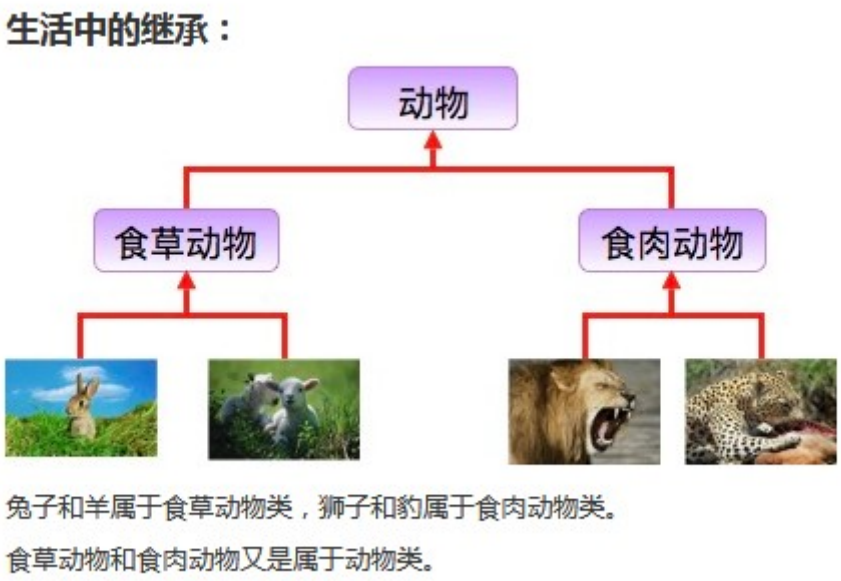
假如我们要定义如下类:
学生类,老师类和工人类，分析如下。

1. 学生类
   属性:姓名,年龄
   行为:吃饭,睡觉

2. 老师类
   属性:姓名,年龄，薪水
   行为:吃饭,睡觉，教书
3. 班主任
   属性:姓名,年龄，薪水
   行为:吃饭,睡觉，管理

如果我们定义了这三个类去开发一个系统，那么这三个类中就存在大量重复的信息（属性:姓名，年龄。行为：吃饭，睡觉）。这样就导致了相同代码大量重复，代码显得很臃肿和冗余，那么如何解决呢?

假如多个类中存在相同属性和行为时，我们可以将这些内容抽取到单独一个类中，那么多个类无需再定义这些属性和行为，只要**继承**那一个类即可。如图所示：



其中，多个类可以称为**子类**，单独被继承的那一个类称为**父类**、**超类（superclass）**或者**基类**。

3.1.2 继承的含义

继承描述的是事物之间的所属关系，这种关系是： `is-a` 的关系。例如，兔子属于食草动物，食草动物属于动物。可见，父类更通用，子类更具体。我们通过继承，可以使多种事物之间形成一种关系体系。

继承：就是子类继承父类的**属性**和**行为**，使得子类对象可以直接具有与父类相同的属性、相同的行为。子类可以直接访问父类中的**非私有**的属性和行为。

3.1.3 继承的好处

1. 提高**代码的复用性**（减少代码冗余，相同代码重复利用）。
2. 使类与类之间产生了关系。

3.2 继承的格式

通过 **extends** 关键字，可以声明一个子类继承另外一个父类，定义格式如下：

```
class 父类 {
    ...
}

class 子类 extends 父类 {
    ...
}
```

需要注意：Java是单继承的，一个类只能继承一个直接父类，跟现实世界很像，但是Java中的子类是更加强大的。

3.3 继承案例

3.3.1 案例

请使用继承定义以下类:

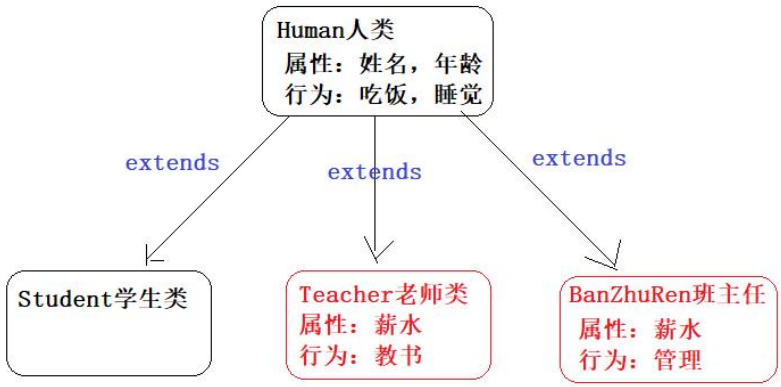
1. 学生类
   属性:姓名,年龄
   行为:吃饭,睡觉
2. 老师类
   属性:姓名,年龄，薪水
   行为:吃饭,睡觉，教书
3. 班主任
   属性:姓名,年龄，薪水
   行为:吃饭,睡觉，管理

3.3.2 案例图解分析

老师类，学生类，还有班主任类，实际上都是属于人类的，我们可以定义一个人类，把他们相同的属性和行为都定义在人类中，然后继承人类即可，子类特有的属性和行为就定义在子类中了。

如下图所示。

1. 学生类
   属性:姓名,年龄
   行为:吃饭,睡觉
2. 老师类
   属性:姓名,年龄，薪水
   行为:吃饭,睡觉，教书
3. 班主任
   属性:姓名,年龄，薪水
   行为:吃饭,睡觉，管理



3.3.3 案例代码实现

1.父类Human类

```java
public class Human {
    // 合理隐藏
    private String name ;
    private int age ;

    // 合理暴露
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

2.子类Teacher类

```java
public class Teacher extends Human {
    // 工资
    private double salary ;

    // 特有方法
    public void teach(){
        System.out.println("老师在认真教技术！");
    }

```

```
    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
}
```

3.子类Student类

```
public class Student extends Human{

}
```

4.子类BanZhuren类

```
public class Teacher extends Human {
    // 工资
    private double salary ;

      // 特有方法
    public void admin(){
        System.out.println("班主任强调纪律问题！");
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
}
```

5.测试类

```
  public class Test {
      public static void main(String[] args) {
          Teacher dlei = new Teacher();
          dlei.setName("播仔");
          dlei.setAge("31");
          dlei.setSalary(1000.99);
          System.out.println(dlei.getName());
          System.out.println(dlei.getAge());
          System.out.println(dlei.getSalary());
          dlei.teach();

          BanZhuRen linTao = new BanZhuRen();
          linTao.setName("灵涛");
```

```
        linTao.setAge("28");
        linTao.setSalary(1000.99);
        System.out.println(linTao.getName());
        System.out.println(linTao.getAge());
        System.out.println(linTao.getSalary());
        linTao.admin();

        Student xugan = new Student();
        xugan.setName("播仔");
        xugan.setAge("31");
        //xugan.setSalary(1000.99); // xugan没有薪水属性，报错!
        System.out.println(xugan.getName());
        System.out.println(xugan.getAge());



    }
}
```

3.3.4 小结

1.继承实际上是子类相同的属性和行为可以定义在父类中，子类特有的属性和行为由自己定义，这样就实现了相同属性和行为的重复利用，从而提高了代码复用。

2.子类继承父类，就可以直接得到父类的成员变量和方法。是否可以继承所有成分呢？请看下节!

3.4 子类不能继承的内容

3.4.1 引入

并不是父类的所有内容都可以给子类继承的：

子类不能继承父类的构造方法。

值得注意的是子类可以继承父类的私有成员（成员变量，方法），只是子类无法直接访问而已，可以通过getter/setter方法访问父类的private成员变量。

3.4.1 演示代码

```
public class Demo03 {
    public static void main(String[] args) {
        Zi z = new Zi();
        System.out.println(z.num1);
//      System.out.println(z.num2); // 私有的子类无法使用
        // 通过getter/setter方法访问父类的private成员变量
```

```
        System.out.println(z.getNum2());

        z.show1();
        // z.show2(); // 私有的子类无法使用
    }
}

class Fu {
    public int num1 = 10;
    private int num2 = 20;

    public void show1() {
        System.out.println("show1");
    }

    private void show2() {
        System.out.println("show2");
    }

    public int getNum2() {
        return num2;
    }

    public void setNum2(int num2) {
        this.num2 = num2;
    }
}

class Zi extends Fu {
}
```

3.5 继承后的特点—成员变量

当类之间产生了继承关系后，其中各类中的成员变量，又产生了哪些影响呢？

3.5.1 成员变量不重名

如果子类父类中出现**不重名**的成员变量，这时的访问是**没有影响的。**代码如下：

```
class Fu {
    // Fu中的成员变量
    int num = 5;
}
class Zi extends Fu {
    // Zi中的成员变量
    int num2 = 6;

```

```
    // Zi中的成员方法
    public void show() {
        // 访问父类中的num
        System.out.println("Fu num="+num); // 继承而来，所以直接访问。
        // 访问子类中的num2
        System.out.println("Zi num2="+num2);
    }
}
class Demo04 {
    public static void main(String[] args) {
        // 创建子类对象
        Zi z = new Zi();
        // 调用子类中的show方法
        z.show();
    }
}

演示结果:
Fu num = 5
Zi num2 = 6
```

3.5.2 成员变量重名

如果子类父类中出现**重名**的成员变量，这时的访问是**有影响的**。代码如下：

```
class Fu1 {
    // Fu中的成员变量。
    int num = 5;
}
class Zi1 extends Fu1 {
    // Zi中的成员变量
    int num = 6;

    public void show() {
        // 访问父类中的num
        System.out.println("Fu num=" + num);
        // 访问子类中的num
        System.out.println("Zi num=" + num);
    }
}
class Demo04 {
    public static void main(String[] args) {
        // 创建子类对象
        Zi1 z = new Zi1();
        // 调用子类中的show方法
        z1.show();
    }
}
```

```
演示结果:
Fu num = 6
Zi num = 6
```

子父类中出现了同名的成员变量时，子类会优先访问自己对象中的成员变量。如果此时想访问父类成员变量如何解决呢？我们可以使用super关键字。

3.5.3 super访问父类成员变量

子父类中出现了同名的成员变量时，在子类中需要访问父类中非私有成员变量时，需要使用 **super** 关键字，修饰父类成员变量，类似于之前学过的 **this** 。

需要注意的是：**super代表的是父类对象的引用，this代表的是当前对象的引用。**

使用格式：

```
super.父类成员变量名
```

子类方法需要修改，代码如下：

```
class Fu {
    // Fu中的成员变量。
    int num = 5;
}

class Zi extends Fu {
    // Zi中的成员变量
    int num = 6;

    public void show() {
        int num = 1;

        // 访问方法中的num
        System.out.println("method num=" + num);
        // 访问子类中的num
        System.out.println("Zi num=" + this.num);
        // 访问父类中的num
        System.out.println("Fu num=" + super.num);
    }
}

class Demo04 {
    public static void main(String[] args) {
        // 创建子类对象
        Zi1 z = new Zi1();
        // 调用子类中的show方法
        z1.show();
    }
}
```

```
演示结果：
method num=1
Zi num=6
Fu num=5
```

> 小贴士：Fu 类中的成员变量是非私有的，子类中可以直接访问。若Fu 类中的成员变量私有了，子类是不能直接访问的。通常编码时，我们遵循封装的原则，使用private修饰成员变量，那么如何访问父类的私有成员变量呢？对！可以在父类中提供公共的getXxx方法和setXxx方法。

3.6 继承后的特点—成员方法

当类之间产生了关系，其中各类中的成员方法，又产生了哪些影响呢？

3.6.1 成员方法不重名

如果子类父类中出现**不重名**的成员方法，这时的调用是**没有影响的。**对象调用方法时，会先在子类中查找有没有对应的方法，若子类中存在就会执行子类中的方法，若子类中不存在就会执行父类中相应的方法。代码如下：

```
class Fu {
    public void show() {
        System.out.println("Fu类中的show方法执行");
    }
}
class Zi extends Fu {
    public void show2() {
        System.out.println("Zi类中的show2方法执行");
    }
}
public  class Demo05 {
    public static void main(String[] args) {
        Zi z = new Zi();
        //子类中没有show方法，但是可以找到父类方法去执行
        z.show();
        z.show2();
    }
}
```

3.6.2 成员方法重名

如果子类父类中出现**重名**的成员方法，则创建子类对象调用该方法的时候，子类对象会优先调用自己的方法。

代码如下：

```
class Fu {
    public void show() {
        System.out.println("Fu show");
    }
}
class Zi extends Fu {
    //子类重写了父类的show方法
    public void show() {
        System.out.println("Zi show");
    }
}
public class ExtendsDemo05{
    public static void main(String[] args) {
        Zi z = new Zi();
        // 子类中有show方法，只执行重写后的show方法
        z.show();  // Zi show
    }
}
```

3.7 方法重写

3.7.1 概念

方法重写：子类中出现与父类一模一样的方法时（返回值类型，方法名和参数列表都相同），会出现覆盖效果，也称为重写或者复写。**声明不变，重新实现。**

3.7.2 使用场景与案例

发生在子父类之间的关系。
子类继承了父类的方法，但是子类觉得父类的这方法不足以满足自己的需求，子类重新写了一个与父类同名的方法，以便覆盖父类的该方 法。

例如：我们定义了一个动物类代码如下：

```
public class Animal  {
    public void run(){
        System.out.println("动物跑的很快！");
    }
    public void cry(){
        System.out.println("动物都可以叫~~~");
    }
}
```

然后定义一个猫类，猫可能认为父类cry()方法不能满足自己的需求

代码如下：

```
public class Cat extends Animal {
    public void cry(){
        System.out.println("我们一起学猫叫，喵喵喵！喵的非常好听！");
    }
}

public class Test {
    public static void main(String[] args) {
        // 创建子类对象
        Cat ddm = new Cat();
        // 调用父类继承而来的方法
        ddm.run();
        // 调用子类重写的方法
        ddm.cry();
    }
}
```

3.7.2 @Override重写注解

- @Override:注解，重写注解校验！
- 这个注解标记的方法，就说明这个方法必须是重写父类的方法，否则编译阶段报错。
- 建议重写都加上这个注解，一方面可以提高代码的可读性，一方面可以防止重写出错！

  加上后的子类代码形式如下：

  ```
  public class Cat extends Animal {
      // 声明不变，重新实现
     // 方法名称与父类全部一样，只是方法体中的功能重写写了！
      @Override
      public void cry(){
          System.out.println("我们一起学猫叫，喵喵喵！喵的非常好听！");
      }
  }
  ```

3.7.3 注意事项

1. 方法重写是发生在子父类之间的关系。
2. 子类方法覆盖父类方法，必须要保证权限大于等于父类权限。
3. 子类方法覆盖父类方法，**返回值类型**、**函数名**和**参数列表**都要一模一样。

3.8 继承后的特点—构造方法

3.8.1 引入

当类之间产生了关系，其中各类中的构造方法，又产生了哪些影响呢？
首先我们要回忆两个事情，构造方法的定义格式和作用。

1. 构造方法的名字是与类名一致的。所以子类是无法继承父类构造方法的。
2. 构造方法的作用是初始化对象成员变量数据的。所以子类的初始化过程中，必须先执行父类的初始化动作。子类的构造方法中默认有一个 `super()` ，表示调用父类的构造方法，父类成员变量初始化后，才可以给子类使用。（**先有爸爸，才能有儿子**）

继承后子类构方法器特点:子类所有构造方法的第一行都会默认先调用父类的无参构造方法

3.8.2 案例演示

按如下需求定义类:

1. 人类
   成员变量: 姓名,年龄
   成员方法: 吃饭
2. 学生类
   成员变量: 姓名,年龄,成绩
   成员方法: 吃饭

代码如下：

```
class Person {
    private String name;
    private int age;

    public Person() {
        System.out.println("父类无参");
    }

    // getter/setter省略
}
```

```

class Student extends Person {
    private double score;

    public Student() {
        //super(); // 调用父类无参,默认就存在，可以不写，必须在第一行
        System.out.println("子类无参");
    }

     public Student(double score) {
        //super();  // 调用父类无参,默认就存在，可以不写，必须在第一行
        this.score = score;
        System.out.println("子类有参");
     }

}

public class Demo07 {
    public static void main(String[] args) {
        Student s1 = new Student();
        System.out.println("----------");
        Student s2 = new Student(99.9);
    }
}

输出结果:
父类无参
子类无参
----------
父类无参
子类有参
```

3.8.3 小结

- 子类构造方法执行的时候，都会在第一行默认先调用父类无参数构造方法一次。
- 子类构造方法的第一行都隐含了一个**super()**去调用父类无参数构造方法，**super()**可以省略不写。

3.9 super(...)和this(...)

3.9.1 引入

请看上节中的如下案例:

```
class Person {
    private String name;
```

```
    private int age;

    public Person() {
        System.out.println("父类无参");
    }

    // getter/setter省略
}

class Student extends Person {
    private double score;

    public Student() {
        //super(); // 调用父类无参构造方法,默认就存在，可以不写，必须再第一行
        System.out.println("子类无参");
    }

     public Student(double score) {
        //super();  // 调用父类无参构造方法,默认就存在，可以不写，必须再第一行
        this.score = score;
        System.out.println("子类有参");
     }
      // getter/setter省略
}

public class Demo07 {
    public static void main(String[] args) {
        // 调用子类有参数构造方法
        Student s2 = new Student(99.9);
        System.out.println(s2.getScore()); // 99.9
        System.out.println(s2.getName()); // 输出 null
        System.out.println(s2.getAge()); // 输出 0
    }
}
```

我们发现，子类有参数构造方法只是初始化了自己对象中的成员变量score，而父类中的成员变量name和age依然是没有数据的，怎么解决这个问题呢，我们可以借助与super(...)去调用父类构造方法，以便初始化继承自父类对象的name和age.

3.9.2 super和this的用法格式

super和this完整的用法如下，其中this，super访问成员我们已经接触过了。

```
this.成员变量      --   本类的
super.成员变量     --   父类的

this.成员方法名()   --   本类的
super.成员方法名()  --   父类的
```

接下来我们使用调用构造方法格式：

```
super(...) -- 调用父类的构造方法，根据参数匹配确认
this(...) -- 调用本类的其他构造方法，根据参数匹配确认
```

3.9.3 super(....)用法演示

代码如下：

```
class Person {
    private String name ="凤姐";
    private int age = 20;

    public Person() {
        System.out.println("父类无参");
    }

    public Person(String name , int age){
        this.name = name ;
        this.age = age ;
    }

    // getter/setter省略
}

class Student extends Person {
    private double score = 100;

    public Student() {
        //super(); // 调用父类无参构造方法,默认就存在，可以不写，必须再第一行
        System.out.println("子类无参");
    }

     public Student(String name ，  int age，double score) {
        super(name ,age);// 调用父类有参构造方法Person(String name , int age)初始化name和age
        this.score = score;
        System.out.println("子类有参");
     }
      // getter/setter省略
}

public class Demo07 {
    public static void main(String[] args) {
        // 调用子类有参数构造方法
        Student s2 = new Student("张三"，20，99);
        System.out.println(s2.getScore()); // 99
        System.out.println(s2.getName()); // 输出 张三
```

```
39          System.out.println(s2.getAge()); // 输出 20
40      }
41  }
```

注意：

子类的每个构造方法中均有默认的super()，调用父类的空参构造。手动调用父类构造会覆盖默认的super()。

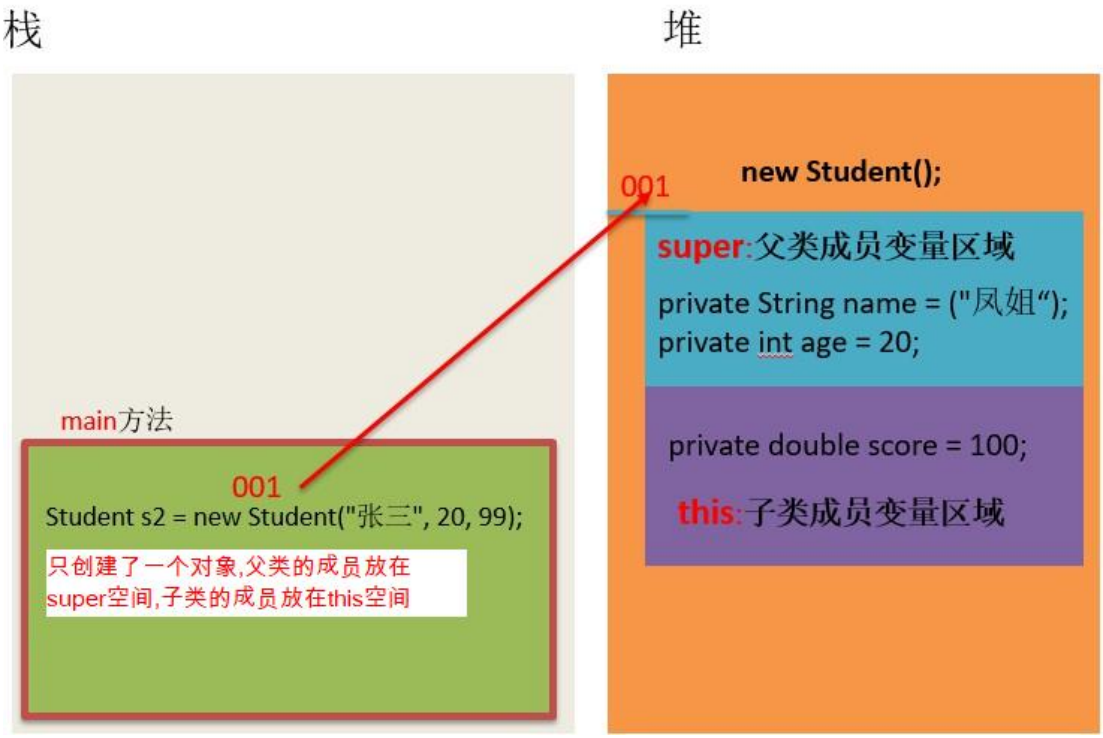
super() 和 this() 都必须是在构造方法的第一行，所以不能同时出现。

super(..)是根据参数去确定调用父类哪个构造方法的。

3.9.4 super(...)案例图解

父类空间优先于子类对象产生

在每次创建子类对象时，**先初始化父类空间，再创建其子类对象本身**。目的在于子类对象中包含了其对应的父类空间，便可以包含其父类的成员，如果父类成员非private修饰，则子类可以随意使用父类成员。代码体现在子类的构造七调用时，一定先调用父类的构造方法。理解图解如下：



3.9.5 this(...)用法演示

this(...)

- 默认是去找本类中的其他构造方法，根据参数来确定具体调用哪一个构造方法。
- 为了借用其他构造方法的功能。

```
package com.itheima._08this和super调用构造方法;
/**
 * this(...):
 *    默认是去找本类中的其他构造方法，根据参数来确定具体调用哪一个构造方法。
 *    为了借用其他构造方法的功能。
 *
 */
public class ThisDemo01 {
    public static void main(String[] args) {
        Student xuGan = new Student();
        System.out.println(xuGan.getName()); // 输出:徐干
        System.out.println(xuGan.getAge());// 输出:21
        System.out.println(xuGan.getSex());// 输出：  男
    }
}

class Student{
    private String name ;
    private int age ;
    private char sex ;

    public Student() {
  // 很弱，我的兄弟很牛逼啊，我可以调用其他构造方法：Student(String name, int age, char sex)
        this("徐干",21,'男');
    }

    public Student(String name, int age, char sex) {
        this.name = name ;
        this.age = age   ;
        this.sex = sex   ;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
40
41      public int getAge() {
42          return age;
43      }
44
45      public void setAge(int age) {
46          this.age = age;
47      }
48
49      public char getSex() {
50          return sex;
51      }
52
53      public void setSex(char sex) {
54          this.sex = sex;
55      }
56  }
```

3.9.6 小结

- **子类的每个构造方法中均有默认的super()，调用父类的空参构造。手动调用父类构造会覆盖默认的super()。**
- **super() 和 this() 都必须是在构造方法的第一行，所以不能同时出现。**
- **super(..)和this(...)是根据参数去确定调用父类哪个构造方法的。**
- super(..)可以调用父类构造方法初始化继承自父类的成员变量的数据。
- this(..)可以调用本类中的其他构造方法。

3.10 继承的特点

1. Java只支持单继承，不支持多继承。

```
1  // 一个类只能有一个父类，不可以有多个父类。
2  class A {}
3  class B {}
4  class C1 extends A {} // ok
5  // class C2 extends A, B {} // error
```

2. 一个类可以有多个子类。

```
1  // A可以有多个子类
2  class A {}
3  class C1 extends A {}
4  class C2 extends  A {}
```

3. 可以多层继承。

```
class A {}
class C1 extends A {}
class D extends C1 {}
```

> 顶层父类是Object类。所有的类默认继承Object，作为父类。

小结:

会写一个继承结构下的标准Javabean即可

需求:

猫：属性，姓名，年龄，颜色

狗：属性，姓名，年龄，颜色，吼叫

分享书写技巧:

1.在大脑中要区分谁是父，谁是子

2.把共性写到父类中，独有的东西写在子类中

3.开始编写标准Javabean（从上往下写）

4.在测试类中，创建对象并赋值调用

代码示例:

```
package com.itheima.test4;

public class Animal {
    //姓名，年龄，颜色
    private String name;
    private int age;
    private String color;


    public Animal() {
    }

    public Animal(String name, int age, String color) {
        this.name = name;
        this.age = age;
        this.color = color;
    }

```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}


public class Cat extends Animal{
    //因为猫类中没有独有的属性。
    //所以此时不需要写私有的成员变量

    //空参
    public Cat() {
    }

    //需要带子类和父类中所有的属性
    public Cat(String name, int age, String color) {
        super(name,age,color);
    }
}


public class Dog extends Animal{
    //Dog ： 吼叫
    private String wang;

    //构造
    public Dog() {
    }
```

```
67
68      //带参构造：带子类加父类所有的属性
69      public Dog(String name, int age, String color,String wang) {
70          //共性的属性交给父类赋值
71          super(name,age,color);
72          //独有的属性自己赋值
73          this.wang = wang;
74      }
75
76      public String getWang() {
77          return wang;
78      }
79
80      public void setWang(String wang) {
81          this.wang = wang;
82      }
83  }
84
85  public class Demo {
86      public static void main(String[] args) {
87          //Animal： 姓名，年龄，颜色
88          //Cat :
89          //Dog： 吼叫
90
91          //创建狗的对象
92          Dog d = new Dog("旺财",2,"黑色","嗷呜~~");
93          System.out.println(d.getName()+", " + d.getAge() + ", " + d.getColor() + ", " +
    d.getWang());
94
95          //创建猫的对象
96          Cat c = new Cat("中华田园猫",3,"黄色");
97          System.out.println(c.getName() + ", " + c.getAge() + ", " + c.getColor());
98      }
99  }
100
101
```

