4. 面向对象

4.1. 面向对象基本概念

- 面向过程：以步骤为单位，一步一步完成某一个具体的事情
- 面向对象：以对象为单位，通过调度组合不同的对象来完成某一个事情

4.2. 类与对象

- 类：是一组具有相同特性（属性）与行为（方法）的事物集合
- 类与对象的关系：
    - 类表示一个共性的产物，是一个综合的特征；对象是一个个性的产物，是一个个体的特征
    - 类由属性和方法组成

    - 属性：就相当于一个个的特征
    - 方法：就相当于人的一个个的行为
- 类与对象的定义格式
    - 类的格式：

```
// 类的格式
class 类名称 {
    属性名称;
    返回值类型 方法名称() {
        // 方法内容
    }
}
```

    - 对象的定义：一个类要想真正的进行操作，则必须依靠对象，对象的定义格式如下：

```
// 对象的定义
类名称 对象名称 = new 类名称();
```

    - 注：如果要想访问类中的属性或方法：

```
// 访问类中的属性
对象.属性;
// 访问类中的方法
对象.方法();
```

    - 对象声明有两种含义：

```
// 表示声明了一个对象，但是此对象无法使用，horse没有具体的内存指向
声明对象: Horse horse = null;

// 表示实例化了对象，可以使用
实例化对象: horse = new Horse();

// 通过对象调用方法
horse.eat();

// 匿名对象调用方法
new Horse().ear();
```

- 对象与内存分析
    - new关键字的作用：
        - 创建一个对象
        - 实例化对象
        - 申请内存空间
    - 对象内存分析：

```
// 对象在内存中的结构
// horse存储在栈内存中，horse = null表示还没有申请空间
Horse horse = null;
// horse存储的是堆内存的地址，此时horse在堆内存中申请了空间，存储horse的属性
horse = new Horse();

// 给对象的属性赋值
// horse存储在栈内存中，horse存储堆内存的地址，给堆内存中的name，age属性赋值
horse.name = "小白";
horse.age = 4;

// 在内存中创建的多个对象
// horse1和horse2都存储在栈内存区域中
Horse horse1 = null;
Horse horse2 = null;
// 对象horse1和horse2实例化并申请堆内存空间，其中String类型初始化为null，int类型初始化为age
horse1 = new Horse();
horse2 = new Horse();
// 分别给两个对象的属性赋值
// 给堆内存中的对象属性赋值
horse1.name = "小白";
horse1.age = 4;
horse2.name = "小黑";
horse2.age = 5;

// 声明两个对象，一个实例化，一个没实例化
// 声明两个对象horse1和horse2
Horse horse1 = null;
Horse horse2 = null;
// 将对象horse1实例化，给分配堆内存空间
horse1 = new Horse();
// 对象之间的赋值
// 给对象horse1中的属性赋予初始值
horse1.name = "小白";
horse1.age = 4;
// 将对象horse1所存储的地址赋值给对象horse2，即现在对象horse1和horse2指向同一片堆内存区域
// 相同类型的对象才可以赋值
horse2 = horse1;
// 修改相同的堆内存区域中的name属性
horse2.name = "小黑";

// 对象之间的赋值
Horse horse1 = new Horse();
Horse horse2 = new Horse();
// 分别给horse1和horse2的属性赋值
horse1.name = "小黑";
horse1.age = 4;
horse2.name = "小白";
horse2.age = 5;
// 将horse1中存储的地址赋值给horse2，此时horse2所对应的堆内存区域成为垃圾
// 此时horse2所对应的堆内存区域被JVM的GC程序认为是垃圾
horse2 = horse1;
// 此时horse1和horse2指向同一片堆内存区域，修改的是horse1对象的name属性
horse2.name = "嘿嘿";
```

- 注：一个对象在内存中的大小，由该对象的所有属性所占的内存大小的总和。引用类型变量在32位系统上占4个字节，在64位系统上占8个字节。还要算上其隐性数据所占的大小
- 注：编程时，在确定不使用对象时，要尽早的释放对象，对象=null

4.3. 封装性

```java
// Person类
class Person {
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }

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

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

- 封装性是面向对象思想的三大特征之一
- 封装就是隐藏实现细节，仅对外提供访问接口
- 封装有：属性的封装、方法的封装、类的封装、组件的封装、模块化封装、系统级封装...
- 封装的好处：
    - 模块化
    - 信息隐藏
    - 代码重用
    - 插件化易于调试
    - 具有安全性
- 封装的缺点：
    - 影响执行