多态

4.1 多态的形式

多态是继封装、继承之后，面向对象的第三大特性。

多态是出现在继承或者实现关系中的。

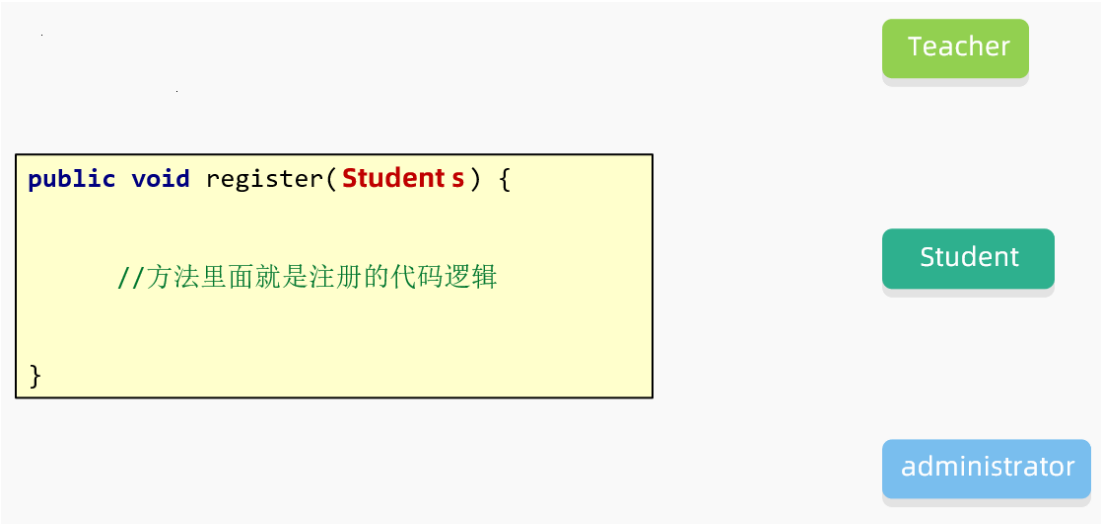
多态体现的格式：

```
父类类型 变量名 = new 子类/实现类构造器;
变量名.方法名();
```

多态的前提：有继承关系，子类对象是可以赋值给父类类型的变量。例如Animal是一个动物类型，而Cat是一个猫类型。Cat继承了Animal，Cat对象也是Animal类型，自然可以赋值给父类类型的变量。

4.2 多态的使用场景

如果没有多态，在下图中register方法只能传递学生对象，其他的Teacher和administrator对象是无法传递给register方法方法的，在这种情况下，只能定义三个不同的register方法分别接收学生，老师和管理员。



有了多态之后，方法的形参就可以定义为共同的父类Person。

要注意的是：

- 当一个方法的形参是一个类，我们可以传递这个类所有的子类对象。
- 当一个方法的形参是一个接口，我们可以传递这个接口所有的实现类对象（后面会学）。
- 而且多态还可以根据传递的不同对象来调用不同类中的方法。

代码示例：

```
父类：
public class Person {
    private String name;
    private int age;

    空参构造
    带全部参数的构造
    get和set方法

    public void show(){
        System.out.println(name + ", " + age);
    }
}

子类1：
public class Administrator extends Person {
    @Override
    public void show() {
        System.out.println("管理员的信息为：" + getName() + ", " + getAge());
    }
}

子类2：
public class Student extends Person{

    @Override
    public void show() {
        System.out.println("学生的信息为：" + getName() + ", " + getAge());
    }
}

子类3：
public class Teacher extends Person{

```

```
    @Override
    public void show() {
        System.out.println("老师的信息为：" + getName() + ", " + getAge());
    }
}

测试类：
public class Test {
    public static void main(String[] args) {
        //创建三个对象，并调用register方法

        Student s = new Student();
        s.setName("张三");
        s.setAge(18);


        Teacher t = new Teacher();
        t.setName("王建国");
        t.setAge(30);

        Administrator admin = new Administrator();
        admin.setName("管理员");
        admin.setAge(35);



        register(s);
        register(t);
        register(admin);


    }



    //这个方法既能接收老师，又能接收学生，还能接收管理员
    //只能把参数写成这三个类型的父类
    public static void register(Person p){
        p.show();
    }
}
```

4.3 多态的定义和前提

多态： 是指同一行为，具有多个不同表现形式。

从上面案例可以看出，Cat和Dog都是动物，都是吃这一行为，但是出现的效果（表现形式）是不一样的。

前提【重点】

1. 有继承或者实现关系
2. 方法的重写【意义体现：不重写，无意义】
3. 父类引用指向子类对象【格式体现】

> 父类类型：指子类对象继承的父类类型，或者实现的父接口类型。

4.4 多态的运行特点

调用成员变量时：编译看左边，运行看左边

调用成员方法时：编译看左边，运行看右边

代码示例：

```
Fu f = new Zi();
//编译看左边的父类中有没有name这个属性，没有就报错
//在实际运行的时候，把父类name属性的值打印出来
System.out.println(f.name);
//编译看左边的父类中有没有show这个方法，没有就报错
//在实际运行的时候，运行的是子类中的show方法
f.show();
```

4.5 多态的弊端

我们已经知道多态编译阶段是看左边父类类型的，如果子类有些独有的功能，此时**多态的写法就无法访问子类独有功能了。**

```
class Animal{
    public  void eat()  {
        System.out.println("动物吃东西！ ")
    }
}
class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
```

```
10
11      public void catchMouse() {
12          System.out.println("抓老鼠");
13      }
14  }
15
16  class Dog extends Animal {
17      public void eat() {
18          System.out.println("吃骨头");
19      }
20  }
21
22  class Test{
23      public static void main(String[] args){
24          Animal a = new Cat();
25          a.eat();
26          a.catchMouse();//编译报错，编译看左边，Animal没有这个方法
27      }
28  }
```

4.6 引用类型转换

4.6.1 为什么要转型

多态的写法就无法访问子类独有功能了。

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误。也就是说，**不能调用**子类拥有，而父类没有的方法。编译都错误，更别说运行了。这也是多态给我们带来的一点"小麻烦"。所以，想要调用子类特有的方法，必须做向下转型。

回顾基本数据类型转换

- 自动转换: 范围小的赋值给范围大的.自动完成:double d = 5;
- 强制转换: 范围大的赋值给范围小的,强制转换:int i = (int)3.14

多态的转型分为向上转型（自动转换）与向下转型（强制转换）两种。

4.6.2 向上转型（自动转换）

- **向上转型**：多态本身是子类类型向父类类型向上转换（自动转换）的过程，这个过程是默认的。
  当父类引用指向一个子类对象时，便是向上转型。
  使用格式：

```
1  父类类型  变量名 = new 子类类型();
2  如：Animal a = new Cat();
```

原因是：父类类型相对与子类来说是大范围的类型，Animal是动物类，是父类类型。Cat是猫类，是子类类型。Animal类型的范围当然很大，包含一切动物。所以子类范围小可以直接自动转型给父类类型的变量。

4.6.3 向下转型（强制转换）

- **向下转型**：父类类型向子类类型向下转换的过程，这个过程是强制的。
  一个已经向上转型的子类对象，将父类引用转为子类引用，可以使用强制类型转换的格式，便是向下转型。

使用格式：

```
子类类型 变量名 = (子类类型) 父类变量名;
如:Aniaml a = new Cat();
  Cat c =(Cat) a;
```

4.6.4 案例演示

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误。也就是说，**不能调用**子类拥有，而父类没有的方法。编译都错误，更别说运行了。这也是多态给我们带来的一点"小麻烦"。所以，想要调用子类特有的方法，必须做向下转型。

转型演示，代码如下：

定义类：

```
abstract class Animal {
    abstract void eat();
}

class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
    public void catchMouse() {
        System.out.println("抓老鼠");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
    public void watchHouse() {
        System.out.println("看家");
    }
}
```

定义测试类：

```
public class Test {
    public static void main(String[] args) {
        // 向上转型
        Animal a = new Cat();
        a.eat();               // 调用的是 Cat 的 eat

        // 向下转型
        Cat c = (Cat)a;
        c.catchMouse();        // 调用的是 Cat 的 catchMouse
    }
}
```

4.6.5 转型的异常

转型的过程中，一不小心就会遇到这样的问题，请看如下代码：

```
public class Test {
    public static void main(String[] args) {
        // 向上转型
        Animal a = new Cat();
        a.eat();               // 调用的是 Cat 的 eat

        // 向下转型
        Dog d = (Dog)a;
        d.watchHouse();        // 调用的是 Dog 的 watchHouse 【运行报错】
    }
}
```

这段代码可以通过编译，但是运行时，却报出了 **ClassCastException** ，类型转换异常！这是因为，明明创建了Cat类型对象，运行时，当然不能转换成Dog对象的。

4.6.6 instanceof关键字

为了避免ClassCastException的发生，Java提供了 **instanceof** 关键字，给引用变量做类型的校验，格式如下：

```
变量名 instanceof 数据类型
如果变量属于该数据类型或者其子类类型，返回true。
如果变量不属于该数据类型或者其子类类型，返回false。
```

所以，转换前，我们最好先做一个判断，代码如下：

```
public class Test {
    public static void main(String[] args) {
        // 向上转型
        Animal a = new Cat();
        a.eat();               // 调用的是 Cat 的 eat
```

```
        // 向下转型
        if (a instanceof Cat){
            Cat c = (Cat)a;
            c.catchMouse();        // 调用的是 Cat 的 catchMouse
        } else if (a instanceof Dog){
            Dog d = (Dog)a;
            d.watchHouse();        // 调用的是 Dog 的 watchHouse
        }
    }
}
```

4.6.7 instanceof新特性

JDK14的时候提出了新特性，把判断和强转合并成了一行

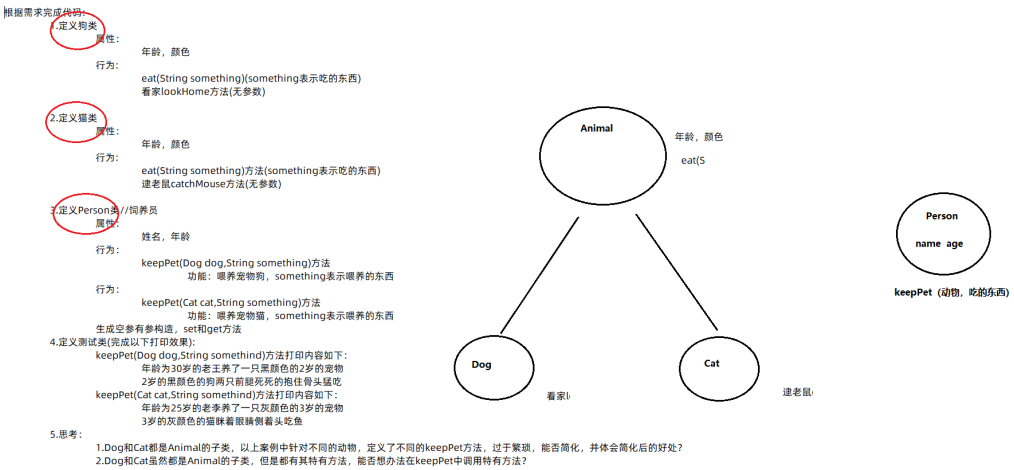
```
//新特性
//先判断a是否为Dog类型，如果是，则强转成Dog类型，转换之后变量名为d
//如果不是，则不强转，结果直接是false
if(a instanceof Dog d){
    d.lookHome();
}else if(a instanceof Cat c){
    c.catchMouse();
}else{
    System.out.println("没有这个类型，无法转换");
}
```

4.7 综合练习

```
需求：根据需求完成代码:
    1.定义狗类
        属性:
            年龄，颜色
        行为:
            eat(String something)(something表示吃的东西)
            看家lookHome方法(无参数)
2.定义猫类
    属性:
        年龄，颜色
    行为:
        eat(String something)方法(something表示吃的东西)
        逮老鼠catchMouse方法(无参数)
3.定义Person类//饲养员
    属性:
        姓名，年龄
    行为:
        keepPet(Dog dog,String something)方法
            功能：喂养宠物狗，something表示喂养的东西
```

行为：
keepPet(Cat cat,String something)方法
功能：喂养宠物猫，something表示喂养的东西
生成空参有参构造，set和get方法
4.定义测试类(完成以下打印效果)：
keepPet(Dog dog,String somethind)方法打印内容如下：
年龄为30岁的老王养了一只黑颜色的2岁的狗
2岁的黑颜色的狗两只前腿死死的抱住骨头猛吃
keepPet(Cat cat,String somethind)方法打印内容如下：
年龄为25岁的老李养了一只灰颜色的3岁的猫
3岁的灰颜色的猫眯着眼睛侧着头吃鱼
5.思考：
1.Dog和Cat都是Animal的子类，以上案例中针对不同的动物，定义了不同的keepPet方法，过于繁琐，能否简化，并体会简化后的好处？
2.Dog和Cat虽然都是Animal的子类，但是都有其特有方法，能否想办法在keepPet中调用特有方法？

画图分析：



代码示例：

```java
//动物类（父类）
public class Animal {
    private int age;
    private String color;


    public Animal() {
    }

    public Animal(int age, String color) {
        this.age = age;
        this.color = color;
    }

```

```
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public void eat(String something){
        System.out.println("动物在吃" + something);
    }
}

//猫类（子类）
public class Cat extends Animal {

    public Cat() {
    }

    public Cat(int age, String color) {
        super(age, color);
    }

    @Override
    public void eat(String something) {
        System.out.println(getAge() + "岁的" + getColor() + "颜色的猫眯着眼睛侧着头
吃" + something);
    }

    public void catchMouse(){
        System.out.println("猫抓老鼠");
    }

}

//狗类（子类）
public class Dog extends Animal {
    public Dog() {
    }

```

```
    public Dog(int age, String color) {
        super(age, color);
    }

    //行为
    //eat(String something)(something表示吃的东西)
    //看家lookHome方法(无参数)
    @Override
    public void eat(String something) {
        System.out.println(getAge() + "岁的" + getColor() + "颜色的狗两只前腿死死的抱住" + something + "猛吃");
    }

    public void lookHome(){
        System.out.println("狗在看家");
    }
}


//饲养员类
public class Person {
    private String name;
    private int age;

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

```

```
    //饲养狗
    /* public void keepPet(Dog dog, String something) {
        System.out.println("年龄为" + age + "岁的" + name + "养了一只" + dog.getColor() + "颜色的" + dog.getAge() + "岁的狗");
        dog.eat(something);
    }

    //饲养猫
    public void keepPet(Cat cat, String something) {
        System.out.println("年龄为" + age + "岁的" + name + "养了一只" + cat.getColor() + "颜色的" + cat.getAge() + "岁的猫");
        cat.eat(something);
    }*/


    //想要一个方法，能接收所有的动物，包括猫，包括狗
    //方法的形参：可以写这些类的父类 Animal
    public void keepPet(Animal a, String something) {
        if(a instanceof Dog d){
            System.out.println("年龄为" + age + "岁的" + name + "养了一只" + a.getColor() + "颜色的" + a.getAge() + "岁的狗");
            d.eat(something);
        }else if(a instanceof Cat c){
            System.out.println("年龄为" + age + "岁的" + name + "养了一只" + c.getColor() + "颜色的" + c.getAge() + "岁的猫");
            c.eat(something);
        }else{
            System.out.println("没有这种动物");
        }
    }
}

//测试类
public class Test {
    public static void main(String[] args) {
        //创建对象并调用方法
        /* Person p1 = new Person("老王",30);
        Dog d = new Dog(2,"黑");
        p1.keepPet(d,"骨头");


        Person p2 = new Person("老李",25);
        Cat c = new Cat(3,"灰");
        p2.keepPet(c,"鱼");*/


        //创建饲养员的对象
        Person p = new Person("老王",30);
```

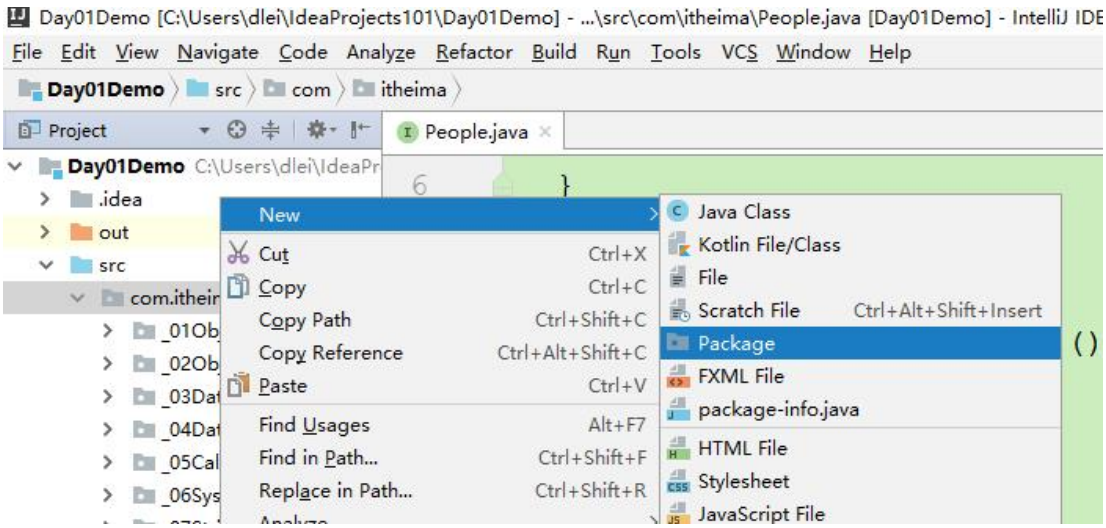
```
153         Dog d = new Dog(2,"黑");
154         Cat c = new Cat(3,"灰");
155         p.keepPet(d,"骨头");
156         p.keepPet(c,"鱼");
157 
158     }
159 }
```

包

5.1 包

包在操作系统中其实就是一个文件夹。**包是用来分门别类的管理技术，不同的技术类放在不同的包下**，方便管理和维护。

在IDEA项目中，建包的操作如下：



包名的命名规范：

```
路径名.路径名.xxx.xxx
// 例如：com.itheima.oa
```

- 包名一般是公司域名的倒写。例如：黑马是 www.itheima.com ,包名就可以定义成com.itheima.技术名称。
- 包名必须用”．“连接。
- 包名的每个路径名必须是一个合法的标识符，而且不能是Java的关键字。

5.2 导包

什么时候需要导包?

情况一：在使用Java中提供的非核心包中的类时

情况二：使用自己写的其他包中的类时

什么时候不需要导包?

情况一：在使用Java核心包（java.lang）中的类时

情况二：在使用自己写的同一个包中的类时

5.3 使用不同包下的相同类怎么办?

假设demo1和demo2中都有一个Student该如何使用?

代码示例：

```
//使用全类名的形式即可。
//全类名：包名 + 类名
//拷贝全类名的快捷键：选中类名crtl + shift + alt + c 或者用鼠标点copy，再点击copy Reference
com.itheima.homework.demo1.Student s1 = new com.itheima.homework.demo1.Student();
com.itheima.homework.demo2.Student s2 = new com.itheima.homework.demo2.Student();
```

权限修饰符

6.1 权限修饰符

在Java中提供了四种访问权限，使用不同的访问权限修饰符修饰时，被修饰的内容会有不同的访问权限，我们之前已经学习过了public 和 private，接下来我们研究一下protected和默认修饰符的作用。

- public：公共的，所有地方都可以访问。
- protected：本类 ，本包，其他包中的子类都可以访问。
- 默认（没有修饰符）：本类 ，本包可以访问。

  注意：默认是空着不写，不是default
- private：私有的，当前类可以访问。

public > protected > 默认 > private

6.2 不同权限的访问能力

	public	protected	默认	private
同一类中	√	√	√	√
同一包中的类	√	√	√	
不同包的子类	√	√		
不同包中的无关类	√			

可见，public具有最大权限。private则是最小权限。

编写代码时，如果没有特殊的考虑，建议这样使用权限：

- 成员变量使用 **private** ，隐藏细节。
- 构造方法使用 **public** ，方便创建对象。
- 成员方法使用 **public** ，方便调用方法。

> 小贴士：不加权限修饰符，就是默认权限

final关键字

7.1 概述

学习了继承后，我们知道，子类可以在父类的基础上改写父类内容，比如，方法重写。

如果有一个方法我不想别人去改写里面内容，该怎么办呢？

Java提供了 **final** 关键字，表示修饰的内容不可变。

- **final**： 不可改变，最终的含义。可以用于修饰类、方法和变量。
  - 类：被修饰的类，**不能被继承。**
  - 方法：被修饰的方法，不能被重写。
  - 变量：被修饰的变量，有且仅能被赋值一次。

7.2 使用方式

7.2.1 修饰类

final修饰的类，不能被继承。

格式如下：

```
final class 类名 {
}
```

代码:

```
final class Fu {
}
// class Zi extends Fu {} // 报错,不能继承final的类
```

查询API发现像 **public final class String** 、 **public final class Math** 、 **public final class Scanner** 等，很多我们学习过的类，都是被final修饰的，目的就是供我们使用，而不让我们所以改变其内容。

7.2.2 修饰方法

final修饰的方法，不能被重写。
格式如下：

```
修饰符 final 返回值类型 方法名(参数列表){
    //方法体
}
```

代码:

```
class Fu2 {
    final public void show1() {
        System.out.println("Fu2 show1");
    }
    public void show2() {
        System.out.println("Fu2 show2");
    }
}

class Zi2 extends Fu2 {
//  @Override
//  public void show1() {
//      System.out.println("Zi2 show1");
//  }
    @Override
    public void show2() {
```

```
        System.out.println("Zi2 show2");
    }
}
```

7.2.3 修饰变量-局部变量

1. **局部变量——基本类型**
   基本类型的局部变量，被final修饰后，只能赋值一次，不能再更改。代码如下：

```
public class FinalDemo1 {
    public static void main(String[] args) {
        // 声明变量，使用final修饰
        final int a;
        // 第一次赋值
        a = 10;
        // 第二次赋值
        a = 20; // 报错,不可重新赋值

        // 声明变量，直接赋值，使用final修饰
        final int b = 10;
        // 第二次赋值
        b = 20; // 报错,不可重新赋值
    }
}
```

思考，下面两种写法，哪种可以通过编译?

写法1：

```
final int c = 0;
for (int i = 0; i < 10; i++) {
    c = i;
    System.out.println(c);
}
```

写法2：

```
for (int i = 0; i < 10; i++) {
    final int c = i;
    System.out.println(c);
}
```

根据 **final** 的定义，写法1报错！写法2，为什么通过编译呢？因为每次循环，都是一次新的变量c。这也是大家需要注意的地方。

7.2.4 修饰变量-成员变量

成员变量涉及到初始化的问题，初始化方式有显示初始化和构造方法初始化，只能选择其中一个：

- 显示初始化(在定义成员变量的时候立马赋值)（常用）；

```
public class Student {
    final int num = 10;
}
```

- 构造方法初始化(在构造方法中赋值一次)（不常用，了解即可）。

  注意：每个构造方法中都要赋值一次！

```
public class Student {
    final int num = 10;
    final int num2;

    public Student() {
        this.num2 = 20;
//      this.num2 = 20;
    }

     public Student(String name) {
        this.num2 = 20;
//      this.num2 = 20;
    }
}
```

> 被final修饰的常量名称，一般都有书写规范，所有字母都**大写**。

抽象类

8.1 概述

8.1.1 抽象类引入

父类中的方法，被它的子类们重写，子类各自的实现都不尽相同。那么父类的方法声明和方法主体，只有声明还有意义，而方法主体则没有存在的意义了(因为子类对象会调用自己重写的方法)。换句话说，父类可能知道子类应该有哪个功能，但是功能具体怎么实现父类是不清楚的（由子类自己决定），父类只需要提供一个没有方法体的定义即可，具体实现交给子类自己去实现。**我们把没有方法体的方法称为抽象方法。Java语法规定，包含抽象方法的类就是抽象类。**

- **抽象方法**： 没有方法体的方法。
- **抽象类**：包含抽象方法的类。

8.2 abstract使用格式

abstract是抽象的意思，用于修饰方法方法和类，修饰的方法是抽象方法，修饰的类是抽象类。

8.2.1 抽象方法

使用 **abstract** 关键字修饰方法，该方法就成了抽象方法，抽象方法只包含一个方法名，而没有方法体。

定义格式：

```
修饰符 abstract 返回值类型 方法名 (参数列表);
```

代码举例：

```
public abstract void run();
```

8.2.2 抽象类

如果一个类包含抽象方法，那么该类必须是抽象类。**注意：抽象类不一定有抽象方法，但是有抽象方法的类必须定义成抽象类。**

定义格式：

```
abstract class 类名字 {

}
```

代码举例：

```
public abstract class Animal {
    public abstract void run();
}
```

8.2.3 抽象类的使用

要求：继承抽象类的子类**必须重写父类所有的抽象方法。**否则，该子类也必须声明为抽象类。

代码举例：

```
// 父类,抽象类
abstract class Employee {
    private String id;
    private String name;
    private double salary;

    public Employee() {
    }

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

    // 抽象方法
    // 抽象方法必须要放在抽象类中
    abstract public void work();
}

// 定义一个子类继承抽象类
class Manager extends Employee {
    public Manager() {
    }
    public Manager(String id, String name, double salary) {
        super(id, name, salary);
    }
    // 2.重写父类的抽象方法
    @Override
    public void work() {
        System.out.println("管理其他人");
    }
}

// 定义一个子类继承抽象类
class Cook extends Employee {
    public Cook() {
    }
    public Cook(String id, String name, double salary) {
        super(id, name, salary);
    }
```

```
    @Override
    public void work() {
        System.out.println("厨师炒菜多加点盐...");
    }
}

// 测试类
public class Demo10 {
    public static void main(String[] args) {
        // 创建抽象类,抽象类不能创建对象
        // 假设抽象类让我们创建对象,里面的抽象方法没有方法体,无法执行.所以不让我们创建对象
//      Employee e = new Employee();
//      e.work();

        // 3.创建子类
        Manager m = new Manager();
        m.work();

        Cook c = new Cook("ap002", "库克", 1);
        c.work();
    }
}
```

此时的方法重写，是子类对父类抽象方法的完成实现，我们将这种方法重写的操作，也叫做**实现方法。**

8.3 抽象类的特征

抽象类的特征总结起来可以说是 **有得有失**

有得：抽象类得到了拥有抽象方法的能力。

有失：抽象类失去了创建对象的能力。

其他成员（构造方法，实例方法，静态方法等）抽象类都是具备的。

8.4 抽象类的细节

关于抽象类的使用，以下为语法上要注意的细节，虽然条目较多，但若理解了抽象的本质，无需死记硬背。

1. 抽象类**不能创建对象**，如果创建，编译无法通过而报错。只能创建其非抽象子类的对象。

> 理解：假设创建了抽象类的对象，调用抽象的方法，而抽象方法没有具体的方法体，没有意义。

2. 抽象类中，可以有构造方法，是供子类创建对象时，初始化父类成员使用的。

> 理解：子类的构造方法中，有默认的super()，需要访问父类构造方法。

3. 抽象类中，不一定包含抽象方法，但是有抽象方法的类必定是抽象类。

> 理解：未包含抽象方法的抽象类，目的就是不想让调用者创建该类对象，通常用于某些特殊的类结构设计。

4. 抽象类的子类，必须重写抽象父类中**所有的**抽象方法，否则子类也必须定义成抽象类，编译无法通过而报错。

> 理解：假设不重写所有抽象方法，则类中可能包含抽象方法。那么创建对象后，调用抽象的方法，没有意义。

5. 抽象类存在的意义是为了被子类继承。

> 理解：抽象类中已经实现的是模板中确定的成员，抽象类不确定如何实现的定义成抽象方法，交给具体的子类去实现。

8.5 抽象类存在的意义

抽象类存在的意义是为了被子类继承，否则抽象类将毫无意义。抽象类可以强制让子类，一定要按照规定的格式进行重写。

接口

9.1 概述

我们已经学完了抽象类，抽象类中可以用抽象方法，也可以有普通方法，构造方法，成员变量等。那么什么是接口呢？**接口是更加彻底的抽象，JDK7之前，包括JDK7，接口中全部是抽象方法。接口同样是不能创建对象的。**

9.2 定义格式

```
//接口的定义格式：
interface 接口名称{
    // 抽象方法
}

// 接口的声明：interface
// 接口名称：首字母大写，满足“驼峰模式”
```

9.3 接口成分的特点

在JDK7，包括JDK7之前，接口中的**只有**包含：抽象方法和常量

9.3.1.抽象方法

注意：接口中的抽象方法默认会自动加上public abstract修饰程序员无需自己手写！！

按照规范：以后接口中的抽象方法建议不要写上public abstract。因为没有必要啊，默认会加上。

9.3.2 常量

在接口中定义的成员变量默认会加上： public static final修饰。也就是说在接口中定义的成员变量实际上是一个常量。这里是使用public static final修饰后，变量值就不可被修改，并且是静态化的变量可以直接用接口名访问，所以也叫常量。常量必须要给初始值。常量命名规范建议字母全部大写，多个单词用下划线连接。

9.3.3 案例演示

```
public interface InterF {
    // 抽象方法!
    //    public abstract void run();
    void run();

    //    public abstract String getName();
    String getName();

    //    public abstract int add(int a , int b);
    int add(int a , int b);


    // 它的最终写法是：
    // public static final int AGE = 12 ;
    int AGE  = 12; //常量
    String SCHOOL_NAME = "黑马程序员";

```

```
18  }
```

9.4 基本的实现

9.4.1 实现接口的概述

类与接口的关系为实现关系，即**类实现接口**，该类可以称为接口的实现类，也可以称为接口的子类。实现的动作类似继承，格式相仿，只是关键字不同，实现使用 **implements** 关键字。

9.4.2 实现接口的格式

```
1  /**接口的实现：
2     在Java中接口是被实现的，实现接口的类称为实现类。
3     实现类的格式:*/
4  class 类名 implements 接口1,接口2,接口3...{
5
6  }
```

从上面格式可以看出，接口是可以被多实现的。大家可以想一想为什么呢？

9.4.3 类实现接口的要求和意义

1. 必须重写实现的全部接口中所有抽象方法。
2. 如果一个类实现了接口，但是没有重写完全部接口的全部抽象方法，这个类也必须定义成抽象类。
3. **意义：接口体现的是一种规范，接口对实现类是一种强制性的约束，要么全部完成接口申明的功能，要么自己也定义成抽象类。这正是一种强制性的规范。**

9.4.4 类与接口基本实现案例

假如我们定义一个运动员的**接口**（规范），代码如下：

```
1  /**
2    接口：接口体现的是规范。
3   * */
4  public interface SportMan {
5     void run(); // 抽象方法，跑步。
6     void law(); // 抽象方法，遵守法律。
7     String compittion(String project);  // 抽象方法，比赛。
8  }
```

接下来定义一个乒乓球运动员类，实现接口，实现接口的**实现类**代码如下：

```
1  package com.itheima._03接口的实现;
```

```java
/**
 * 接口的实现：
 *    在Java中接口是被实现的，实现接口的类称为实现类。
 *    实现类的格式:
 *      class 类名 implements 接口1,接口2,接口3...{
 *
 *
 *      }
 * */
public class PingPongMan  implements SportMan {
    @Override
    public void run() {
        System.out.println("乒乓球运动员稍微跑一下！！");
    }

    @Override
    public void law() {
        System.out.println("乒乓球运动员守法！");
    }

    @Override
    public String compittion(String project) {
        return "参加"+project+"得金牌！";
    }
}
```

测试代码：

```java
public class TestMain {
    public static void main(String[] args) {
        // 创建实现类对象。
        PingPongMan zjk = new PingPongMan();
        zjk.run();
        zjk.law();
        System.out.println(zjk.compittion("全球乒乓球比赛"));

    }
}
```

9.4.5 类与接口的多实现案例

类与接口之间的关系是多实现的，一个类可以同时实现多个接口。

首先我们先定义两个接口，代码如下：

```java
/** 法律规范：接口*/
public interface Law {
    void rule();
}

/** 这一个运动员的规范：接口*/
public interface SportMan {
    void run();
}

```

然后定义一个实现类：

```java
/**
 * Java中接口是可以被多实现的：
 *    一个类可以实现多个接口: Law, SportMan
 *
 * */
public class JumpMan implements Law ,SportMan {
    @Override
    public void rule() {
        System.out.println("尊长守法");
    }

    @Override
    public void run() {
        System.out.println("训练跑步！");
    }
}
```

从上面可以看出类与接口之间是可以多实现的，我们可以理解成实现多个规范，这是合理的。

9.5 接口与接口的多继承

Java中，接口与接口之间是可以多继承的：也就是一个接口可以同时继承多个接口。大家一定要注意：

类与接口是实现关系

接口与接口是继承关系

接口继承接口就是把其他接口的抽象方法与本接口进行了合并。

案例演示：

```java
public interface Abc {
    void go();
    void test();
```

```java
}

/** 法律规范：接口*/
public interface Law {
    void rule();
    void test();
}

 *
 * 总结:
 *    接口与类之间是多实现的。
 *    接口与接口之间是多继承的。
 * */
public interface SportMan extends Law , Abc {
    void run();
}
```

默认接口

- 允许在接口中定义默认方法，需要使用关键字 default 修饰

  作用：解决接口升级的问题

接口中**默认方法**的定义格式：

- 格式：public default 返回值类型 方法名(参数列表) { }
- 范例：public default void show() { }

接口中默认方法的**注意事项**：

- 默认方法不是抽象方法，所以不强制被重写。但是如果被重写，重写的时候去掉default关键字
- public可以省略，default不能省略
- 如果实现了多个接口，多个接口中存在相同名字的默认方法，子类就必须对该方法进行重写

静态接口

JDK8以后接口中新增的方法

- 允许在接口中定义定义静态方法，需要用static修饰

接口中**静态方法**的定义格式：

- 格式：public static 返回值类型 方法名(参数列表) { }
- 范例：public static void show() { }

接口中静态方法的**注意事项**：

- 静态方法只能通过接口名调用，不能通过实现类名或者对象名调用
- public可以省略，static不能省略

私有接口

接口中私有方法的定义格式：

- 格式1：private 返回值类型 方法名(参数列表) { }
- 范例1：private void show() { }

- 格式2：private static 返回值类型 方法名(参数列表) { }
- 范例2：private static void method() { }

适配器

1. 当一个接口中抽象方法过多，但是我只要使用其中一部分的

   时候，就可以适配器设计模式

2. 书写步骤：

   编写中间类XXXAdapter，实现对应的接口
   对接口中的抽象方法进行空实现
   让真正的实现类继承中间类，并重写需要用的方法
   为了避免其他类创建适配器类的对象，中间的适配器类
   用abstract进行修饰

9.6扩展：接口的细节

不需要背，只要当idea报错之后，知道如何修改即可。

关于接口的使用，以下为语法上要注意的细节，虽然条目较多，但若理解了抽象的本质，
无需死记硬背。

1. 当两个接口中存在相同抽象方法的时候，该怎么办?

> 只要重写一次即可。此时重写的方法，既表示重写1接口的，也表示重写2接口的。

2. 实现类能不能继承A类的时候，同时实现其他接口呢?

> 继承的父类，就好比是亲爸爸一样
> 实现的接口，就好比是干爹一样
> 可以继承一个类的同时，再实现多个接口，只不过，要把接口里面所有的抽象方法，全部实现。

3. 实现类能不能继承一个抽象类的时候，同时实现其他接口呢?

> 实现类可以继承一个抽象类的同时，再实现其他多个接口，只不过要把里面所有的抽象方法全部重写。

4. 实现类Zi，实现了一个接口，还继承了一个Fu类。假设在接口中有一个方法，父类中也有一个相同的方法。子类如何操作呢?

> 处理办法一：如果父类中的方法体，能满足当前业务的需求，在子类中可以不用重写。
> 处理办法二：如果父类中的方法体，不能满足当前业务的需求，需要在子类中重写。

5. 如果一个接口中，有10个抽象方法，但是我在实现类中，只需要用其中一个，该怎么办?

> 可以在接口跟实现类中间，新建一个中间类（适配器类）
> 让这个适配器类去实现接口，对接口里面的所有的方法做空重写。
> 让子类继承这个适配器类，想要用到哪个方法，就重写哪个方法。
> 因为中间类没有什么实际的意义，所以一般会把中间类定义为抽象的，不让外界创建对象

内部类（好像没学）

10.1 概述

10.1.1 什么是内部类

将一个类A定义在另一个类B里面，里面的那个类A就称为**内部类**，B则称为**外部类**。可以把内部类理解成寄生，外部类理解成宿主。

10.1.2 什么时候使用内部类

一个事物内部还有一个独立的事物，内部的事物脱离外部的事物无法独立使用

1. 人里面有一颗心脏。
2. 汽车内部有一个发动机。
3. 为了实现更好的封装性。

10.2 内部类的分类

按定义的位置来分

1. **成员内部内**，类定义在了成员位置 (类中方法外称为成员位置，无static修饰的内部类)
2. **静态内部类**，类定义在了成员位置 (类中方法外称为成员位置，有static修饰的内部类)
3. **局部内部类**，类定义在方法内
4. **匿名内部类**，没有名字的内部类，可以在方法中，也可以在类中方法外。

10.3 成员内部类

成员内部类特点：

- 无static修饰的内部类，属于外部类对象的。
- 宿主：外部类对象。

内部类的使用格式：

```
外部类.内部类。 // 访问内部类的类型都是用 外部类.内部类
```

获取成员内部类对象的两种方式：

方式一：外部直接创建成员内部类的对象

```
外部类.内部类 变量 = new 外部类（）.new 内部类（）；
```

方式二：在外部类中定义一个方法提供内部类的对象

案例演示

```
方式一：
public class Test {
    public static void main(String[] args) {
        //  宿主：外部类对象。
       // Outer out = new Outer();
        // 创建内部类对象。
        Outer.Inner oi = new Outer().new Inner();
        oi.method();
    }
}

class Outer {
    // 成员内部类，属于外部类对象的。
    // 拓展：成员内部类不能定义静态成员。
    public class Inner{
        // 这里面的东西与类是完全一样的。
        public void method(){
            System.out.println("内部类中的方法被调用了");
        }
    }
}


方式二：
public class Outer {
    String name;
    private class Inner{
        static int a = 10;
```

```
    }
    public Inner getInstance(){
        return new Inner();
    }
}

public class Test {
    public static void main(String[] args) {
        Outer o = new Outer();
        System.out.println(o.getInstance());


    }
}
```

10.4 成员内部类的细节

编写成员内部类的注意点：

1. 成员内部类可以被一些修饰符所修饰，比如： private，默认，protected，public，static等
2. 在成员内部类里面，JDK16之前不能定义静态变量，JDK16开始才可以定义静态变量。
3. 创建内部类对象时，对象中有一个隐含的Outer.this记录外部类对象的地址值。（请参见3.6节的内存图）

详解：

内部类被private修饰，外界无法直接获取内部类的对象，只能通过3.3节中的方式二获取内部类的对象

被其他权限修饰符修饰的内部类一般用3.3节中的方式一直接获取内部类的对象

内部类被static修饰是成员内部类中的特殊情况，叫做静态内部类下面单独学习。

内部类如果想要访问外部类的成员变量，外部类的变量必须用final修饰，JDK8以前必须手动写final，JDK8之后不需要手动写，JDK默认加上。

10.5 成员内部类面试题

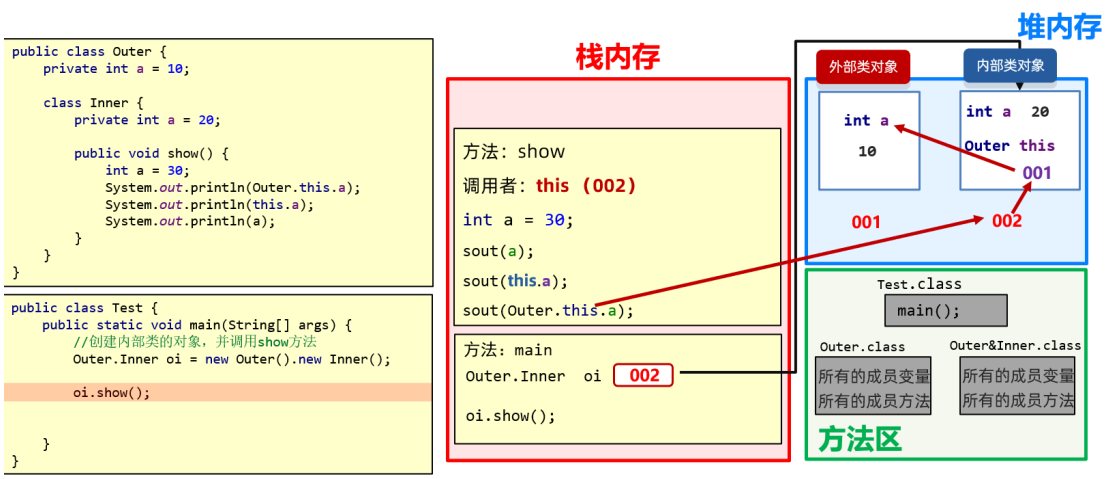
请在?地方向上相应代码,以达到输出的内容

注意：内部类访问外部类对象的格式是：**外部类名.this**

```
public class Test {
    public static void main(String[] args) {
        Outer.inner oi = new Outer().new inner();
        oi.method();
    }
```

```
}

class Outer {  // 外部类
    private int a = 30;

    // 在成员位置定义一个类
    class inner {
        private int a = 20;

        public void method() {
            int a = 10;
            System.out.println(???);   // 10  答案：a
            System.out.println(???);   // 20  答案：this.a
            System.out.println(???);   // 30  答案：Outer.this.a
        }
    }
}
```

10.6 成员内部类内存图



10.7 静态内部类

静态内部类特点：

- 静态内部类是一种特殊的成员内部类。
- 有static修饰，属于外部类本身的。
- 总结：静态内部类与其他类的用法完全一样。只是访问的时候需要加上外部类.内部类。
- **拓展1**:静态内部类可以直接访问外部类的静态成员。
- **拓展2**:静态内部类不可以直接访问外部类的非静态成员，如果要访问需要创建外部类的对象。
- **拓展3**:静态内部类中没有银行的Outer.this。

内部类的使用格式：

```
外部类.内部类。
```

静态内部类对象的创建格式：

```
外部类.内部类  变量 = new  外部类.内部类构造器;
```

调用方法的格式：

- 调用非静态方法的格式：先创建对象，用对象调用
- 调用静态方法的格式：外部类名.内部类名.方法名();

案例演示：

```
// 外部类：Outer01
class Outer01{
    private static  String sc_name = "黑马程序";
    // 内部类: Inner01
    public static class Inner01{
        // 这里面的东西与类是完全一样的。
        private String name;
        public Inner01(String name) {
            this.name = name;
        }
        public void showName(){
            System.out.println(this.name);
            // 拓展:静态内部类可以直接访问外部类的静态成员。
            System.out.println(sc_name);
        }
    }
}

public class InnerClassDemo01 {
    public static void main(String[] args) {
        // 创建静态内部类对象。
        // 外部类.内部类  变量 = new  外部类.内部类构造器;
        Outer01.Inner01 in  = new Outer01.Inner01("张三");
        in.showName();
    }
}
```

10.8 局部内部类

- **局部内部类** ： 定义在**方法中**的类。

定义格式:

```
class 外部类名 {
    数据类型 变量名;

    修饰符 返回值类型 方法名(参数列表) {
        // ...
        class 内部类 {
            // 成员变量
            // 成员方法
        }
    }
}
```

10.9 匿名内部类【重点】

10.9.1 概述

匿名内部类： 是内部类的简化写法。他是一个隐含了名字的内部类。开发中，最常用到的内部类就是匿名内部类了。

10.9.2 格式

```
new 类名或者接口名() {
    重写方法;
};
```

包含了：

- 继承或者实现关系
- 方法重写
- 创建对象

所以从语法上来讲，这个整体其实是匿名内部类对象

10.9.2 什么时候用到匿名内部类

实际上，如果我们希望定义一个只要使用一次的类，就可考虑使用匿名内部类。匿名内部类的本质作用

是为了简化代码。

之前我们使用接口时，似乎得做如下几步操作：

1. 定义子类
2. 重写接口中的方法
3. 创建子类对象

4. 调用重写后的方法

```
interface Swim {
    public abstract void swimming();
}

// 1. 定义接口的实现类
class Student implements Swim {
    // 2. 重写抽象方法
    @Override
    public void swimming() {
        System.out.println("狗刨式...");
    }
}

public class Test {
    public static void main(String[] args) {
        // 3. 创建实现类对象
        Student s = new Student();
        // 4. 调用方法
        s.swimming();
    }
}
```

我们的目的，最终只是为了调用方法，那么能不能简化一下，把以上四步合成一步呢？匿名内部类就是做这样的快捷方式。

10.9.3 匿名内部类前提和格式

匿名内部类必须**继承一个父类**或者**实现一个父接口。**

匿名内部类格式

```
new 父类名或者接口名(){
    // 方法重写
    @Override
    public void method() {
        // 执行语句
    }
};
```

10.9.4 使用方式

以接口为例，匿名内部类的使用，代码如下：

```
interface Swim {
    public abstract void swimming();
}
```

```
4
5  public class Demo07 {
6      public static void main(String[] args) {
7          // 使用匿名内部类
8          new Swim() {
9              @Override
10             public void swimming() {
11                 System.out.println("自由泳...");
12             }
13         }.swimming();
14
15         // 接口 变量 = new 实现类(); // 多态,走子类的重写方法
16         Swim s2 = new Swim() {
17             @Override
18             public void swimming() {
19                 System.out.println("蛙泳...");
20             }
21         };
22
23         s2.swimming();
24         s2.swimming();
25     }
26 }
```

10.9.5 匿名内部类的特点

1. 定义一个没有名字的内部类
2. 这个类实现了父类，或者父类接口
3. 匿名内部类会创建这个没有名字的类的对象

10.9.6 匿名内部类的使用场景

通常在方法的形式参数是接口或者抽象类时，也可以将匿名内部类作为参数传递。代码如下：

```
1  interface Swim {
2      public abstract void swimming();
3  }
4
5  public class Demo07 {
6      public static void main(String[] args) {
7          // 普通方式传入对象
8          // 创建实现类对象
9          Student s = new Student();
10
11         goSwimming(s);
12         // 匿名内部类使用场景:作为方法参数传递
```

```
        Swim s3 = new Swim() {
            @Override
            public void swimming() {
                System.out.println("蝶泳...");
            }
        };
        // 传入匿名内部类
        goSwimming(s3);

        // 完美方案: 一步到位
        goSwimming(new Swim() {
            public void swimming() {
                System.out.println("大学生, 蛙泳...");
            }
        });

        goSwimming(new Swim() {
            public void swimming() {
                System.out.println("小学生, 自由泳...");
            }
        });
    }

    // 定义一个方法,模拟请一些人去游泳
    public static void goSwimming(Swim s) {
        s.swimming();
    }
}
```

Date类

11.1 Date概述

java.util.Date`类 表示特定的瞬间，精确到毫秒。

继续查阅Date类的描述，发现Date拥有多个构造函数，只是部分已经过时，我们重点看以下两个构造函数

- **public Date()**：从运行程序的此时此刻到时间原点经历的毫秒值,转换成Date对象，分配Date对象并初始化此对象，以表示分配它的时间（精确到毫秒）。
- **public Date(long date)**：将指定参数的毫秒值date,转换成Date对象，分配Date对象并初始化此对象，以表示自从标准基准时间（称为“历元（epoch）”，即1970年1月1日00:00:00 GMT）以来的指定毫秒数。

> tips: 由于中国处于东八区（GMT+08:00）是比世界协调时间/格林尼治时间（GMT）快8小时的时区，当格林尼治标准时间为0:00时，东八区的标准时间为08:00。

简单来说：使用无参构造，可以自动设置当前系统时间的毫秒时刻；指定long类型的构造参数，可以自定义毫秒时刻。例如：

```
import java.util.Date;

public class Demo01Date {
    public static void main(String[] args) {
        // 创建日期对象，把当前的时间
        System.out.println(new Date()); // Tue Jan 16 14:37:35 CST 2020
        // 创建日期对象，把当前的毫秒值转成日期对象
        System.out.println(new Date(0L)); // Thu Jan 01 08:00:00 CST 1970
    }
}
```

> tips:在使用println方法时，会自动调用Date类中的toString方法。Date类对Object类中的toString方法进行了覆盖重写，所以结果为指定格式的字符串。

11.2 Date常用方法

Date类中的多数方法已经过时，常用的方法有：

- **`public long getTime()`** 把日期对象转换成对应的时间毫秒值。
- **`public void setTime(long time)`** 把方法参数给定的毫秒值设置给日期对象

示例代码

```
public class DateDemo02 {
    public static void main(String[] args) {
        //创建日期对象
        Date d = new Date();

        //public long getTime():获取的是日期对象从1970年1月1日 00:00:00到现在的毫秒值
        //System.out.println(d.getTime());
        //System.out.println(d.getTime() * 1.0 / 1000 / 60 / 60 / 24 / 365 + "年");

        //public void setTime(long time):设置时间，给的是毫秒值
        //long time = 1000*60*60;
        long time = System.currentTimeMillis();
        d.setTime(time);

        System.out.println(d);
```

```
    }
}
```

> 小结：Date表示特定的时间瞬间，我们可以使用Date对象对时间进行操作。

第二章 SimpleDateFormat类（好像没学）

`java.text.SimpleDateFormat` 是日期/时间格式化类，我们通过这个类可以帮我们完成日期和文本之间的转换,也就是可以在Date对象与String对象之间进行来回转换。

- **格式化**：按照指定的格式，把Date对象转换为String对象。
- **解析**：按照指定的格式，把String对象转换为Date对象。

12.1 构造方法

由于DateFormat为抽象类，不能直接使用，所以需要常用的子类 `java.text.SimpleDateFormat` 。这个类需要一个模式（格式）来指定格式化或解析的标准。构造方法为：

- `public SimpleDateFormat(String pattern)` ：用给定的模式和默认语言环境的日期格式符号构造SimpleDateFormat。参数pattern是一个字符串，代表日期时间的自定义格式。

12.2 格式规则

常用的格式规则为：

标识字母（区分大小写）	含义
y	年
M	月
d	日
H	时
m	分
s	秒

> 备注：更详细的格式规则，可以参考SimpleDateFormat类的API文档。

12.3 常用方法

DateFormat类的常用方法有：

- **public String format(Date date)** ： 将Date对象格式化为字符串。
- **public Date parse(String source)** ： 将字符串解析为Date对象。

```java
package com.itheima.a01jdk7datedemo;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class A03_SimpleDateFormatDemo1 {
    public static void main(String[] args) throws ParseException {
        /*
            public simpleDateFormat() 默认格式
            public simpleDateFormat(String pattern) 指定格式
            public final string format(Date date) 格式化(日期对象 ->字符串)
            public Date parse(string source) 解析(字符串 ->日期对象)
        */

        //1.定义一个字符串表示时间
        String str = "2023-11-11 11:11:11";
        //2.利用空参构造创建simpleDateFormat对象
        // 细节:
        //创建对象的格式要跟字符串的格式完全一致
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date date = sdf.parse(str);
        //3.打印结果
        System.out.println(date.getTime());//1699672271000


    }

    private static void method1() {
        //1.利用空参构造创建simpleDateFormat对象，默认格式
        SimpleDateFormat sdf1 = new SimpleDateFormat();
        Date d1 = new Date(0L);
        String str1 = sdf1.format(d1);
        System.out.println(str1);//1970/1/1 上午8:00

        //2.利用带参构造创建simpleDateFormat对象，指定格式
        SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy年MM月dd日HH:mm:ss");
        String str2 = sdf2.format(d1);
        System.out.println(str2);//1970年01月01日 08:00:00
```

```
40
41          //课堂练习:yyyy年MM月dd日 时:分:秒 星期
42      }
43  }
44
```

> 小结：DateFormat可以将Date对象和字符串相互转换。

12.4 练习1(初恋女友的出生日期)

```
1  /*
2      假设，你初恋的出生年月日为:2000-11-11
3      请用字符串表示这个数据，并将其转换为:2000年11月11日
4
5      创建一个Date对象表示2000年11月11日
6      创建一个SimpleDateFormat对象，并定义格式为年月日把时间变成:2000年11月11日
7  */
8
9  //1.可以通过2000-11-11进行解析，解析成一个Date对象
10 String str = "2000-11-11";
11 //2.解析
12 SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd");
13 Date date = sdf1.parse(str);
14 //3.格式化
15 SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy年MM月dd日");
16 String result = sdf2.format(date);
17 System.out.println(result);
```

12.5 练习2(秒杀活动)

```
1  /* 需求:
2          秒杀活动开始时间:2023年11月11日 0:0:0(毫秒值)
3          秒杀活动结束时间:2023年11月11日 0:10:0(毫秒值)
4
5          小贾下单并付款的时间为:2023年11月11日 0:01:0
6          小皮下单并付款的时间为:2023年11月11日 0:11:0
7          用代码说明这两位同学有没有参加上秒杀活动?
8        */
9
10 //1.定义字符串表示三个时间
11 String startstr = "2023年11月11日 0:0:0";
12 String endstr = "2023年11月11日 0:10:0";
13 String orderstr = "2023年11月11日 0:01:00";
14 //2.解析上面的三个时间，得到Date对象
15 SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日HH:mm:ss");
16 Date startDate = sdf.parse(startstr);
```

```
Date endDate = sdf.parse(endstr);
Date orderDate = sdf.parse(orderstr);

//3.得到三个时间的毫秒值
long startTime = startDate.getTime();
long endTime = endDate.getTime();
long orderTime = orderDate.getTime();

//4.判断
if (orderTime >= startTime && orderTime <= endTime) {
    System.out.println("参加秒杀活动成功");
} else {
    System.out.println("参加秒杀活动失败");
}
```

第三章 Calendar类（好像没学）

13.1 概述

- java.util.Calendar类表示一个“日历类”，可以进行日期运算。它是一个抽象类，不能创建对象，我们可以使用它的子类：java.util.GregorianCalendar类。
- 有两种方式可以获取GregorianCalendar对象：
    - 直接创建GregorianCalendar对象；
    - 通过Calendar的静态方法getInstance()方法获取GregorianCalendar对象【本次课使用】

13.2 常用方法

方法名	说明
public static Calendar getInstance()	获取一个它的子类GregorianCalendar对象。

方法名	说明
public int get(int field)	获取某个字段的值。field参数表示获取哪个字段的值， 可以使用Calender中定义的常量来表示： Calendar.YEAR : 年 Calendar.MONTH ： 月 Calendar.DAY_OF_MONTH：月中的日期 Calendar.HOUR：小时 Calendar.MINUTE：分钟 Calendar.SECOND：秒 Calendar.DAY_OF_WEEK：星期
public void set(int field,int value)	设置某个字段的值
public void add(int field,int amount)	为某个字段增加/减少指定的值

13.3 get方法示例

```
public class Demo {
    public static void main(String[] args) {
        //1.获取一个GregorianCalendar对象
        Calendar instance = Calendar.getInstance();//获取子类对象

        //2.打印子类对象
        System.out.println(instance);

        //3.获取属性
        int year = instance.get(Calendar.YEAR);
        int month = instance.get(Calendar.MONTH) + 1;//Calendar的月份值是0-11
        int day = instance.get(Calendar.DAY_OF_MONTH);

        int hour = instance.get(Calendar.HOUR);
        int minute = instance.get(Calendar.MINUTE);
        int second = instance.get(Calendar.SECOND);

        int week = instance.get(Calendar.DAY_OF_WEEK);//返回值范围：1--7，分别表示："星期日","星期一","星期二",...,"星期六"

        System.out.println(year + "年" + month + "月" + day + "日" +
                           hour + ":" + minute + ":" + second);
        System.out.println(getWeek(week));

    }

```

```
    //查表法，查询星期几
    public static String getWeek(int w) {//w = 1 --- 7
        //做一个表(数组)
        String[] weekArray = {"星期日", "星期一", "星期二", "星期三", "星期四", "星期五",
"星期六"};
        //          索引     [0]     [1]      [2]      [3]      [4]      [5]      [6]
        //查表
        return weekArray[w - 1];
    }
}

```

13.4 set方法示例：

```
public class Demo {
    public static void main(String[] args) {
        //设置属性——set(int field,int value):
        Calendar c1 = Calendar.getInstance();//获取当前日期

        //计算班长出生那天是星期几(假如班长出生日期为：1998年3月18日)
        c1.set(Calendar.YEAR, 1998);
        c1.set(Calendar.MONTH, 3 - 1);//转换为Calendar内部的月份值
        c1.set(Calendar.DAY_OF_MONTH, 18);

        int w = c1.get(Calendar.DAY_OF_WEEK);
        System.out.println("班长出生那天是：" + getWeek(w));


    }
    //查表法，查询星期几
    public static String getWeek(int w) {//w = 1 --- 7
        //做一个表(数组)
        String[] weekArray = {"星期日", "星期一", "星期二", "星期三", "星期四", "星期五",
"星期六"};
        //          索引     [0]     [1]      [2]      [3]      [4]      [5]      [6]
        //查表
        return weekArray[w - 1];
    }
}
```

13.5 add方法示例：

```
public class Demo {
    public static void main(String[] args) {
        //计算200天以后是哪年哪月哪日，星期几?
        Calendar c2 = Calendar.getInstance();//获取当前日期
        c2.add(Calendar.DAY_OF_MONTH, 200);//日期加200

        int y = c2.get(Calendar.YEAR);
        int m = c2.get(Calendar.MONTH) + 1;//转换为实际的月份
        int d = c2.get(Calendar.DAY_OF_MONTH);

        int wk = c2.get(Calendar.DAY_OF_WEEK);
        System.out.println("200天后是：" + y + "年" + m + "月" + d + "日" +
getWeek(wk));

    }
    //查表法，查询星期几
    public static String getWeek(int w) {//w = 1 --- 7
        //做一个表(数组)
        String[] weekArray = {"星期日", "星期一", "星期二", "星期三", "星期四", "星期五",
"星期六"};
        //          索引     [0]     [1]      [2]      [3]      [4]      [5]      [6]
        //查表
        return weekArray[w - 1];
    }
}
```

包装类

14.1 概述

Java提供了两个类型系统，基本类型与引用类型，使用基本类型在于效率，然而很多情况，会创建对象使用，因为对象可以做更多的功能，如果想要我们的基本类型像对象一样操作，就可以使用基本类型对应的包装类，如下：

基本类型	对应的包装类（位于java.lang包中）
byte	Byte
short	Short
int	**Integer**

基本类型	对应的包装类（位于java.lang包中）
long	Long
float	Float
double	Double
char	**Character**
boolean	Boolean

14.2 Integer类

- Integer类概述

  包装一个对象中的原始类型 int 的值

- Integer类构造方法及静态方法

方法名	说明
public Integer(int value)	根据 int 值创建 Integer 对象(过时)
public Integer(String s)	根据 String 值创建 Integer 对象(过时)
public static Integer valueOf(int i)	返回表示指定的 int 值的 Integer 实例
public static Integer valueOf(String s)	返回保存指定String值的 Integer 对象
static string tobinarystring(int i)	得到二进制
static string tooctalstring(int i)	得到八进制
static string toHexstring(int i)	得到十六进制
static int parseInt(string s)	将字符串类型的整数转成int类型的整数

- 示例代码

```
//public Integer(int value)：根据 int 值创建 Integer 对象(过时)
Integer i1 = new Integer(100);
System.out.println(i1);

//public Integer(String s)：根据 String 值创建 Integer 对象(过时)
Integer i2 = new Integer("100");
//Integer i2 = new Integer("abc"); //NumberFormatException
System.out.println(i2);
System.out.println("--------");

//public static Integer valueOf(int i)：返回表示指定的 int 值的 Integer 实例
```

```
Integer i3 = Integer.valueOf(100);
System.out.println(i3);

//public static Integer valueOf(String s)：返回保存指定String值的Integer对象
Integer i4 = Integer.valueOf("100");
System.out.println(i4);
```

```
/*
        public static string tobinarystring(int i) 得到二进制
        public static string tooctalstring(int i) 得到八进制
        public static string toHexstring(int i) 得到十六进制
        public static int parseInt(string s) 将字符串类型的整数转成int类型的整数
 */

//1.把整数转成二进制，十六进制
String str1 = Integer.toBinaryString(100);
System.out.println(str1);//1100100

//2.把整数转成八进制
String str2 = Integer.toOctalString(100);
System.out.println(str2);//144

//3.把整数转成十六进制
String str3 = Integer.toHexString(100);
System.out.println(str3);//64

//4.将字符串类型的整数转成int类型的整数
//强类型语言:每种数据在java中都有各自的数据类型
//在计算的时候，如果不是同一种数据类型，是无法直接计算的。
int i = Integer.parseInt("123");
System.out.println(i);
System.out.println(i + 1);//124
//细节1:
//在类型转换的时候，括号中的参数只能是数字不能是其他，否则代码会报错
//细节2:
//8种包装类当中，除了Character都有对应的parseXxx的方法，进行类型转换
String str = "true";
boolean b = Boolean.parseBoolean(str);
System.out.println(b);
```

14.3 装箱与拆箱

基本类型与对应的包装类对象之间，来回转换的过程称为”装箱“与”拆箱“：

- **装箱**：从基本类型转换为对应的包装类对象。
- **拆箱**：从包装类对象转换为对应的基本类型。

用Integer与 int为例：（看懂代码即可）

基本数值---->包装对象

```
Integer i = new Integer(4);//使用构造函数函数
Integer iii = Integer.valueOf(4);//使用包装类中的valueOf方法
```

包装对象---->基本数值

```
int num = i.intValue();
```

14.4 自动装箱与自动拆箱

由于我们经常要做基本类型与包装类之间的转换，从Java 5（JDK 1.5）开始，基本类型与包装类的装箱、拆箱动作可以自动完成。例如：

```
Integer i = 4;//自动装箱。相当于Integer i = Integer.valueOf(4);
i = i + 5;//等号右边：将i对象转成基本数值(自动拆箱) i.intValue() + 5;
//加法运算完成后，再次装箱，把基本数值转成对象。
```

14.5 基本类型与字符串之间的转换

基本类型转换为String

- 转换方式
- 方式一：直接在数字后加一个空字符串
- 方式二：通过String类静态方法valueOf()
- 示例代码

```
public class IntegerDemo {
    public static void main(String[] args) {
        //int --- String
        int number = 100;
        //方式1
        String s1 = number + "";
        System.out.println(s1);
        //方式2
        //public static String valueOf(int i)
        String s2 = String.valueOf(number);
        System.out.println(s2);
        System.out.println("--------");
    }
}
```

String转换成基本类型

除了Character类之外，其他所有包装类都具有parseXxx静态方法可以将字符串参数转换为对应的基本类型：

- **public static byte parseByte(String s)**：将字符串参数转换为对应的byte基本类型。
- **public static short parseShort(String s)**：将字符串参数转换为对应的short基本类型。
- **public static int parseInt(String s)：将字符串参数转换为对应的int基本类型。**
- **public static long parseLong(String s)：将字符串参数转换为对应的long基本类型。**
- **public static float parseFloat(String s)**：将字符串参数转换为对应的float基本类型。
- **public static double parseDouble(String s)**：将字符串参数转换为对应的double基本类型。
- **public static boolean parseBoolean(String s)**：将字符串参数转换为对应的boolean基本类型。

代码使用（仅以Integer类的静态方法parseXxx为例）如：

- 转换方式
    - 方式一：先将字符串数字转成Integer，再调用valueOf()方法
    - 方式二：通过Integer静态方法parseInt()进行转换
- 示例代码

```
public class IntegerDemo {
    public static void main(String[] args) {
        //String --- int
        String s = "100";
        //方式1：String --- Integer --- int
        Integer i = Integer.valueOf(s);
        //public int intValue()
        int x = i.intValue();
        System.out.println(x);
        //方式2
        //public static int parseInt(String s)
        int y = Integer.parseInt(s);
        System.out.println(y);
    }
}
```

> 注意:如果字符串参数的内容无法正确转换为对应的基本类型，则会抛出 **java.lang.NumberFormatException** 异常。

14.6 底层原理

建议：获取Integer对象的时候不要自己new，而是采取直接赋值或者静态方法valueOf的方式

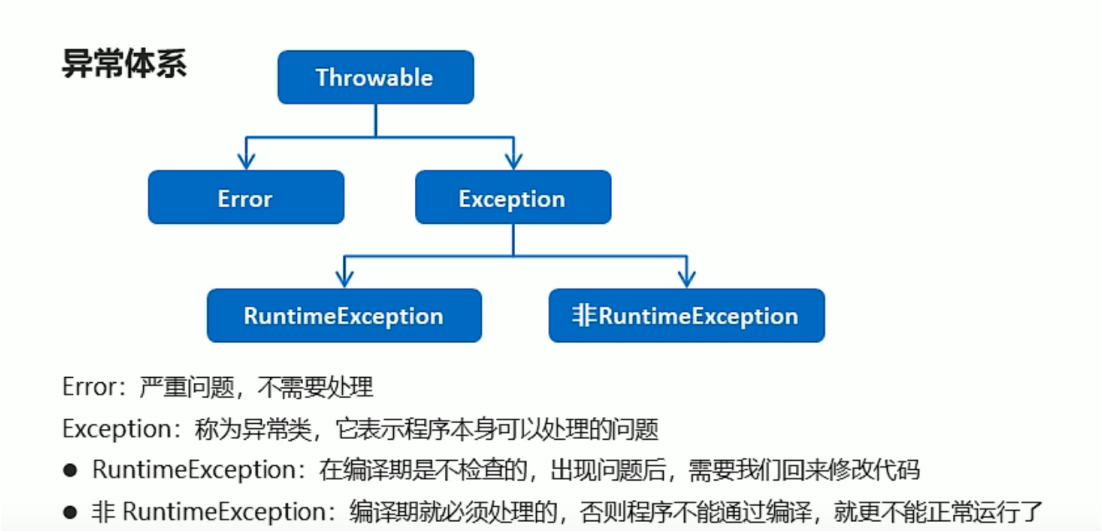
因为在实际开发中，-128~127之间的数据，用的比较多。如果每次使用都是new对象，那么太浪费内存了。

所以，提前把这个范围之内的每一个数据都创建好对象，如果要用到了不会创建新的，而是返回已经创建好的对象。

```
//1.利用构造方法获取Integer的对象(JDK5以前的方式)
/*Integer i1 = new Integer(1);
        Integer i2 = new Integer("1");
        System.out.println(i1);
        System.out.println(i2);*/

//2.利用静态方法获取Integer的对象(JDK5以前的方式)
Integer i3 = Integer.valueOf(123);
Integer i4 = Integer.valueOf("123");
Integer i5 = Integer.valueOf("123", 8);

System.out.println(i3);
System.out.println(i4);
System.out.println(i5);

//3.这两种方式获取对象的区别(掌握)
//底层原理：
//因为在实际开发中，-128~127之间的数据，用的比较多。
//如果每次使用都是new对象，那么太浪费内存了
//所以，提前把这个范围之内的每一个数据都创建好对象
//如果要用到了不会创建新的，而是返回已经创建好的对象。
Integer i6 = Integer.valueOf(127);
Integer i7 = Integer.valueOf(127);
System.out.println(i6 == i7);//true

Integer i8 = Integer.valueOf(128);
Integer i9 = Integer.valueOf(128);
System.out.println(i8 == i9);//false

//因为看到了new关键字，在Java中，每一次new都是创建了一个新的对象
//所以下面的两个对象都是new出来，地址值不一样。
/*Integer i10 = new Integer(127);
        Integer i11 = new Integer(127);
        System.out.println(i10 == i11);

        Integer i12 = new Integer(128);
```

```
37        Integer i13 = new Integer(128);
38        System.out.println(i12 == i13);*/
```

异常



Error：严重问题，不需要处理

Exception：称为异常类，它表示程序本身可以处理的问题

- RuntimeException：在编译期是不检查的，出现问题后，需要我们回来修改代码
- 非 RuntimeException：编译期就必须处理的，否则程序不能通过编译，就更不能正常运行了

15.1 try catch

```
try {
        可能出现异常的代码;
} catch(异常类名 变量名) {
        异常的处理代码;
}
```

```
1  package milkSystem;
2  public class Test {
3      public static void main(String[] args) {
4          System.out.println("开始");
5          method();
6          System.out.println("结束");
7      }
8      public static void  method(){
9          try {
10             int[] arr={1,2,3};
11             System.out.println(arr[3]);
12         }catch (ArrayIndexOutOfBoundsException e){
13 //             System.out.println("err");
```

```
14              e.printStackTrace();//打印错误信息
15          }
16
17      }
18  }
```

1.5 Throwable 的成员方法

方法名	说明
public String getMessage()	返回此 throwable 的详细消息字符串
public String toString()	返回此可抛出的简短描述
public void printStackTrace()	把异常的错误信息输出在控制台

15.2 throws

格式：

```
throws 异常类名;
```

注意：这个格式是跟在方法的括号后面的

IO

16.1 IO流

16.1.1 IO流概述和分类【理解】

- IO流介绍
  - IO：输入/输出(Input/Output)
  - 流：是一种抽象概念，是对数据传输的总称。也就是说数据在设备间的传输称为流，流的本质是数据传输
  - IO流就是用来处理设备间数据传输问题的。常见的应用：文件复制；文件上传；文件下载
- IO流的分类
  - 按照数据的流向

        - 输入流：读数据
        - 输出流：写数据
    - 按照数据类型来分
        - 字节流
            - 字节输入流
            - 字节输出流
        - 字符流
            - 字符输入流
            - 字符输出流
- IO流的使用场景
    - 如果操作的是纯文本文件，优先使用字符流
    - 如果操作的是图片、视频、音频等二进制文件。优先使用字节流
    - 如果不确定文件类型，优先使用字节流。字节流是万能的流

16.1.2字节流写数据【应用】

- 字节流抽象基类
    - InputStream：这个抽象类是表示字节输入流的所有类的超类
    - OutputStream：这个抽象类是表示字节输出流的所有类的超类
    - 子类名特点：子类名称都是以其父类名作为子类名的后缀
- 字节输出流
    - FileOutputStream(String name)：创建文件输出流以指定的名称写入文件
- 使用字节输出流写数据的步骤
    - 创建字节输出流对象(调用系统功能创建了文件，创建字节输出流对象，让字节输出流对象指向文件)
    - 调用字节输出流对象的写数据方法
    - 释放资源(关闭此文件输出流并释放与此流相关联的任何系统资源)
- 示例代码

```
public class FileOutputStreamDemo01 {
    public static void main(String[] args) throws IOException {
        //创建字节输出流对象
        //FileOutputStream(String name)：创建文件输出流以指定的名称写入文件
        FileOutputStream fos = new FileOutputStream("myByteStream\\fos.txt");
        /*
            做了三件事情：
                A:调用系统功能创建了文件
                B:创建了字节输出流对象
```

```
            C:让字节输出流对象指向创建好的文件
        */

        //void write(int b)：将指定的字节写入此文件输出流
        fos.write(97);
//        fos.write(57);
//        fos.write(55);

        //最后都要释放资源
        //void close()：关闭此文件输出流并释放与此流相关联的任何系统资源。
        fos.close();
    }
}
```

16.1.3字节流写数据的三种方式【应用】

- 写数据的方法分类

方法名	说明
void write(int b)	将指定的字节写入此文件输出流 一次写一个字节数据
void write(byte[] b)	将 b.length字节从指定的字节数组写入此文件输出流 一次写一个字节数组数据
void write(byte[] b, int off, int len)	将 len字节从指定的字节数组开始，从偏移量off开始写入此文件输出流 一次写一个字节数组的部分数据

- 示例代码

```
public class FileOutputStreamDemo02 {
    public static void main(String[] args) throws IOException {
        //FileOutputStream(String name)：创建文件输出流以指定的名称写入文件
        FileOutputStream fos = new FileOutputStream("myByteStream\\fos.txt");
        //new File(name)
//        FileOutputStream fos = new FileOutputStream(new File("myByteStream\\fos.txt"));

        //FileOutputStream(File file)：创建文件输出流以写入由指定的 File对象表示的文件
//        File file = new File("myByteStream\\fos.txt");
//        FileOutputStream fos2 = new FileOutputStream(file);
//        FileOutputStream fos2 = new FileOutputStream(new File("myByteStream\\fos.txt"));

        //void write(int b)：将指定的字节写入此文件输出流
//        fos.write(97);
//        fos.write(98);
//        fos.write(99);
//        fos.write(100);
```

```
//        fos.write(101);

//        void write(byte[] b)：将 b.length字节从指定的字节数组写入此文件输出流
//        byte[] bys = {97, 98, 99, 100, 101};
        //byte[] getBytes()：返回字符串对应的字节数组
        byte[] bys = "abcde".getBytes();
//        fos.write(bys);

        //void write(byte[] b, int off, int len)：将 len字节从指定的字节数组开始，从偏移量off开始写入此文件输出流
//        fos.write(bys,0,bys.length);
        fos.write(bys,1,3);

        //释放资源
        fos.close();
    }
}
```

16.1.4字节流写数据的两个小问题【应用】

- 字节流写数据如何实现换行
  - windows:\r\n
  - linux:\n
  - mac:\r
- 字节流写数据如何实现追加写入
  - public FileOutputStream(String name,boolean append)
  - 创建文件输出流以指定的名称写入文件。如果第二个参数为true ， 则字节将写入文件的末尾而不是开头
- 示例代码

```
public class FileOutputStreamDemo03 {
    public static void main(String[] args) throws IOException {
        //创建字节输出流对象
//        FileOutputStream fos = new FileOutputStream("myByteStream\\fos.txt");
        FileOutputStream fos = new FileOutputStream("myByteStream\\fos.txt",true);

        //写数据
        for (int i = 0; i < 10; i++) {
            fos.write("hello".getBytes());
            fos.write("\r\n".getBytes());
        }

        //释放资源
        fos.close();
    }
```

```
16 }
```

16.1.5字节流写数据加异常处理【应用】

- 异常处理格式
  - try-catch-finally

```
try{
    可能出现异常的代码;
}catch(异常类名 变量名){
    异常的处理代码;
}finally{
    执行所有清除操作;
}
```

  - finally特点
    - 被finally控制的语句一定会执行，除非JVM退出
- 示例代码

```
public class FileOutputStreamDemo04 {
    public static void main(String[] args) {
        //加入finally来实现释放资源
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream("myByteStream\\fos.txt");
            fos.write("hello".getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

16.1.6字节流读数据(一次读一个字节数据)【应用】

- 字节输入流
  - FileInputStream(String name)：通过打开与实际文件的连接来创建一个FileInputStream ，该文件由文件系统中的路径名name命名
- 字节输入流读取数据的步骤

- 创建字节输入流对象
- 调用字节输入流对象的读数据方法
- 释放资源

- 示例代码

```
public class FileInputStreamDemo01 {
    public static void main(String[] args) throws IOException {
        //创建字节输入流对象
        //FileInputStream(String name)
        FileInputStream fis = new FileInputStream("myByteStream\\fos.txt");

        int by;
        /*
            fis.read()：读数据
            by=fis.read()：把读取到的数据赋值给by
            by != -1：判断读取到的数据是否是-1
         */
        while ((by=fis.read())!=-1) {
            System.out.print((char)by);
        }

        //释放资源
        fis.close();
    }
}
```

16.1.7字节流复制文本文件【应用】

- 案例需求

  把“E:\itcast\窗里窗外.txt”复制到模块目录下的“窗里窗外.txt”

- 实现步骤
  - 复制文本文件，其实就把文本文件的内容从一个文件中读取出来(数据源)，然后写入到另一个文件中(目的地)
  - 数据源：

    E:\itcast\窗里窗外.txt --- 读数据 --- InputStream --- FileInputStream
  - 目的地：

    myByteStream\窗里窗外.txt --- 写数据 --- OutputStream --- FileOutputStream

- 代码实现

```
public class CopyTxtDemo {
    public static void main(String[] args) throws IOException {
        //根据数据源创建字节输入流对象
        FileInputStream fis = new FileInputStream("E:\\itcast\\窗里窗外.txt");
```

```
        //根据目的地创建字节输出流对象
        FileOutputStream fos = new FileOutputStream("myByteStream\\窗里窗外.txt");

        //读写数据，复制文本文件(一次读取一个字节，一次写入一个字节)
        int by;
        while ((by=fis.read())!=-1) {
            fos.write(by);
        }

        //释放资源
        fos.close();
        fis.close();
    }
}
```

16.1.8字节流读数据(一次读一个字节数组数据)【应用】

- 一次读一个字节数组的方法
  - public int read(byte[] b)：从输入流读取最多b.length个字节的数据
  - 返回的是读入缓冲区的总字节数,也就是实际的读取字节个数
- 示例代码

```
public class FileInputStreamDemo02 {
    public static void main(String[] args) throws IOException {
        //创建字节输入流对象
        FileInputStream fis = new FileInputStream("myByteStream\\fos.txt");

        /*
            hello\r\n
            world\r\n

            第一次：hello
            第二次：\r\nwor
            第三次：ld\r\nr

         */

        byte[] bys = new byte[1024]; //1024及其整数倍
        int len;
        while ((len=fis.read(bys))!=-1) {
            System.out.print(new String(bys,0,len));
        }

        //释放资源
        fis.close();
    }
}
```

16.1.9字节流复制图片【应用】

- 案例需求

  把“E:\itcast\mn.jpg”复制到模块目录下的“mn.jpg”

- 实现步骤
  - 根据数据源创建字节输入流对象
  - 根据目的地创建字节输出流对象
  - 读写数据，复制图片(一次读取一个字节数组，一次写入一个字节数组)
  - 释放资源

- 代码实现

```
public class CopyJpgDemo {
    public static void main(String[] args) throws IOException {
        //根据数据源创建字节输入流对象
        FileInputStream fis = new FileInputStream("E:\\itcast\\mn.jpg");
        //根据目的地创建字节输出流对象
        FileOutputStream fos = new FileOutputStream("myByteStream\\mn.jpg");

        //读写数据，复制图片(一次读取一个字节数组，一次写入一个字节数组)
        byte[] bys = new byte[1024];
        int len;
        while ((len=fis.read(bys))!=-1) {
            fos.write(bys,0,len);
        }

        //释放资源
        fos.close();
        fis.close();
    }
}
```

17.1 字节缓冲流

17.1.1字节缓冲流构造方法【应用】

- 字节缓冲流介绍
  - lBufferOutputStream：该类实现缓冲输出流。 通过设置这样的输出流，应用程序可以向底层输出流写入字节，而不必为写入的每个字节导致底层系统的调用

    - lBufferedInputStream：创建BufferedInputStream将创建一个内部缓冲区数组。 当从流中读取或跳过字节时，内部缓冲区将根据需要从所包含的输入流中重新填充，一次很多字节

- 构造方法：

方法名	说明
BufferedOutputStream(OutputStream out)	创建字节缓冲输出流对象
BufferedInputStream(InputStream in)	创建字节缓冲输入流对象

- 示例代码

```
public class BufferStreamDemo {
    public static void main(String[] args) throws IOException {
        //字节缓冲输出流：BufferedOutputStream(OutputStream out)

        BufferedOutputStream bos = new BufferedOutputStream(new
                FileOutputStream("myByteStream\\bos.txt"));
        //写数据
        bos.write("hello\r\n".getBytes());
        bos.write("world\r\n".getBytes());
        //释放资源
        bos.close();


        //字节缓冲输入流：BufferedInputStream(InputStream in)
        BufferedInputStream bis = new BufferedInputStream(new
                FileInputStream("myByteStream\\bos.txt"));

        //一次读取一个字节数据
//        int by;
//        while ((by=bis.read())!=-1) {
//            System.out.print((char)by);
//        }

        //一次读取一个字节数组数据
        byte[] bys = new byte[1024];
        int len;
        while ((len=bis.read(bys))!=-1) {
            System.out.print(new String(bys,0,len));
        }

        //释放资源
        bis.close();
    }
}
```

17.1.2字节流复制视频【应用】

- 案例需求

  把“E:\itcast\字节流复制图片.avi”复制到模块目录下的“字节流复制图片.avi”

- 实现步骤
  - 根据数据源创建字节输入流对象
  - 根据目的地创建字节输出流对象
  - 读写数据，复制视频
  - 释放资源

- 代码实现

```
public class CopyAviDemo {
    public static void main(String[] args) throws IOException {
        //记录开始时间
        long startTime = System.currentTimeMillis();

        //复制视频
//        method1();
//        method2();
//        method3();
        method4();

        //记录结束时间
        long endTime = System.currentTimeMillis();
        System.out.println("共耗时：" + (endTime - startTime) + "毫秒");
    }

    //字节缓冲流一次读写一个字节数组
    public static void method4() throws IOException {
        BufferedInputStream bis = new BufferedInputStream(new FileInputStream("E:\\itcast\\字节流复制图片.avi"));
        BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("myByteStream\\字节流复制图片.avi"));

        byte[] bys = new byte[1024];
        int len;
        while ((len=bis.read(bys))!=-1) {
            bos.write(bys,0,len);
        }

        bos.close();
        bis.close();
    }

    //字节缓冲流一次读写一个字节
```

```
33      public static void method3() throws IOException {
34          BufferedInputStream bis = new BufferedInputStream(new
    FileInputStream("E:\\itcast\\字节流复制图片.avi"));
35          BufferedOutputStream bos = new BufferedOutputStream(new
    FileOutputStream("myByteStream\\字节流复制图片.avi"));
36
37          int by;
38          while ((by=bis.read())!=-1) {
39              bos.write(by);
40          }
41
42          bos.close();
43          bis.close();
44      }
45
46
47      //基本字节流一次读写一个字节数组
48      public static void method2() throws IOException {
49          //E:\\itcast\\字节流复制图片.avi
50          //模块目录下的 字节流复制图片.avi
51          FileInputStream fis = new FileInputStream("E:\\itcast\\字节流复制图片.avi");
52          FileOutputStream fos = new FileOutputStream("myByteStream\\字节流复制图
    片.avi");
53
54          byte[] bys = new byte[1024];
55          int len;
56          while ((len=fis.read(bys))!=-1) {
57              fos.write(bys,0,len);
58          }
59
60          fos.close();
61          fis.close();
62      }
63
64      //基本字节流一次读写一个字节
65      public static void method1() throws IOException {
66          //E:\\itcast\\字节流复制图片.avi
67          //模块目录下的 字节流复制图片.avi
68          FileInputStream fis = new FileInputStream("E:\\itcast\\字节流复制图片.avi");
69          FileOutputStream fos = new FileOutputStream("myByteStream\\字节流复制图
    片.avi");
70
71          int by;
72          while ((by=fis.read())!=-1) {
73              fos.write(by);
74          }
75
76          fos.close();
```

```
        fis.close();
    }
}
```

18.1 字符流

18.1.1为什么会出现字符流【理解】

- 字符流的介绍

  由于字节流操作中文不是特别的方便，所以Java就提供字符流

  字符流 = 字节流 + 编码表

- 中文的字节存储方式

  用字节流复制文本文件时，文本文件也会有中文，但是没有问题，原因是最终底层操作会自动进行字节拼接成中文，如何识别是中文的呢?

  汉字在存储的时候，无论选择哪种编码存储，第一个字节都是负数

18.1.2编码表【理解】

- 什么是字符集

  是一个系统支持的所有字符的集合，包括各国家文字、标点符号、图形符号、数字等

  l计算机要准确的存储和识别各种字符集符号，就需要进行字符编码，一套字符集必然至少有一套字符编码。常见字符集有ASCII字符集、GBXXX字符集、Unicode字符集等

- 常见的字符集
  - ASCII字符集：

    lASCII：是基于拉丁字母的一套电脑编码系统，用于显示现代英语，主要包括控制字符(回车键、退格、换行键等)和可显示字符(英文大小写字符、阿拉伯数字和西文符号)

    基本的ASCII字符集，使用7位表示一个字符，共128字符。ASCII的扩展字符集使用8位表示一个字符，共256字符，方便支持欧洲常用字符。是一个系统支持的所有字符的集合，包括各国家文字、标点符号、图形符号、数字等

  - GBXXX字符集：

    GBK：最常用的中文码表。是在GB2312标准基础上的扩展规范，使用了双字节编码方案，共收录了21003个汉字，完全兼容GB2312标准，同时支持繁体汉字以及日韩汉字等

  - Unicode字符集：

    UTF-8编码：可以用来表示Unicode标准中任意字符，它是电子邮件、网页及其他存储或传送文字的应用 中，优先采用的编码。互联网工程工作小组（IETF）要求所有互联网协议都必须支持UTF-8编码。它使用一至四个字节为每个字符编码

    编码规则：

128个US-ASCII字符，只需一个字节编码

拉丁文等字符，需要二个字节编码

大部分常用字（含中文），使用三个字节编码

其他极少使用的Unicode辅助字符，使用四字节编码

18.1.3字符串中的编码解码问题【应用】

- 相关方法

方法名	说明
byte[] getBytes()	使用平台的默认字符集将该 String编码为一系列字节
byte[] getBytes(String charsetName)	使用指定的字符集将该 String编码为一系列字节
String(byte[] bytes)	使用平台的默认字符集解码指定的字节数组来创建字符串
String(byte[] bytes, String charsetName)	通过指定的字符集解码指定的字节数组来创建字符串

- 代码演示

```
public class StringDemo {
    public static void main(String[] args) throws UnsupportedEncodingException {
        //定义一个字符串
        String s = "中国";

        //byte[] bys = s.getBytes(); //[-28, -72, -83, -27, -101, -67]
        //byte[] bys = s.getBytes("UTF-8"); //[-28, -72, -83, -27, -101, -67]
        byte[] bys = s.getBytes("GBK"); //[-42, -48, -71, -6]
        System.out.println(Arrays.toString(bys));

        //String ss = new String(bys);
        //String ss = new String(bys,"UTF-8");
        String ss = new String(bys,"GBK");
        System.out.println(ss);
    }
}
```

18.1.4字符流中的编码解码问题【应用】

- 字符流中和编码解码问题相关的两个类
  - InputStreamReader：是从字节流到字符流的桥梁

    它读取字节，并使用指定的编码将其解码为字符

    它使用的字符集可以由名称指定，也可以被明确指定，或者可以接受平台的默认字符集
  - OutputStreamWriter：是从字符流到字节流的桥梁

    是从字符流到字节流的桥梁，使用指定的编码将写入的字符编码为字节

    它使用的字符集可以由名称指定，也可以被明确指定，或者可以接受平台的默认字符集
- 构造方法

方法名	说明
InputStreamReader(InputStream in)	使用默认字符编码创建InputStreamReader对象
InputStreamReader(InputStream in,String chatset)	使用指定的字符编码创建InputStreamReader对象
OutputStreamWriter(OutputStream out)	使用默认字符编码创建OutputStreamWriter对象
OutputStreamWriter(OutputStream out,String charset)	使用指定的字符编码创建OutputStreamWriter对象

- 代码演示

```
1  public class ConversionStreamDemo {
2      public static void main(String[] args) throws IOException {
3          //OutputStreamWriter osw = new OutputStreamWriter(new
           FileOutputStream("myCharStream\\osw.txt"));
4          OutputStreamWriter osw = new OutputStreamWriter(new
       FileOutputStream("myCharStream\\osw.txt"),"GBK");
5          osw.write("中国");
6          osw.close();
7
8          //InputStreamReader isr = new InputStreamReader(new
     FileInputStream("myCharStream\\osw.txt"));
9          InputStreamReader isr = new InputStreamReader(new
     FileInputStream("myCharStream\\osw.txt"),"GBK");
10         //一次读取一个字符数据
11         int ch;
12         while ((ch=isr.read())!=-1) {
```

```
            System.out.print((char)ch);
        }
        isr.close();
    }
}
```

18.1.5字符流写数据的5种方式【应用】

- 方法介绍

方法名	说明
void　write(int c)	写一个字符
void　write(char[] cbuf)	写入一个字符数组
void write(char[] cbuf, int off, int len)	写入字符数组的一部分
void write(String str)	写一个字符串
void write(String str, int off, int len)	写一个字符串的一部分

- 刷新和关闭的方法

方法名	说明
flush()	刷新流，之后还可以继续写数据
close()	关闭流，释放资源，但是在关闭之前会先刷新流。一旦关闭，就不能再写数据

- 代码演示

```
public class OutputStreamWriterDemo {
    public static void main(String[] args) throws IOException {
        OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("myCharStream\\osw.txt"));

        //void write(int c)：写一个字符
//        osw.write(97);
//        osw.write(98);
//        osw.write(99);

        //void writ(char[] cbuf)：写入一个字符数组
        char[] chs = {'a', 'b', 'c', 'd', 'e'};
//        osw.write(chs);

        //void write(char[] cbuf, int off, int len)：写入字符数组的一部分
//        osw.write(chs, 0, chs.length);
//        osw.write(chs, 1, 3);
```

```
17
18          //void write(String str)：写一个字符串
19  //          osw.write("abcde");
20
21          //void write(String str, int off, int len)：写一个字符串的一部分
22  //          osw.write("abcde", 0, "abcde".length());
23          osw.write("abcde", 1, 3);
24
25          //释放资源
26          osw.close();
27      }
28  }
```

18.1.6字符流读数据的2种方式【应用】

- 方法介绍

方法名	说明
int read()	一次读一个字符数据
int read(char[] cbuf)	一次读一个字符数组数据

- 代码演示

```
1   public class InputStreamReaderDemo {
2       public static void main(String[] args) throws IOException {
3
4           InputStreamReader isr = new InputStreamReader(new
    FileInputStream("myCharStream\\ConversionStreamDemo.java"));
5
6           //int read()：一次读一个字符数据
7   //          int ch;
8   //          while ((ch=isr.read())!=-1) {
9   //              System.out.print((char)ch);
10  //          }
11
12          //int read(char[] cbuf)：一次读一个字符数组数据
13          char[] chs = new char[1024];
14          int len;
15          while ((len = isr.read(chs)) != -1) {
16              System.out.print(new String(chs, 0, len));
17          }
18
19          //释放资源
20          isr.close();
21      }
22  }
```

18.1.7字符流复制Java文件【应用】

- 案例需求

  把模块目录下的“ConversionStreamDemo.java” 复制到模块目录下的“Copy.java”

- 实现步骤
  - 根据数据源创建字符输入流对象
  - 根据目的地创建字符输出流对象
  - 读写数据，复制文件
  - 释放资源

- 代码实现

```
public class CopyJavaDemo01 {
    public static void main(String[] args) throws IOException {
        //根据数据源创建字符输入流对象
        InputStreamReader isr = new InputStreamReader(new FileInputStream("myCharStream\\ConversionStreamDemo.java"));
        //根据目的地创建字符输出流对象
        OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("myCharStream\\Copy.java"));

        //读写数据，复制文件
        //一次读写一个字符数据
//        int ch;
//        while ((ch=isr.read())!=-1) {
//            osw.write(ch);
//        }

        //一次读写一个字符数组数据
        char[] chs = new char[1024];
        int len;
        while ((len=isr.read(chs))!=-1) {
            osw.write(chs,0,len);
        }

        //释放资源
        osw.close();
        isr.close();
    }
}
```

18.1.8字符流复制Java文件改进版【应用】

- 案例需求

  使用便捷流对象，把模块目录下的“ConversionStreamDemo.java” 复制到模块目录下的“Copy.java”

- 实现步骤
  - 根据数据源创建字符输入流对象
  - 根据目的地创建字符输出流对象
  - 读写数据，复制文件
  - 释放资源
- 代码实现

```
public class CopyJavaDemo02 {
    public static void main(String[] args) throws IOException {
        //根据数据源创建字符输入流对象
        FileReader fr = new FileReader("myCharStream\\ConversionStreamDemo.java");
        //根据目的地创建字符输出流对象
        FileWriter fw = new FileWriter("myCharStream\\Copy.java");

        //读写数据，复制文件
//        int ch;
//        while ((ch=fr.read())!=-1) {
//            fw.write(ch);
//        }

        char[] chs = new char[1024];
        int len;
        while ((len=fr.read(chs))!=-1) {
            fw.write(chs,0,len);
        }

        //释放资源
        fw.close();
        fr.close();
    }
}
```

18.1.9字符缓冲流【应用】

- 字符缓冲流介绍
  - BufferedWriter：将文本写入字符输出流，缓冲字符，以提供单个字符，数组和字符串的高效写入，可以指定缓冲区大小，或者可以接受默认大小。默认值足够大，可用于大多数用途

- BufferedReader：从字符输入流读取文本，缓冲字符，以提供字符，数组和行的高效读取，可以指定缓冲区大小，或者可以使用默认大小。 默认值足够大，可用于大多数用途

- 构造方法

方法名	说明
BufferedWriter(Writer out)	创建字符缓冲输出流对象
BufferedReader(Reader in)	创建字符缓冲输入流对象

- 代码演示

```
public class BufferedStreamDemo01 {
    public static void main(String[] args) throws IOException {
        //BufferedWriter(Writer out)
        BufferedWriter bw = new BufferedWriter(new
 FileWriter("myCharStream\\bw.txt"));
        bw.write("hello\r\n");
        bw.write("world\r\n");
        bw.close();

        //BufferedReader(Reader in)
        BufferedReader br = new BufferedReader(new
FileReader("myCharStream\\bw.txt"));

        //一次读取一个字符数据
//        int ch;
//        while ((ch=br.read())!=-1) {
//            System.out.print((char)ch);
//        }

        //一次读取一个字符数组数据
        char[] chs = new char[1024];
        int len;
        while ((len=br.read(chs))!=-1) {
            System.out.print(new String(chs,0,len));
        }

        br.close();
    }
}
```

18.1.10字符缓冲流复制Java文件【应用】

- 案例需求

  把模块目录下的ConversionStreamDemo.java 复制到模块目录下的 Copy.java

- 实现步骤
  - 根据数据源创建字符缓冲输入流对象
  - 根据目的地创建字符缓冲输出流对象
  - 读写数据，复制文件，使用字符缓冲流特有功能实现
  - 释放资源

- 代码实现

```
public class CopyJavaDemo01 {
    public static void main(String[] args) throws IOException {
        //根据数据源创建字符缓冲输入流对象
        BufferedReader br = new BufferedReader(new
FileReader("myCharStream\\ConversionStreamDemo.java"));
        //根据目的地创建字符缓冲输出流对象
        BufferedWriter bw = new BufferedWriter(new
FileWriter("myCharStream\\Copy.java"));

        //读写数据，复制文件
        //一次读写一个字符数据
//        int ch;
//        while ((ch=br.read())!=-1) {
//            bw.write(ch);
//        }

        //一次读写一个字符数组数据
        char[] chs = new char[1024];
        int len;
        while ((len=br.read(chs))!=-1) {
            bw.write(chs,0,len);
        }

        //释放资源
        bw.close();
        br.close();
    }
}
```

18.1.11字符缓冲流特有功能【应用】

- 方法介绍

  BufferedWriter：

方法名	说明
void newLine()	写一行行分隔符，行分隔符字符串由系统属性定义

  BufferedReader:

方法名	说明
String readLine()	读一行文字。 结果包含行的内容的字符串，不包括任何行终止字符如果流的结尾已经到达，则为null

- 代码演示

```
public class BufferedStreamDemo02 {
    public static void main(String[] args) throws IOException {

        //创建字符缓冲输出流
        BufferedWriter bw = new BufferedWriter(new
FileWriter("myCharStream\\bw.txt"));

        //写数据
        for (int i = 0; i < 10; i++) {
            bw.write("hello" + i);
            //bw.write("\r\n");
            bw.newLine();
            bw.flush();
        }

        //释放资源
        bw.close();

        //创建字符缓冲输入流
        BufferedReader br = new BufferedReader(new
FileReader("myCharStream\\bw.txt"));

        String line;
        while ((line=br.readLine())!=null) {
            System.out.println(line);
        }

        br.close();
    }
```

```
28 }
```

18.1.12字符缓冲流特有功能复制Java文件【应用】

- 案例需求

  使用特有功能把模块目录下的ConversionStreamDemo.java 复制到模块目录下的Copy.java

- 实现步骤
  - 根据数据源创建字符缓冲输入流对象
  - 根据目的地创建字符缓冲输出流对象
  - 读写数据，复制文件，使用字符缓冲流特有功能实现
  - 释放资源
- 代码实现

```
1  public class CopyJavaDemo02 {
2      public static void main(String[] args) throws IOException {
3          //根据数据源创建字符缓冲输入流对象
4          BufferedReader br = new BufferedReader(new
   FileReader("myCharStream\\ConversionStreamDemo.java"));
5          //根据目的地创建字符缓冲输出流对象
6          BufferedWriter bw = new BufferedWriter(new
   FileWriter("myCharStream\\Copy.java"));
7
8          //读写数据，复制文件
9          //使用字符缓冲流特有功能实现
10         String line;
11         while ((line=br.readLine())!=null) {
12             bw.write(line);
13             bw.newLine();
14             bw.flush();
15         }
16
17         //释放资源
18         bw.close();
19         br.close();
20     }
21 }
```

设计模式

单例设计模式

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

单例模式的结构

单例模式的主要有以下角色：

- 单例类。只能创建一个实例的类
- 访问类。使用单例类

单例模式的实现

> 单例设计模式分类两种：
> 饿汉式：类加载就会导致该单实例对象被创建
> 懒汉式：类加载不会导致该单实例对象被创建，而是首次使用该对象时才会创建

1. 饿汉式-方式1（静态变量方式）

```
/**
 * 饿汉式
 *     静态变量创建类的对象
 */
public class Singleton {
    //私有构造方法
    private Singleton() {}

    //在成员位置创建该类的对象
    private static Singleton instance = new Singleton();

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        return instance;
    }
}
```

说明：

该方式在成员位置声明Singleton类型的静态变量，并创建Singleton类的对象instance。instance对象是随着类的加载而创建的。如果该对象足够大的话，而一直没有使用就会造成内存的浪费。

2. 饿汉式-方式2（静态代码块方式）

```
/**
 * 恶汉式
 *     在静态代码块中创建该类对象
 */
public class Singleton {

    //私有构造方法
    private Singleton() {}

    //在成员位置创建该类的对象
    private static Singleton instance;

    static {
        instance = new Singleton();
    }

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        return instance;
    }
}
```

说明：

该方式在成员位置声明Singleton类型的静态变量，而对象的创建是在静态代码块中，也是对着类的加载而创建。所以和饿汉式的方式1基本上一样，当然该方式也存在内存浪费问题。

3. 懒汉式-方式1（线程不安全）

```
/**
 * 懒汉式
 *  线程不安全
 */
public class Singleton {
    //私有构造方法
    private Singleton() {}

    //在成员位置创建该类的对象
    private static Singleton instance;
```

```
11
12      //对外提供静态方法获取该对象
13      public static Singleton getInstance() {
14
15          if(instance == null) {
16              instance = new Singleton();
17          }
18          return instance;
19      }
20  }
```

说明：

从上面代码我们可以看出该方式在成员位置声明Singleton类型的静态变量，并没有进行对象的赋值操作，那么什么时候赋值的呢？当调用getInstance()方法获取Singleton类的对象的时候才创建Singleton类的对象，这样就实现了懒加载的效果。但是，如果是多线程环境，会出现线程安全问题。

4. 懒汉式-方式2（线程安全）

```
1   /**
2    * 懒汉式
3    *  线程安全
4    */
5   public class Singleton {
6       //私有构造方法
7       private Singleton() {}
8
9       //在成员位置创建该类的对象
10      private static Singleton instance;
11
12      //对外提供静态方法获取该对象
13      public static synchronized Singleton getInstance() {
14
15          if(instance == null) {
16              instance = new Singleton();
17          }
18          return instance;
19      }
20  }
```

说明：

该方式也实现了懒加载效果，同时又解决了线程安全问题。但是在getInstance()方法上添加了synchronized关键字，导致该方法的执行效果特别低。从上面代码我们可以看出，其实就是在初始化instance的时候才会出现线程安全问题，一旦初始化完成就不存在了。

5. 懒汉式-方式3（双重检查锁）

再来讨论一下懒汉模式中加锁的问题，对于 **getInstance()** 方法来说，绝大部分的操作都是读操作，读操作是线程安全的，所以我们没必让每个线程必须持有锁才能调用该方法，我们需要调整加锁的时机。由此也产生了一种新的实现模式：双重检查锁模式

```
/**
 * 双重检查方式
 */
public class Singleton {

    //私有构造方法
    private Singleton() {}

    private static Singleton instance;

   //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        //第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实例
        if(instance == null) {
            synchronized (Singleton.class) {
                //抢到锁之后再次判断是否为null
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

双重检查锁模式是一种非常好的单例实现模式，解决了单例、性能、线程安全问题，上面的双重检测锁模式看上去完美无缺，其实是存在问题，在多线程的情况下，可能会出现空指针问题，出现问题的原因是JVM在实例化对象的时候会进行优化和指令重排序操作。

要解决双重检查锁模式带来空指针异常的问题，只需要使用 **volatile** 关键字，**volatile** 关键字可以保证可见性和有序性。

```
/**
 * 双重检查方式
 */
public class Singleton {

    //私有构造方法
    private Singleton() {}

    private static volatile Singleton instance;

   //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        //第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实际
```

```
        if(instance == null) {
            synchronized (Singleton.class) {
                //抢到锁之后再次判断是否为空
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

小结：

添加 **volatile** 关键字之后的双重检查锁模式是一种比较好的单例实现模式，能够保证在多线程的情况下线程安全也不会有性能问题。

6. 懒汉式-方式4（静态内部类方式）

静态内部类单例模式中实例由内部类创建，由于 JVM 在加载外部类的过程中, 是不会加载静态内部类的, 只有内部类的属性/方法被调用时才会被加载, 并初始化其静态属性。静态属性由于被 **static** 修饰，保证只被实例化一次，并且严格保证实例化顺序。

```
/**
 * 静态内部类方式
 */
public class Singleton {

    //私有构造方法
    private Singleton() {}

    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

说明：

第一次加载Singleton类时不会去初始化INSTANCE，只有第一次调用getInstance，虚拟机加载SingletonHolder

并初始化INSTANCE，这样不仅能确保线程安全，也能保证 Singleton 类的唯一性。

小结：

静态内部类单例模式是一种优秀的单例模式，是开源项目中比较常用的一种单例模式。在没有加任何锁的情况下，保证了多线程下的安全，并且没有任何性能影响和空间的浪费。

7. 枚举方式

枚举类实现单例模式是极力推荐的单例实现模式，因为枚举类型是线程安全的，并且只会装载一次，设计者充分的利用了枚举的这个特性来实现单例模式，枚举的写法非常简单，而且枚举类型是所用单例实现中唯一一种不会被破坏的单例实现模式。

```
/**
 * 枚举方式
 */
public enum Singleton {
    INSTANCE;
}
```

说明：

枚举方式属于恶汉式方式。

存在的问题

问题演示

破坏单例模式：

使上面定义的单例类（Singleton）可以创建多个对象，枚举方式除外。有两种方式，分别是序列化和反射。

- 序列化反序列化

  Singleton类：

```
public class Singleton implements Serializable {

    //私有构造方法
    private Singleton() {}

    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

Test类：

```
public class Test {
    public static void main(String[] args) throws Exception {
        //往文件中写对象
        //writeObject2File();
        //从文件中读取对象
        Singleton s1 = readObjectFromFile();
        Singleton s2 = readObjectFromFile();

        //判断两个反序列化后的对象是否是同一个对象
        System.out.println(s1 == s2);
    }

    private static Singleton readObjectFromFile() throws Exception {
        //创建对象输入流对象
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("C:\\Users\\Think\\Desktop\\a.txt"));
        //第一个读取Singleton对象
        Singleton instance = (Singleton) ois.readObject();

        return instance;
    }

    public static void writeObject2File() throws Exception {
        //获取Singleton类的对象
        Singleton instance = Singleton.getInstance();
        //创建对象输出流
        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("C:\\Users\\Think\\Desktop\\a.txt"));
        //将instance对象写出到文件中
        oos.writeObject(instance);
    }
}
```

> 上面代码运行结果是 **false** ， 表明序列化和反序列化已经破坏了单例设计模式。

- 反射

  Singleton类：

```
public class Singleton {

    //私有构造方法
    private Singleton() {}

    private static volatile Singleton instance;

```

```
    //对外提供静态方法获取该对象
    public static Singleton getInstance() {

        if(instance != null) {
            return instance;
        }

        synchronized (Singleton.class) {
            if(instance != null) {
                return instance;
            }
            instance = new Singleton();
            return instance;
        }
    }
}
```

Test类：

```
public class Test {
    public static void main(String[] args) throws Exception {
        //获取Singleton类的字节码对象
        Class clazz = Singleton.class;
        //获取Singleton类的私有无参构造方法对象
        Constructor constructor = clazz.getDeclaredConstructor();
        //取消访问检查
        constructor.setAccessible(true);

        //创建Singleton类的对象s1
        Singleton s1 = (Singleton) constructor.newInstance();
        //创建Singleton类的对象s2
        Singleton s2 = (Singleton) constructor.newInstance();

        //判断通过反射创建的两个Singleton对象是否是同一个对象
        System.out.println(s1 == s2);
    }
}
```

> 上面代码运行结果是 **false** ， 表明序列化和反序列化已经破坏了单例设计模式

> 注意：枚举方式不会出现这两个问题。

问题的解决

- 序列化、反序列方式破坏单例模式的解决方法

  在Singleton类中添加 **readResolve()** 方法，在反序列化时被反射调用，如果定义了这个方法，就返回这个方法的值，如果没有定义，则返回新new出来的对象。

  Singleton类：

```
public class Singleton implements Serializable {

    //私有构造方法
    private Singleton() {}

    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }

    /**
     * 下面是为了解决序列化反序列化破解单例模式
     */
    private Object readResolve() {
        return SingletonHolder.INSTANCE;
    }
}
```

  源码解析：

  ObjectInputStream类

```
public final Object readObject() throws IOException, ClassNotFoundException{
    ...
    // if nested read, passHandle contains handle of enclosing object
    int outerHandle = passHandle;
    try {
        Object obj = readObject0(false);//重点查看readObject0方法
    .....
}

private Object readObject0(boolean unshared) throws IOException {
    ...
    try {
        switch (tc) {
            ...
            case TC_OBJECT:
```

```
            return checkResolve(readOrdinaryObject(unshared));//重点查看readOrdinaryObject方法
            ...
        }
    } finally {
        depth--;
        bin.setBlockDataMode(oldMode);
    }
}

private Object readOrdinaryObject(boolean unshared) throws IOException {
    ...
    //isInstantiable 返回true，执行 desc.newInstance()，通过反射创建新的单例类，
    obj = desc.isInstantiable() ? desc.newInstance() : null;
    ...
    // 在Singleton类中添加 readResolve 方法后 desc.hasReadResolveMethod() 方法执行结果为true
    if (obj != null &&
        handles.lookupException(passHandle) == null &&
        desc.hasReadResolveMethod()) {
        // 通过反射调用 Singleton 类中的 readResolve 方法，将返回值赋值给rep变量
        // 这样多次调用ObjectInputStream类中的readObject方法，继而就会调用我们定义的readResolve方法，所以返回的是同一个对象。
        Object rep = desc.invokeReadResolve(obj);
        ...
    }
    return obj;
}
```

- 反射方式破解单例的解决方法

```
public class Singleton {

    //私有构造方法
    private Singleton() {
        /*
          反射破解单例模式需要添加的代码
        */
        if(instance != null) {
            throw new RuntimeException();
        }
    }

    private static volatile Singleton instance;

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {

        if(instance != null) {
            return instance;
```

```
        }

        synchronized (Singleton.class) {
            if(instance != null) {
                return instance;
            }
            instance = new Singleton();
            return instance;
        }
    }
}
```

说明:

这种方式比较好理解。当通过反射方式调用构造方法进行创建创建时，直接抛异常。不运行此中操作。

JDK源码解析-Runtime类

Runtime类就是使用的单例设计模式。

1. 通过源代码查看使用的是哪儿种单例模式

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

    /**
     * Returns the runtime object associated with the current Java application.
     * Most of the methods of class <code>Runtime</code> are instance
     * methods and must be invoked with respect to the current runtime object.
     *
     * @return  the <code>Runtime</code> object associated with the current
     *          Java application.
     */
    public static Runtime getRuntime() {
        return currentRuntime;
    }

    /** Don't let anyone else instantiate this class */
    private Runtime() {}
    ...
}
```

从上面源代码中可以看出Runtime类使用的是恶汉式（静态属性）方式来实现单例模式的。

2. 使用Runtime类中的方法

```java
public class RuntimeDemo {
    public static void main(String[] args) throws IOException {
        //获取Runtime类对象
        Runtime runtime = Runtime.getRuntime();

        //返回 Java 虚拟机中的内存总量。
        System.out.println(runtime.totalMemory());
        //返回 Java 虚拟机试图使用的最大内存量。
        System.out.println(runtime.maxMemory());

        //创建一个新的进程执行指定的字符串命令，返回进程对象
        Process process = runtime.exec("ipconfig");
        //获取命令执行后的结果，通过输入流获取
        InputStream inputStream = process.getInputStream();
        byte[] arr = new byte[1024 * 1024* 100];
        int b = inputStream.read(arr);
        System.out.println(new String(arr,0,b,"gbk"));
    }
}
```

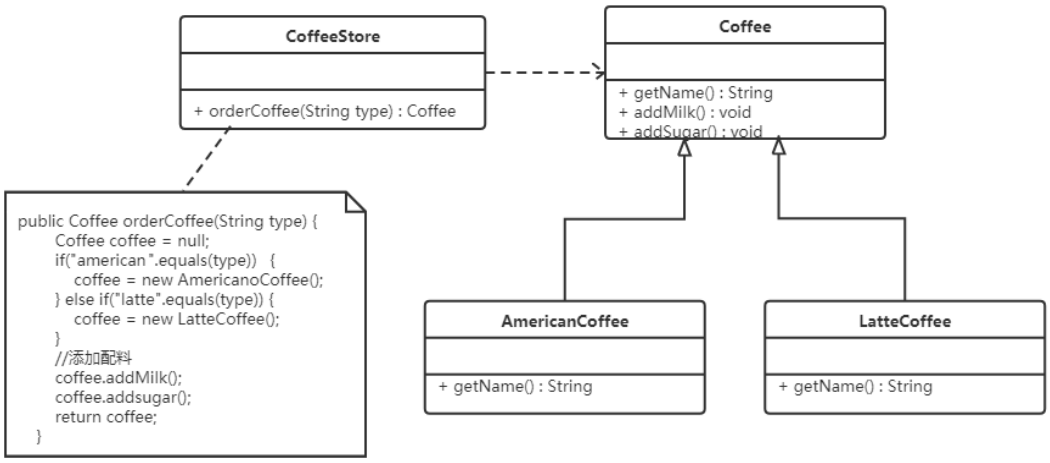
工厂模式

概述

需求：设计一个咖啡店点餐系统。

设计一个咖啡类（Coffee），并定义其两个子类（美式咖啡【AmericanCoffee】和拿铁咖啡【LatteCoffee】）；再设计一个咖啡店类（CoffeeStore），咖啡店具有点咖啡的功能。

具体类的设计如下：



在java中，万物皆对象，这些对象都需要创建，如果创建的时候直接new该对象，就会对该对象耦合严重，假如我们要更换对象，所有new对象的地方都需要修改一遍，这显然违背了软件设计的开闭原则。如果我们使用工厂来生产对象，我们就只和工厂打交道就可以了，彻底和对象解耦，如果要更换对象，直接在工厂里更换该对象即可，达到了与对象解耦的目的；所以说，工厂模式最大的优点就是：**解耦。**

在本教程中会介绍三种工厂的使用

- 简单工厂模式（不属于GOF的23种经典设计模式）
- 工厂方法模式
- 抽象工厂模式

简单工厂模式

简单工厂不是一种设计模式，反而比较像是一种编程习惯。

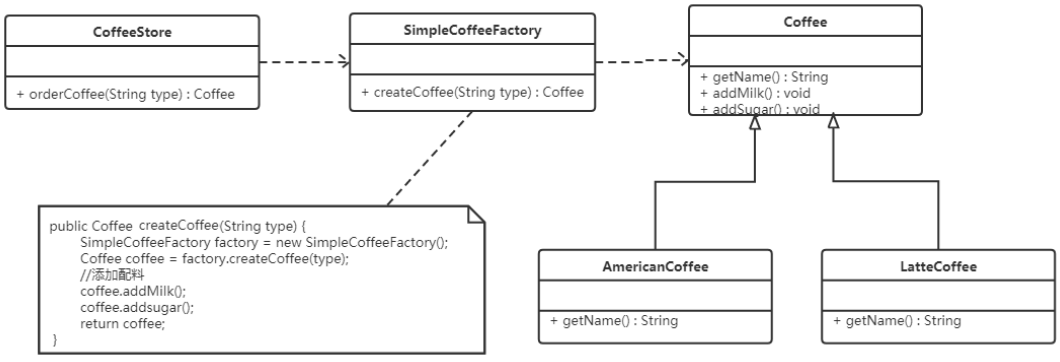
结构

简单工厂包含如下角色：

- 抽象产品 ： 定义了产品的规范，描述了产品的主要特性和功能。
- 具体产品 ： 实现或者继承抽象产品的子类
- 具体工厂 ： 提供了创建产品的方法，调用者通过该方法来获取产品。

实现

现在使用简单工厂对上面案例进行改进，类图如下：



工厂类代码如下：

```java
public class SimpleCoffeeFactory {

    public Coffee createCoffee(String type) {
        Coffee coffee = null;
        if("americano".equals(type)) {
            coffee = new AmericanoCoffee();
        } else if("latte".equals(type)) {
            coffee = new LatteCoffee();
        }
        return coffee;
    }
}
```

工厂（factory）处理创建对象的细节，一旦有了SimpleCoffeeFactory，CoffeeStore类中的orderCoffee()就变成此对象的客户，后期如果需要Coffee对象直接从工厂中获取即可。这样也就解除了和Coffee实现类的耦合，同时又产生了新的耦合，CoffeeStore对象和SimpleCoffeeFactory工厂对象的耦合，工厂对象和商品对象的耦合。

后期如果再加新品种的咖啡，我们势必要需求修改SimpleCoffeeFactory的代码，违反了开闭原则。工厂类的客户端可能有很多，比如创建美团外卖等，这样只需要修改工厂类的代码，省去其他的修改操作。

优缺点

优点：

封装了创建对象的过程，可以通过参数直接获取对象。把对象的创建和业务逻辑层分开，这样以后就避免了修改客户代码，如果要实现新产品直接修改工厂类，而不需要在原代码中修改，这样就降低了客户代码修改的可能性，更加容易扩展。

缺点：

增加新产品时还是需要修改工厂类的代码，违背了“开闭原则”。

扩展

静态工厂

在开发中也有一部分人将工厂类中的创建对象的功能定义为静态的，这个就是静态工厂模式，它也不是23种设计模式中的。代码如下：

```java
public class SimpleCoffeeFactory {

    public static Coffee createCoffee(String type) {
        Coffee coffee = null;
        if("americano".equals(type)) {
            coffee = new AmericanoCoffee();
        } else if("latte".equals(type)) {
            coffee = new LatteCoffee();
        }
        return coffe;
    }
}
```

工厂方法模式

针对上例中的缺点，使用工厂方法模式就可以完美的解决，完全遵循开闭原则。

概念

定义一个用于创建对象的接口，让子类决定实例化哪个产品类对象。工厂方法使一个产品类的实例化延迟到其工厂的子类。

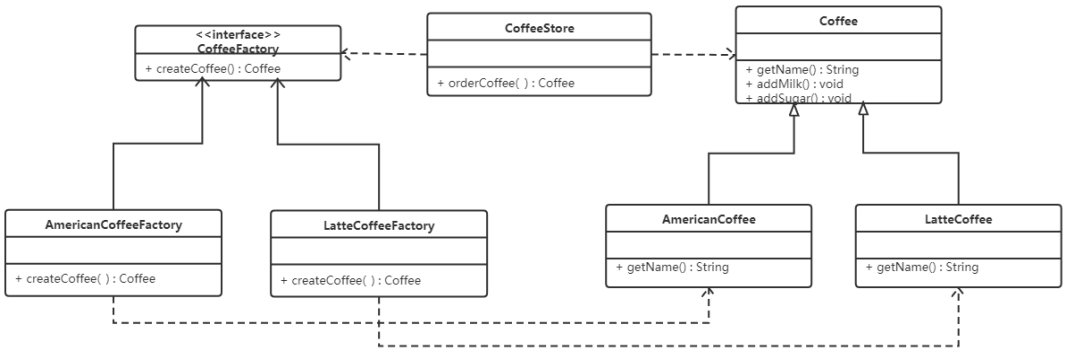
结构

工厂方法模式的主要角色：

- 抽象工厂（Abstract Factory）：提供了创建产品的接口，调用者通过它访问具体工厂的工厂方法来创建产品。
- 具体工厂（ConcreteFactory）：主要是实现抽象工厂中的抽象方法，完成具体产品的创建。
- 抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能。
- 具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应。

实现

使用工厂方法模式对上例进行改进，类图如下：



代码如下：

抽象工厂：

```
public interface CoffeeFactory {

    Coffee createCoffee();
}
```

具体工厂：

```
public class LatteCoffeeFactory implements CoffeeFactory {

    public Coffee createCoffee() {
        return new LatteCoffee();
    }
}

public class AmericanCoffeeFactory implements CoffeeFactory {

    public Coffee createCoffee() {
        return new AmericanCoffee();
    }
}
```

咖啡店类：

```
public class CoffeeStore {

    private CoffeeFactory factory;

    public CoffeeStore(CoffeeFactory factory) {
        this.factory = factory;
    }

    public Coffee orderCoffee(String type) {
        Coffee coffee = factory.createCoffee();
        coffee.addMilk();
        coffee.addsugar();
```

```
        return coffee;
    }
}
```

从以上的编写的代码可以看到，要增加产品类时也要相应地增加工厂类，不需要修改工厂类的代码了，这样就解决了简单工厂模式的缺点。

工厂方法模式是简单工厂模式的进一步抽象。由于使用了多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。

优缺点

优点：

- 用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程；
- 在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，满足开闭原则；

缺点：

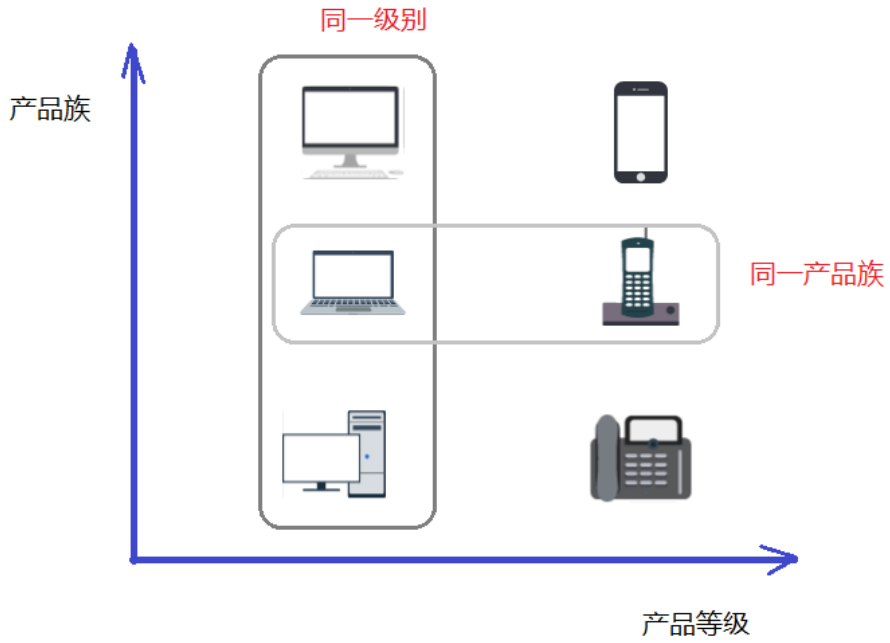
- 每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度。

抽象工厂模式

前面介绍的工厂方法模式中考虑的是一类产品的生产，如畜牧场只养动物、电视机厂只生产电视机、传智播客只培养计算机软件专业的学生等。

这些工厂只生产同种类产品，同种类产品称为同等级产品，也就是说：工厂方法模式只考虑生产同等级的产品，但是在现实生活中许多工厂是综合型的工厂，能生产多等级（种类） 的产品，如电器厂既生产电视机又生产洗衣机或空调，大学既有软件专业又有生物专业等。

本节要介绍的抽象工厂模式将考虑多等级产品的生产，将同一个具体工厂所生产的位于不同等级的一组产品称为一个产品族，下图所示横轴是产品等级，也就是同一类产品；纵轴是产品族，也就是同一品牌的产品，同一品牌的产品产自同一个工厂。



同一产品族

同一产品族

概念

是一种为访问类提供一个创建一组相关或相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构。

抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。

结构

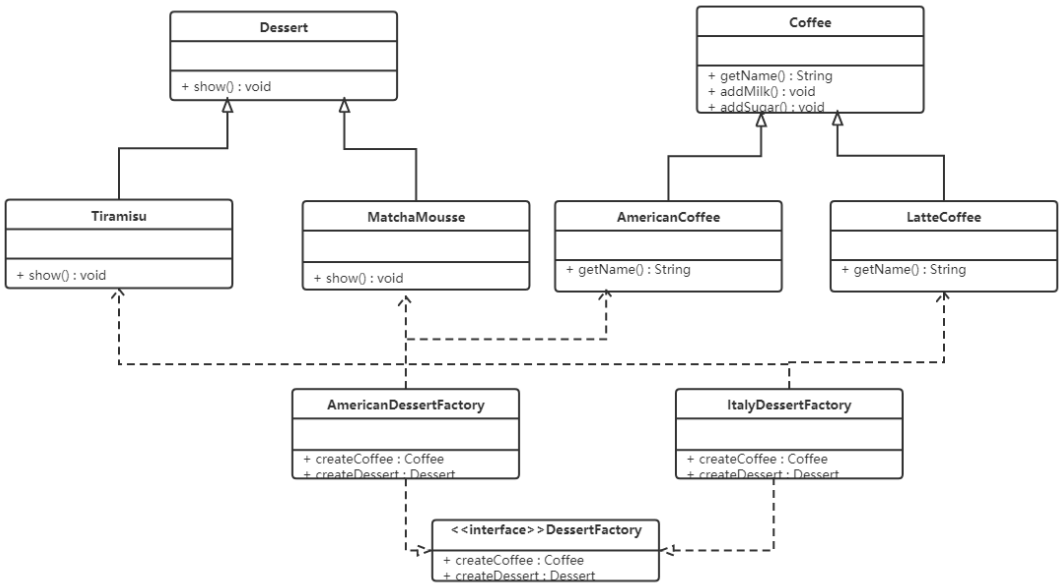
抽象工厂模式的主要角色如下：

- 抽象工厂（Abstract Factory）：提供了创建产品的接口，它包含多个创建产品的方法，可以创建多个不同等级的产品。
- 具体工厂（Concrete Factory）：主要是实现抽象工厂中的多个抽象方法，完成具体产品的创建。

- 抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能，抽象工厂模式有多个抽象产品。
- 具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它 同具体工厂之间是多对一的关系。

实现

现咖啡店业务发生改变，不仅要生产咖啡还要生产甜点，如提拉米苏、抹茶慕斯等，要是按照工厂方法模式，需要定义提拉米苏类、抹茶慕斯类、提拉米苏工厂、抹茶慕斯工厂、甜点工厂类，很容易发生类爆炸情况。其中拿铁咖啡、美式咖啡是一个产品等级，都是咖啡；提拉米苏、抹茶慕斯也是一个产品等级；拿铁咖啡和提拉米苏是同一产品族（也就是都属于意大利风味），美式咖啡和抹茶慕斯是同一产品族（也就是都属于美式风味）。所以这个案例可以使用抽象工厂模式实现。类图如下：



代码如下：

抽象工厂：

```
public interface DessertFactory {

    Coffee createCoffee();

    Dessert createDessert();
}
```

具体工厂：

```
//美式甜点工厂
public class AmericanDessertFactory implements DessertFactory {

    public Coffee createCoffee() {
        return new AmericanCoffee();
```

```
    }

    public Dessert createDessert() {
        return new MatchaMousse();
    }
}
//意大利风味甜点工厂
public class ItalyDessertFactory implements DessertFactory {

    public Coffee createCoffee() {
        return new LatteCoffee();
    }

    public Dessert createDessert() {
        return new Tiramisu();
    }
}
```

如果要加同一个产品族的话，只需要再加一个对应的工厂类即可，不需要修改其他的类。

优缺点

优点：

当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

缺点：

当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。

使用场景

- 当需要创建的对象是一系列相互关联或相互依赖的产品族时，如电器工厂中的电视机、洗衣机、空调等。
- 系统中有多个产品族，但每次只使用其中的某一族产品。如有人只喜欢穿某一个品牌的衣服和鞋。
- 系统中提供了产品的类库，且所有产品的接口相同，客户端不依赖产品实例的创建细节和内部结构。

如：输入法换皮肤，一整套一起换。生成不同操作系统的程序。

模式扩展

简单工厂+配置文件解除耦合

可以通过工厂模式+配置文件的方式解除工厂对象和产品对象的耦合。在工厂类中加载配置文件中的全类名，并创建对象进行存储，客户端如果需要对象，直接进行获取即可。

第一步：定义配置文件

为了演示方便，我们使用properties文件作为配置文件，名称为bean.properties

```
american=com.itheima.pattern.factory.config_factory.AmericanCoffee
latte=com.itheima.pattern.factory.config_factory.LatteCoffee
```

第二步：改进工厂类

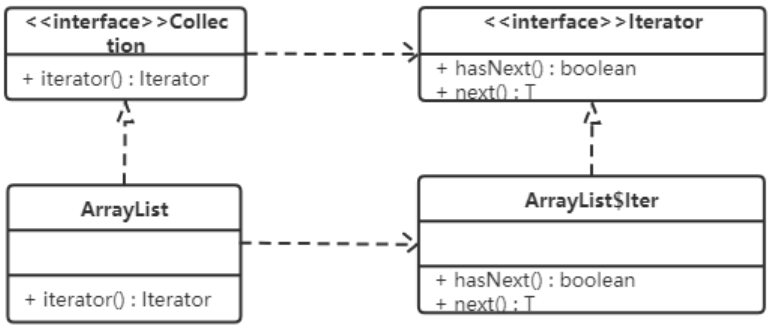
```
public class CoffeeFactory {

    private static Map<String,Coffee> map = new HashMap();

    static {
        Properties p = new Properties();
        InputStream is =
CoffeeFactory.class.getClassLoader().getResourceAsStream("bean.properties");
        try {
            p.load(is);
            //遍历Properties集合对象
            Set<Object> keys = p.keySet();
            for (Object key : keys) {
                //根据键获取值（全类名）
                String className = p.getProperty((String) key);
                //获取字节码对象
                Class clazz = Class.forName(className);
                Coffee obj = (Coffee) clazz.newInstance();
                map.put((String)key,obj);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static Coffee createCoffee(String name) {

        return map.get(name);
    }
}
```

静态成员变量用来存储创建的对象（键存储的是名称，值存储的是对应的对象），而读取配置文件以及创建对象写在静态代码块中，目的就是只需要执行一次。

JDK源码解析-Collection.iterator方法

```
public class Demo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("令狐冲");
        list.add("风清扬");
        list.add("任我行");

        //获取迭代器对象
        Iterator<String> it = list.iterator();
        //使用迭代器遍历
        while(it.hasNext()) {
            String ele = it.next();
            System.out.println(ele);
        }
    }
}
```

对上面的代码大家应该很熟，使用迭代器遍历集合，获取集合中的元素。而单列集合获取迭代器的方法就使用到了工厂方法模式。我们看通过类图看看结构：



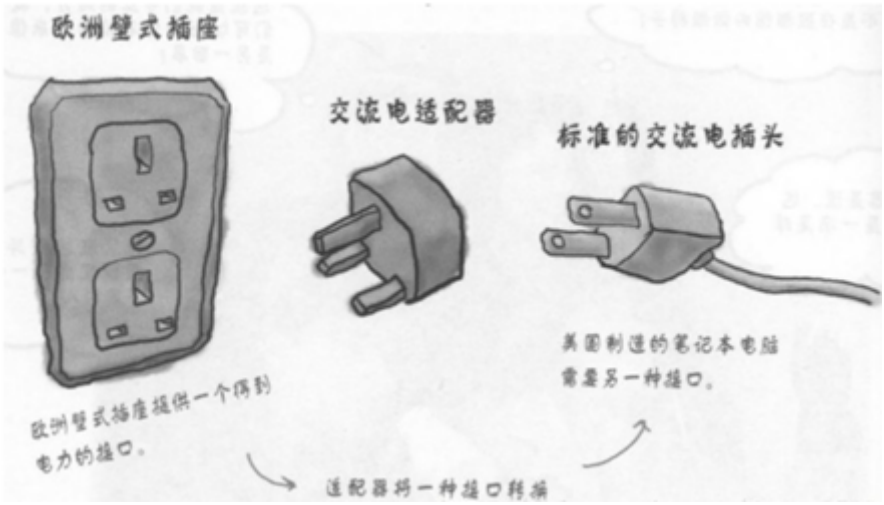
Collection接口是抽象工厂类，ArrayList是具体的工厂类；Iterator接口是抽象商品类，ArrayList类中的Iter内部类是具体的商品类。在具体的工厂类中iterator()方法创建具体的商品类的对象。

> 另：
> 1,DateForamt类中的getInstance()方法使用的是工厂模式；
> 2,Calendar类中的getInstance()方法使用的是工厂模式；

适配器模式

概述

如果去欧洲国家去旅游的话，他们的插座如下图最左边，是欧洲标准。而我们使用的插头如下图最右边的。因此我们的笔记本电脑，手机在当地不能直接充电。所以就需要一个插座转换器，转换器第1面插入当地的插座，第2面供我们充电，这样使得我们的插头在当地能使用。生活中这样的例子很多，手机充电器（将220v转换为5v的电压），读卡器等，其实就是使用到了适配器模式。



定义：

将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

适配器模式分为类适配器模式和对象适配器模式，前者类之间的耦合度比后者高，且要求程序员了解现有组件库中的相关组件的内部结构，所以应用相对较少些。

结构

适配器模式（Adapter）包含以下主要角色：

- 目标（Target）接口：当前系统业务所期待的接口，它可以是抽象类或接口。
- 适配者（Adaptee）类：它是被访问和适配的现存组件库中的组件接口。
- 适配器（Adapter）类：它是一个转换器，通过继承或引用适配者的对象，把适配者接口转换成目标接口，让客户按目标接口的格式访问适配者。

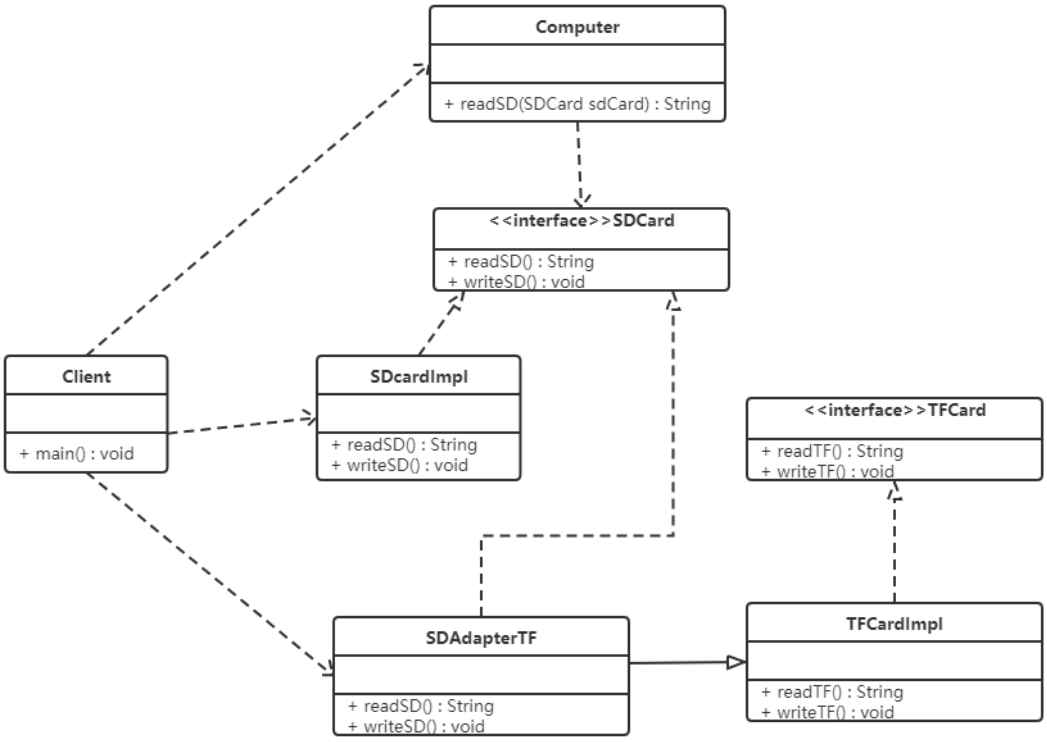
类适配器模式

实现方式：定义一个适配器类来实现当前系统的业务接口，同时又继承现有组件库中已经存在的组件。

【例】读卡器

现有一台电脑只能读取SD卡，而要读取TF卡中的内容的话就需要使用到适配器模式。创建一个读卡器，将TF卡中的内容读取出来。

类图如下：



代码如下：

```
//SD卡的接口
public interface SDCard {
    //读取SD卡方法
    String readSD();
    //写入SD卡功能
    void writeSD(String msg);
}

//SD卡实现类
public class SDCardImpl implements SDCard {
    public String readSD() {
        String msg = "sd card read a msg :hello word SD";
        return msg;
    }
```

```
15
16      public void writeSD(String msg) {
17          System.out.println("sd card write msg : " + msg);
18      }
19  }
20
21  //电脑类
22  public class Computer {
23
24      public String readSD(SDCard sdCard) {
25          if(sdCard == null) {
26              throw new NullPointerException("sd card null");
27          }
28          return sdCard.readSD();
29      }
30  }
31
32  //TF卡接口
33  public interface TFCard {
34      //读取TF卡方法
35      String readTF();
36      //写入TF卡功能
37      void writeTF(String msg);
38  }
39
40  //TF卡实现类
41  public class TFCardImpl implements TFCard {
42
43      public String readTF() {
44          String msg ="tf card read msg : hello word tf card";
45          return msg;
46      }
47
48      public void writeTF(String msg) {
49          System.out.println("tf card write a msg : " + msg);
50      }
51  }
52
53  //定义适配器类（SD兼容TF）
54  public class SDAdapterTF extends TFCardImpl implements SDCard {
55
56      public String readSD() {
57          System.out.println("adapter read tf card ");
58          return readTF();
59      }
60
61      public void writeSD(String msg) {
62          System.out.println("adapter write tf card");
```

```
        writeTF(msg);
    }
}

//测试类
public class Client {
    public static void main(String[] args) {
        Computer computer = new Computer();
        SDCard sdCard = new SDCardImpl();
        System.out.println(computer.readSD(sdCard));

        System.out.println("------------");

        SDAdapterTF adapter = new SDAdapterTF();
        System.out.println(computer.readSD(adapter));
    }
}
```

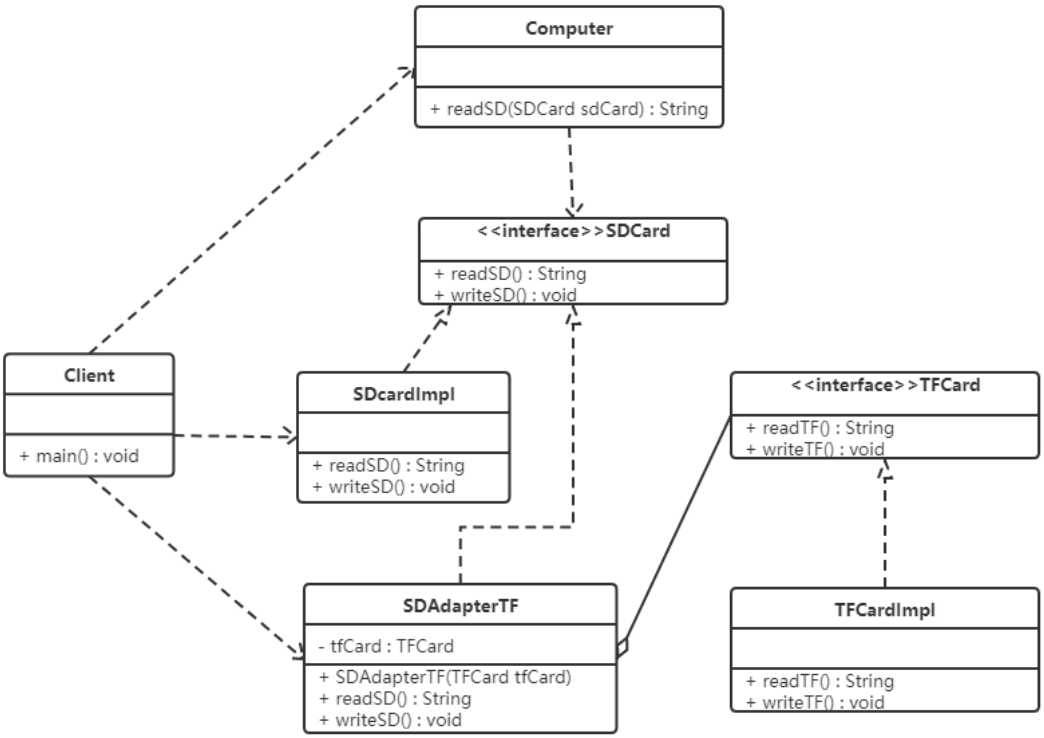
类适配器模式违背了合成复用原则。类适配器是客户类有一个接口规范的情况下可用，反之不可用。

对象适配器模式

实现方式：对象适配器模式可采用将现有组件库中已经实现的组件引入适配器类中，该类同时实现当前系统的业务接口。

【例】读卡器

我们使用对象适配器模式将读卡器的案例进行改写。类图如下：



代码如下：

类适配器模式的代码，我们只需要修改适配器类（SDAdapterTF）和测试类。

```
//创建适配器对象（SD兼容TF）
public class SDAdapterTF  implements SDCard {

    private TFCard tfCard;

    public SDAdapterTF(TFCard tfCard) {
        this.tfCard = tfCard;
    }

    public String readSD() {
        System.out.println("adapter read tf card ");
        return tfCard.readTF();
    }

    public void writeSD(String msg) {
        System.out.println("adapter write tf card");
        tfCard.writeTF(msg);
    }
}

//测试类
public class Client {
    public static void main(String[] args) {
        Computer computer = new Computer();
```

```
        SDCard sdCard = new SDCardImpl();
        System.out.println(computer.readSD(sdCard));

        System.out.println("------------");

        TFCard tfCard = new TFCardImpl();
        SDAdapterTF adapter = new SDAdapterTF(tfCard);
        System.out.println(computer.readSD(adapter));
    }
}
```

> 注意：还有一个适配器模式是接口适配器模式。当不希望实现一个接口中所有的方法时，可以创建一个抽象类Adapter ，实现所有方法。而此时我们只需要继承该抽象类即可。

应用场景

- 以前开发的系统存在满足新系统功能需求的类，但其接口同新系统的接口不一致。
- 使用第三方提供的组件，但组件接口定义和自己要求的接口定义不同。

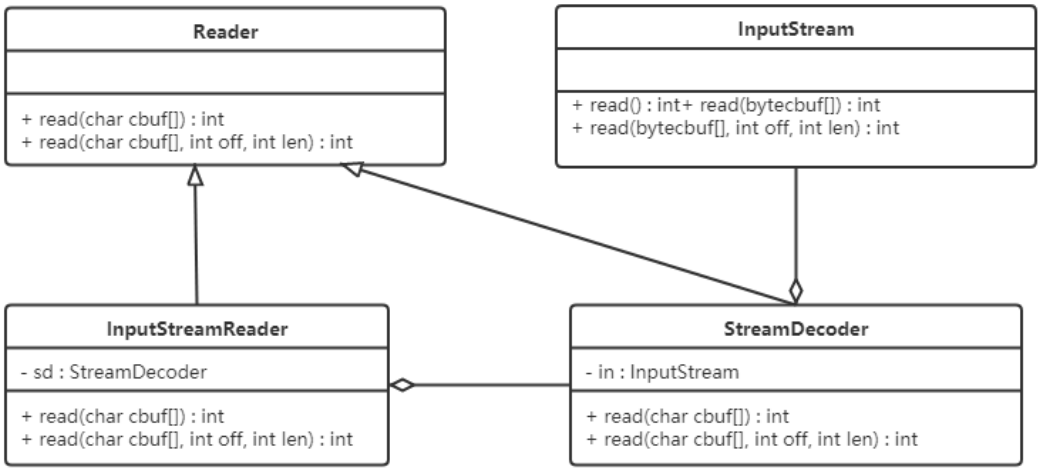
JDK源码解析

Reader（字符流）、InputStream（字节流）的适配使用的是InputStreamReader。

InputStreamReader继承自java.io包中的Reader，对他中的抽象的未实现的方法给出实现。如：

```
public int read() throws IOException {
    return sd.read();
}

public int read(char cbuf[], int offset, int length) throws IOException {
    return sd.read(cbuf, offset, length);
}
```

如上代码中的sd（StreamDecoder类对象），在Sun的JDK实现中，实际的方法实现是对sun.nio.cs.StreamDecoder类的同名方法的调用封装。类结构图如下：



从上图可以看出：

- InputStreamReader是对同样实现了Reader的StreamDecoder的封装。
- StreamDecoder不是Java SE API中的内容，是Sun JDK给出的自身实现。但我们知道他们对构造方法中的字节流类（InputStream）进行封装，并通过该类进行了字节流和字符流之间的解码转换。

结论：

从表层来看，InputStreamReader做了InputStream字节流类到Reader字符流之间的转换。而从如上Sun JDK中的实现类关系结构中可以看出，是StreamDecoder的设计实现在实际上采用了适配器模式。

装饰者模式

概述

我们先来看一个快餐店的例子。

快餐店有炒面、炒饭这些快餐，可以额外附加鸡蛋、火腿、培根这些配菜，当然加配菜需要额外加钱，每个配菜的价钱通常不太一样，那么计算总价就会显得比较麻烦。

使用继承的方式存在的问题：

- 扩展性不好

  如果要再加一种配料（火腿肠），我们就会发现需要给FriedRice和FriedNoodles分别定义一个子类。如果要新增一个快餐品类（炒河粉）的话，就需要定义更多的子类。

- 产生过多的子类

定义：

指在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即增加其额外功能）的模式。

结构

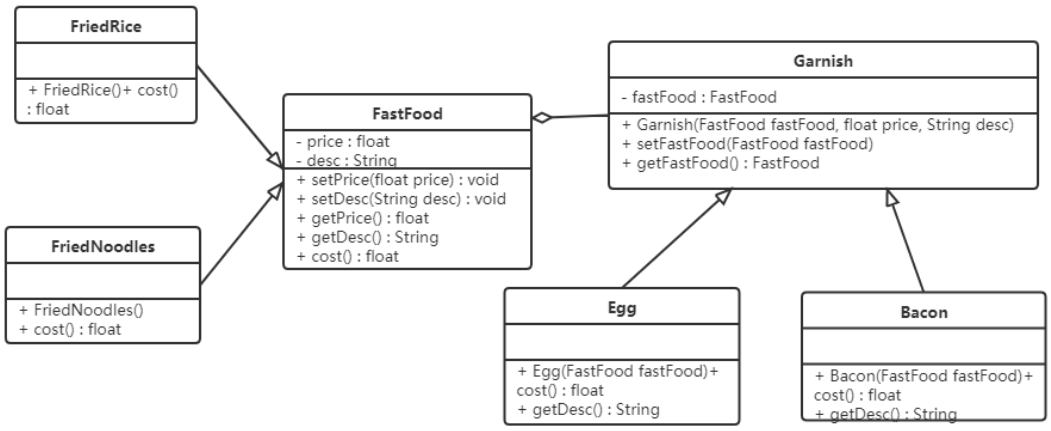
装饰（Decorator）模式中的角色：

- 抽象构件（Component）角色 ： 定义一个抽象接口以规范准备接收附加责任的对象。
- 具体构件（Concrete Component）角色 ： 实现抽象构件，通过装饰角色为其添加一些职责。
- 抽象装饰（Decorator）角色 ： 继承或实现抽象构件，并包含具体构件的实例，可以通过其子类扩展具体构件的功能。
- 具体装饰（ConcreteDecorator）角色 ： 实现抽象装饰的相关方法，并给具体构件对象添加附加的责任。

案例

我们使用装饰者模式对快餐店案例进行改进，体会装饰者模式的精髓。

类图如下：



代码如下：

```
//快餐接口
public abstract class FastFood {
    private float price;
    private String desc;

    public FastFood() {
    }

    public FastFood(float price, String desc) {
```

```
        this.price = price;
        this.desc = desc;
    }

    public void setPrice(float price) {
        this.price = price;
    }

    public float getPrice() {
        return price;
    }

    public String getDesc() {
        return desc;
    }

    public void setDesc(String desc) {
        this.desc = desc;
    }

    public abstract float cost();  //获取价格
}

//炒饭
public class FriedRice extends FastFood {

    public FriedRice() {
        super(10, "炒饭");
    }

    public float cost() {
        return getPrice();
    }
}

//炒面
public class FriedNoodles extends FastFood {

    public FriedNoodles() {
        super(12, "炒面");
    }

    public float cost() {
        return getPrice();
    }
}

//配料类
```

```java
public abstract class Garnish extends FastFood {

    private FastFood fastFood;

    public FastFood getFastFood() {
        return fastFood;
    }

    public void setFastFood(FastFood fastFood) {
        this.fastFood = fastFood;
    }

    public Garnish(FastFood fastFood, float price, String desc) {
        super(price,desc);
        this.fastFood = fastFood;
    }
}

//鸡蛋配料
public class Egg extends Garnish {

    public Egg(FastFood fastFood) {
        super(fastFood,1,"鸡蛋");
    }

    public float cost() {
        return getPrice() + getFastFood().getPrice();
    }

    @Override
    public String getDesc() {
        return super.getDesc() + getFastFood().getDesc();
    }
}

//培根配料
public class Bacon extends Garnish {

    public Bacon(FastFood fastFood) {

        super(fastFood,2,"培根");
    }

    @Override
    public float cost() {
        return getPrice() + getFastFood().getPrice();
    }

```

```
    @Override
    public String getDesc() {
        return super.getDesc() + getFastFood().getDesc();
    }
}

//测试类
public class Client {
    public static void main(String[] args) {
        //点一份炒饭
        FastFood food = new FriedRice();
        //花费的价格
        System.out.println(food.getDesc() + " " + food.cost() + "元");

        System.out.println("========");
        //点一份加鸡蛋的炒饭
        FastFood food1 = new FriedRice();

        food1 = new Egg(food1);
        //花费的价格
        System.out.println(food1.getDesc() + " " + food1.cost() + "元");

        System.out.println("========");
        //点一份加培根的炒面
        FastFood food2 = new FriedNoodles();
        food2 = new Bacon(food2);
        //花费的价格
        System.out.println(food2.getDesc() + " " + food2.cost() + "元");
    }
}
```

好处：

- 饰者模式可以带来比继承更加灵活性的扩展功能，使用更加方便，可以通过组合不同的装饰者对象来获取具有不同行为状态的多样化的结果。装饰者模式比继承更具良好的扩展性，完美的遵循开闭原则，继承是静态的附加责任，装饰者则是动态的附加责任。
- 装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

使用场景

- 当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。

  不能采用继承的情况主要有两类：

  - 第一类是系统中存在大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长；

    - 第二类是因为类定义不能继承（如final类）
- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 当对象的功能要求可以动态地添加，也可以再动态地撤销时。

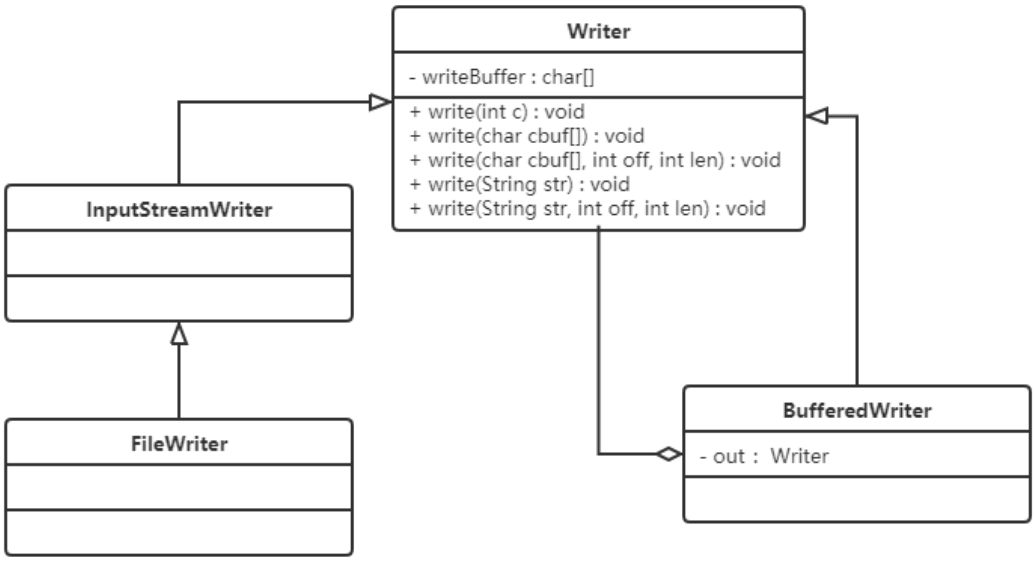
JDK源码解析

IO流中的包装类使用到了装饰者模式。BufferedInputStream，BufferedOutputStream，BufferedReader，BufferedWriter。

我们以BufferedWriter举例来说明，先看看如何使用BufferedWriter

```
public class Demo {
    public static void main(String[] args) throws Exception{
        //创建BufferedWriter对象
        //创建FileWriter对象
        FileWriter fw = new FileWriter("C:\\Users\\Think\\Desktop\\a.txt");
        BufferedWriter bw = new BufferedWriter(fw);

        //写数据
        bw.write("hello Buffered");

        bw.close();
    }
}
```

使用起来感觉确实像是装饰者模式，接下来看它们的结构：



> 小结：

> BufferedWriter使用装饰者模式对Writer子实现类进行了增强，添加了缓冲区，提高了写数据的效率。

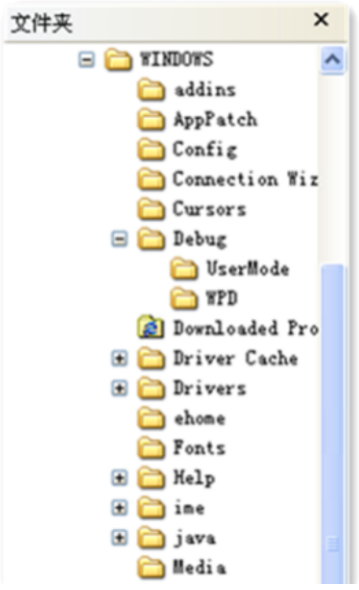
代理和装饰者的区别

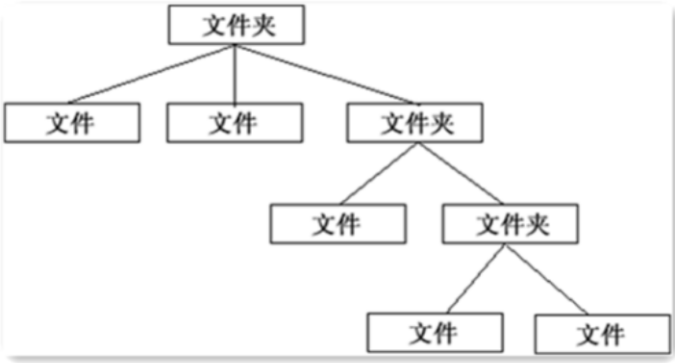
静态代理和装饰者模式的区别：

- 相同点：
  - 都要实现与目标类相同的业务接口
  - 在两个类中都要声明目标对象
  - 都可以在不修改目标类的前提下增强目标方法
- 不同点：
  - 目的不同
    装饰者是为了增强目标对象
    静态代理是为了保护和隐藏目标对象
  - 获取目标对象构建的地方不同
    装饰者是由外界传递进来，可以通过构造方法传递
    静态代理是在代理类内部创建，以此来隐藏目标对象

组合模式

概述





对于这个图片肯定会非常熟悉，上图我们可以看做是一个文件系统，对于这样的结构我们称之为树形结构。在树形结构中可以通过调用某个方法来遍历整个树，当我们找到某个叶子节点后，就可以对叶子节点进行相关的操作。可以将这颗树理解成一个大的容器，容器里面包含很多的成员对象，这些成员对象即可是容器对象也可以是叶子对象。但是由于容器对象和叶子对象在功能上面的区别，使得我们在使用的过程中必须要区分容器对象和叶子对象，但是这样就会给客户带来不必要的麻烦，作为客户而已，它始终希望能够一致的对待容器对象和叶子对象。

定义：

又名部分整体模式，是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。

结构

组合模式主要包含三种角色：

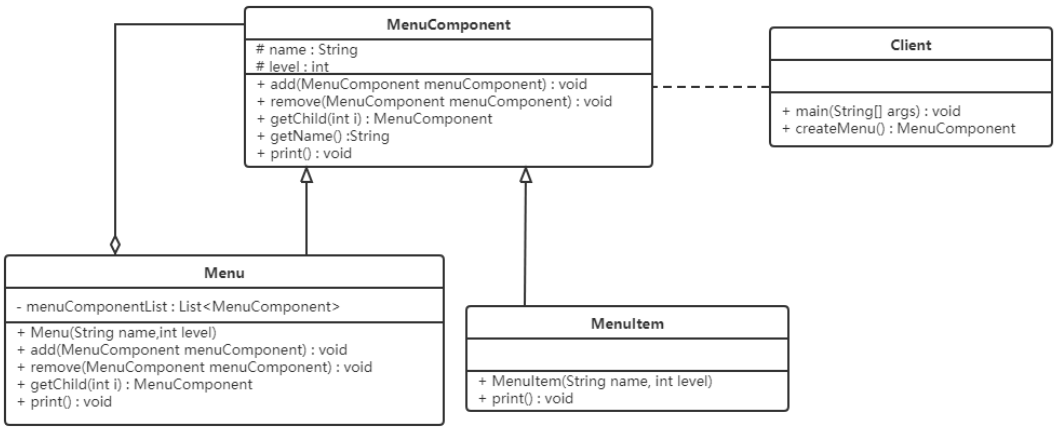
- 抽象根节点（Component）：定义系统各层次对象的共有方法和属性，可以预先定义一些默认行为和属性。
- 树枝节点（Composite）：定义树枝节点的行为，存储子节点，组合树枝节点和叶子节点形成一个树形结构。
- 叶子节点（Leaf）：叶子节点对象，其下再无分支，是系统层次遍历的最小单位。

案例实现

【例】软件菜单

如下图，我们在访问别的一些管理系统时，经常可以看到类似的菜单。一个菜单可以包含菜单项（菜单项是指不再包含其他内容的菜单条目），也可以包含带有其他菜单项的菜单，因此使用组合模式描述菜单就很恰当，我们的需求是针对一个菜单，打印出其包含的所有菜单以及菜单项的名称。

要实现该案例，我们先画出类图：



代码实现：

不管是菜单还是菜单项，都应该继承自统一的接口，这里姑且将这个统一的接口称为菜单组件。

```
//菜单组件  不管是菜单还是菜单项，都应该继承该类
public abstract class MenuComponent {

    protected String name;
    protected int level;

    //添加菜单
    public void add(MenuComponent menuComponent){
        throw new UnsupportedOperationException();
    }

```

```
    //移除菜单
    public void remove(MenuComponent menuComponent){
        throw new UnsupportedOperationException();
    }

    //获取指定的子菜单
    public MenuComponent getChild(int i){
        throw new UnsupportedOperationException();
    }

    //获取菜单名称
    public String getName(){
        return name;
    }

    public void print(){
        throw new UnsupportedOperationException();
    }
}
```

这里的MenuComponent定义为抽象类，因为有一些共有的属性和行为要在该类中实现，Menu和MenuItem类就可以只覆盖自己感兴趣的方法，而不用搭理不需要或者不感兴趣的方法，举例来说，Menu类可以包含子菜单，因此需要覆盖add()、remove()、getChild()方法，但是MenuItem就不应该有这些方法。这里给出的默认实现是抛出异常，你也可以根据自己的需要改写默认实现。

```
public class Menu extends MenuComponent {

    private List<MenuComponent> menuComponentList;

    public Menu(String name,int level){
        this.level = level;
        this.name = name;
        menuComponentList = new ArrayList<MenuComponent>();
    }

    @Override
    public void add(MenuComponent menuComponent) {
        menuComponentList.add(menuComponent);
    }

    @Override
    public void remove(MenuComponent menuComponent) {
        menuComponentList.remove(menuComponent);
    }

    @Override
    public MenuComponent getChild(int i) {
```

```
        return menuComponentList.get(i);
    }

    @Override
    public void print() {

        for (int i = 1; i < level; i++) {
            System.out.print("--");
        }
        System.out.println(name);
        for (MenuComponent menuComponent : menuComponentList) {
            menuComponent.print();
        }
    }
}
```

Menu类已经实现了除了getName方法的其他所有方法，因为Menu类具有添加菜单，移除菜单和获取子菜单的功能。

```
public class MenuItem extends MenuComponent {

    public MenuItem(String name,int level) {
        this.name = name;
        this.level = level;
    }

    @Override
    public void print() {
        for (int i = 1; i < level; i++) {
            System.out.print("--");
        }
        System.out.println(name);
    }
}
```

MenuItem是菜单项，不能再有子菜单，所以添加菜单，移除菜单和获取子菜单的功能并不能实现。

组合模式的分类

在使用组合模式时，根据抽象构件类的定义形式，我们可将组合模式分为透明组合模式和安全组合模式两种形式。

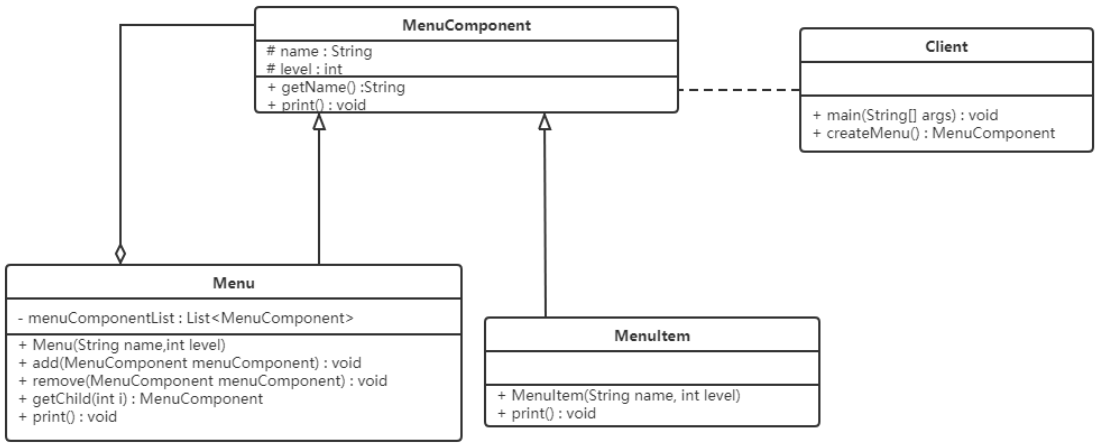
- 透明组合模式

透明组合模式中，抽象根节点角色中声明了所有用于管理成员对象的方法，比如在示例中 `MenuComponent` 声明了 `add` 、 `remove` 、 `getChild` 方法，这样做的好处是确保所有的构件类都有相同的接口。透明组合模式也是组合模式的标准形式。

透明组合模式的缺点是不够安全，因为叶子对象和容器对象在本质上是有区别的，叶子对象不可能有下一个层次的对象，即不可能包含成员对象，因此为其提供 add()、remove() 等方法是没有意义的，这在编译阶段不会出错，但在运行阶段如果调用这些方法可能会出错（如果没有提供相应的错误处理代码）

- 安全组合模式

  在安全组合模式中，在抽象构件角色中没有声明任何用于管理成员对象的方法，而是在树枝节点 `Menu` 类中声明并实现这些方法。安全组合模式的缺点是不够透明，因为叶子构件和容器构件具有不同的方法，且容器构件中那些用于管理成员对象的方法没有在抽象构件类中定义，因此客户端不能完全针对抽象编程，必须有区别地对待叶子构件和容器构件。



优点

- 组合模式可以清楚地定义分层次的复杂对象，表示对象的全部或部分层次，它让客户端忽略了层次的差异，方便对整个层次结构进行控制。
- 客户端可以一致地使用一个组合结构或其中单个对象，不必关心处理的是单个对象还是整个组合结构，简化了客户端代码。
- 在组合模式中增加新的树枝节点和叶子节点都很方便，无须对现有类库进行任何修改，符合“开闭原则”。
- 组合模式为树形结构的面向对象实现提供了一种灵活的解决方案，通过叶子节点和树枝节点的递归组合，可以形成复杂的树形结构，但对树形结构的控制却非常简单。

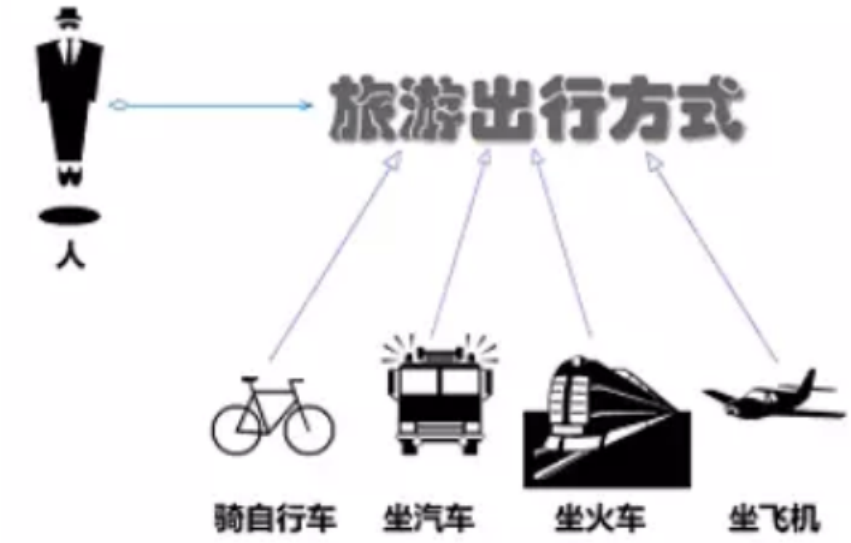
使用场景

组合模式正是应树形结构而生，所以组合模式的使用场景就是出现树形结构的地方。比如：文件目录显示，多级目录呈现等树形结构数据的操作。

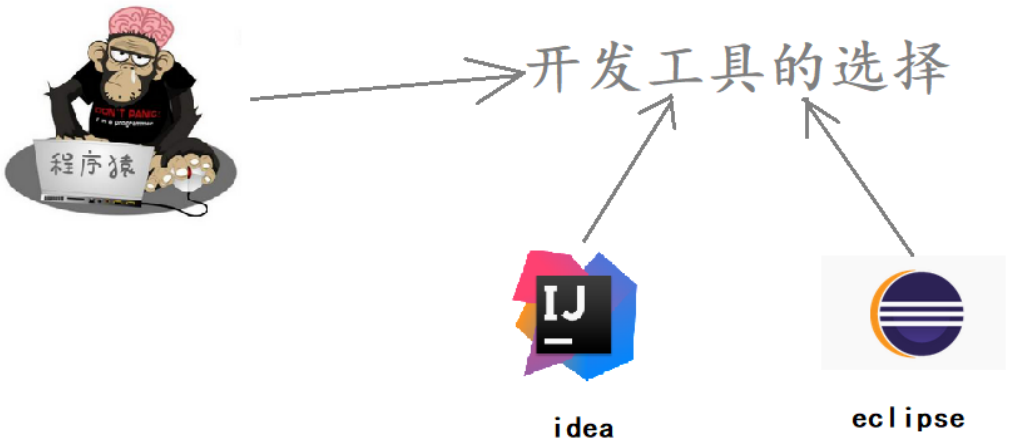
策略模式

概述

先看下面的图片，我们去旅游选择出行模式有很多种，可以骑自行车、可以坐汽车、可以坐火车、可以坐飞机。



作为一个程序猿，开发需要选择一款开发工具，当然可以进行代码开发的工具有很多，可以选择Idea进行开发，也可以使用eclipse进行开发，也可以使用其他的一些开发工具。



定义：

该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

结构

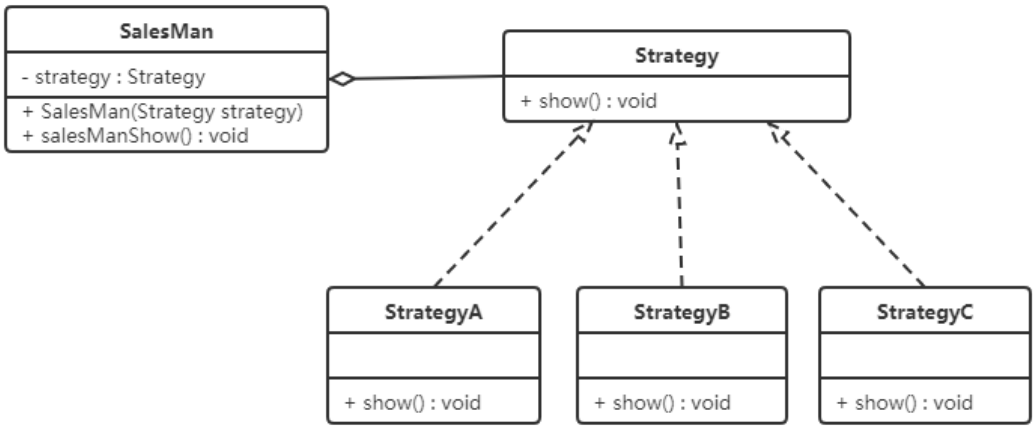
策略模式的主要角色如下：

- 抽象策略（Strategy）类：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。
- 具体策略（Concrete Strategy）类：实现了抽象策略定义的接口，提供具体的算法实现或行为。
- 环境（Context）类：持有一个策略类的引用，最终给客户端调用。

案例实现

【例】促销活动

一家百货公司在定年度的促销活动。针对不同的节日（春节、中秋节、圣诞节）推出不同的促销活动，由促销员将促销活动展示给客户。类图如下：



代码如下：

定义百货公司所有促销活动的共同接口

```
1  public interface Strategy {
2      void show();
3  }
```

定义具体策略角色（Concrete Strategy）：每个节日具体的促销活动

```java
//为春节准备的促销活动A
public class StrategyA implements Strategy {

    public void show() {
        System.out.println("买一送一");
    }
}

//为中秋准备的促销活动B
public class StrategyB implements Strategy {

    public void show() {
        System.out.println("满200元减50元");
    }
}

//为圣诞准备的促销活动C
public class StrategyC implements Strategy {

    public void show() {
        System.out.println("满1000元加一元换购任意200元以下商品");
    }
}
```

定义环境角色（Context）：用于连接上下文，即把促销活动推销给客户，这里可以理解为销售员

```java
public class SalesMan {
    //持有抽象策略角色的引用
    private Strategy strategy;

    public SalesMan(Strategy strategy) {
        this.strategy = strategy;
    }

    //向客户展示促销活动
    public void salesManShow(){
        strategy.show();
    }
}
```

优缺点

1，优点：

- 策略类之间可以自由切换

  由于策略类都实现同一个接口，所以使它们之间可以自由切换。

- 易于扩展

  增加一个新的策略只需要添加一个具体的策略类即可，基本不需要改变原有的代码，符合“开闭原则“

- 避免使用多重条件选择语句（if else），充分体现面向对象设计思想。

2，缺点：

- 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。
- 策略模式将造成产生很多策略类，可以通过使用享元模式在一定程度上减少对象的数量。

使用场景

- 一个系统需要动态地在几种算法中选择一种时，可将每个算法封装到策略类中。
- 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现，可将每个条件分支移入它们各自的策略类中以代替这些条件语句。
- 系统中各算法彼此完全独立，且要求对客户隐藏具体算法的实现细节时。
- 系统要求使用算法的客户不应该知道其操作的数据时，可使用策略模式来隐藏与算法相关的数据结构。
- 多个类只区别在表现行为不同，可以使用策略模式，在运行时动态选择具体要执行的行为。

JDK源码解析

`Comparator` 中的策略模式。在Arrays类中有一个 `sort()` 方法，如下：

```
public class Arrays{
    public static <T> void sort(T[] a, Comparator<? super T> c) {
        if (c == null) {
            sort(a);
        } else {
            if (LegacyMergeSort.userRequested)
                legacyMergeSort(a, c);
            else
                TimSort.sort(a, 0, a.length, c, null, 0, 0);
        }
    }
}
```

Arrays就是一个环境角色类，这个sort方法可以传一个新策略让Arrays根据这个策略来进行排序。就比如下面的测试类。

```
public class demo {
    public static void main(String[] args) {

        Integer[] data = {12, 2, 3, 2, 4, 5, 1};
        // 实现降序排序
        Arrays.sort(data, new Comparator<Integer>() {
            public int compare(Integer o1, Integer o2) {
                return o2 - o1;
            }
        });
        System.out.println(Arrays.toString(data)); //[12, 5, 4, 3, 2, 2, 1]
    }
}
```

这里我们在调用Arrays的sort方法时，第二个参数传递的是Comparator接口的子实现类对象。所以Comparator充当的是抽象策略角色，而具体的子实现类充当的是具体策略角色。环境角色类（Arrays）应该持有抽象策略的引用来调用。那么，Arrays类的sort方法到底有没有使用Comparator子实现类中的 **compare()** 方法吗？让我们继续查看TimSort类的 **sort()** 方法，代码如下：

```
class TimSort<T> {
    static <T> void sort(T[] a, int lo, int hi, Comparator<? super T> c,
                         T[] work, int workBase, int workLen) {
        assert c != null && a != null && lo >= 0 && lo <= hi && hi <= a.length;

        int nRemaining  = hi - lo;
        if (nRemaining < 2)
            return;  // Arrays of size 0 and 1 are always sorted

        // If array is small, do a "mini-TimSort" with no merges
        if (nRemaining < MIN_MERGE) {
            int initRunLen = countRunAndMakeAscending(a, lo, hi, c);
            binarySort(a, lo, hi, lo + initRunLen, c);
```

```
            return;
        }
        ...
    }

    private static <T> int countRunAndMakeAscending(T[] a, int lo, int
hi,Comparator<? super T> c) {
        assert lo < hi;
        int runHi = lo + 1;
        if (runHi == hi)
            return 1;

        // Find end of run, and reverse range if descending
        if (c.compare(a[runHi++], a[lo]) < 0) { // Descending
            while (runHi < hi && c.compare(a[runHi], a[runHi - 1]) < 0)
                runHi++;
            reverseRange(a, lo, runHi);
        } else {                              // Ascending
            while (runHi < hi && c.compare(a[runHi], a[runHi - 1]) >= 0)
                runHi++;
        }

        return runHi - lo;
    }
}
```

上面的代码中最终会跑到 **countRunAndMakeAscending()** 这个方法中。我们可以看见，只用了compare方法，所以在调用Arrays.sort方法只传具体compare重写方法的类对象就行，这也是Comparator接口中必须要子类实现的一个方法。

观察者模式

概述

定义：

又被称为发布-订阅（Publish/Subscribe）模式，它定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态变化时，会通知所有的观察者对象，使他们能够自动更新自己。

结构

在观察者模式中有如下角色：

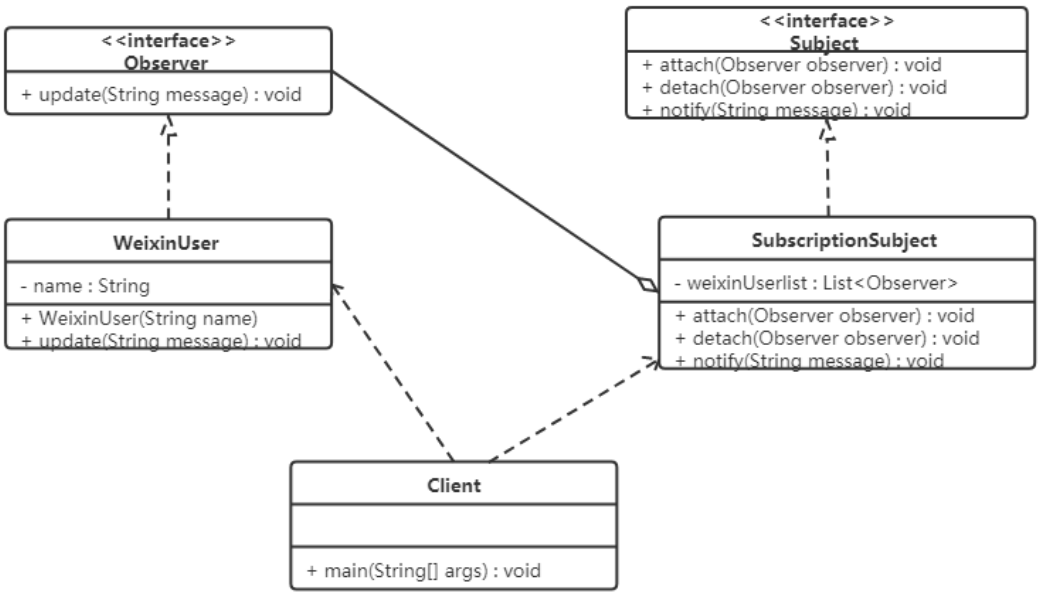
- Subject：抽象主题（抽象被观察者），抽象主题角色把所有观察者对象保存在一个集合里，每个主题都可以有任意数量的观察者，抽象主题提供一个接口，可以增加和删除观察者对象。
- ConcreteSubject：具体主题（具体被观察者），该角色将有关状态存入具体观察者对象，在具体主题的内部状态发生改变时，给所有注册过的观察者发送通知。
- Observer：抽象观察者，是观察者的抽象类，它定义了一个更新接口，使得在得到主题更改通知时更新自己。
- ConcrereObserver：具体观察者，实现抽象观察者定义的更新接口，以便在得到主题更改通知时更新自身的状态。

案例实现

【例】微信公众号

在使用微信公众号时，大家都会有这样的体验，当你关注的公众号中有新内容更新的话，它就会推送给关注公众号的微信用户端。我们使用观察者模式来模拟这样的场景，微信用户就是观察者，微信公众号是被观察者，有多个的微信用户关注了程序猿这个公众号。

类图如下：



代码如下：

定义抽象观察者类，里面定义一个更新的方法

```java
public interface Observer {
    void update(String message);
}
```

定义具体观察者类，微信用户是观察者，里面实现了更新的方法

```java
public class WeixinUser implements Observer {
    // 微信用户名
    private String name;

    public WeixinUser(String name) {
        this.name = name;
    }
    @Override
    public void update(String message) {
        System.out.println(name + "-" + message);
    }
}
```

定义抽象主题类，提供了attach、detach、notify三个方法

```java
public interface Subject {
    //增加订阅者
    public void attach(Observer observer);

    //删除订阅者
    public void detach(Observer observer);

    //通知订阅者更新消息
    public void notify(String message);
}

```

微信公众号是具体主题（具体被观察者），里面存储了订阅该公众号的微信用户，并实现了抽象主题中的方法

```java
public class SubscriptionSubject implements Subject {
    //储存订阅公众号的微信用户
    private List<Observer> weixinUserlist = new ArrayList<Observer>();

    @Override
    public void attach(Observer observer) {
        weixinUserlist.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        weixinUserlist.remove(observer);
    }

```

```
    @Override
    public void notify(String message) {
        for (Observer observer : weixinUserlist) {
            observer.update(message);
        }
    }
}
```

客户端程序

```
public class Client {
    public static void main(String[] args) {
        SubscriptionSubject mSubscriptionSubject=new SubscriptionSubject();
        //创建微信用户
        WeixinUser user1=new WeixinUser("孙悟空");
        WeixinUser user2=new WeixinUser("猪悟能");
        WeixinUser user3=new WeixinUser("沙悟净");
        //订阅公众号
        mSubscriptionSubject.attach(user1);
        mSubscriptionSubject.attach(user2);
        mSubscriptionSubject.attach(user3);
        //公众号更新发出消息给订阅的微信用户
        mSubscriptionSubject.notify("传智黑马的专栏更新了");
    }
}

```

优缺点

1，优点：

- 降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。
- 被观察者发送通知，所有注册的观察者都会收到信息【可以实现广播机制】

2，缺点：

- 如果观察者非常多的话，那么所有的观察者收到被观察者发送的通知会耗时
- 如果被观察者有循环依赖的话，那么被观察者发送通知会使观察者循环调用，会导致系统崩溃

使用场景

- 对象间存在一对多关系，一个对象的状态发生改变会影响其他对象。
- 当一个抽象模型有两个方面，其中一个方面依赖于另一方面时。

JDK中提供的实现

在 Java 中，通过 java.util.Observable 类和 java.util.Observer 接口定义了观察者模式，只要实现它们的子类就可以编写观察者模式实例。

1，Observable类

Observable 类是抽象目标类（被观察者），它有一个 Vector 集合成员变量，用于保存所有要通知的观察者对象，下面来介绍它最重要的 3 个方法。

- void addObserver(Observer o) 方法：用于将新的观察者对象添加到集合中。
- void notifyObservers(Object arg) 方法：调用集合中的所有观察者对象的 update方法，通知它们数据发生改变。通常越晚加入集合的观察者越先得到通知。
- void setChange() 方法：用来设置一个 boolean 类型的内部标志，注明目标对象发生了变化。当它为true时，notifyObservers() 才会通知观察者。

2，Observer 接口

Observer 接口是抽象观察者，它监视目标对象的变化，当目标对象发生变化时，观察者得到通知，并调用 update 方法，进行相应的工作。

【例】警察抓小偷

警察抓小偷也可以使用观察者模式来实现，警察是观察者，小偷是被观察者。代码如下：

小偷是一个被观察者，所以需要继承Observable类

```
1  public class Thief extends Observable {
2
3      private String name;
4
5      public Thief(String name) {
6          this.name = name;
7      }
8
9      public void setName(String name) {
10         this.name = name;
11     }
12
13     public String getName() {
```

```
        return name;
    }

    public void steal() {
        System.out.println("小偷：我偷东西了，有没有人来抓我！！！");
        super.setChanged(); //changed  = true
        super.notifyObservers();
    }
}

```

警察是一个观察者，所以需要让其实现Observer接口

```
public class Policemen implements Observer {

    private String name;

    public Policemen(String name) {
        this.name = name;
    }
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public void update(Observable o, Object arg) {
        System.out.println("警察：" + ((Thief) o).getName() + "，我已经盯你很久了，你可以保持沉默，但你所说的将成为呈堂证供！！！");
    }
}
```

客户端代码

```
public class Client {
    public static void main(String[] args) {
        //创建小偷对象
        Thief t = new Thief("隔壁老王");
        //创建警察对象
        Policemen p = new Policemen("小李");
        //让警察盯着小偷
        t.addObserver(p);
        //小偷偷东西
        t.steal();
    }
}
```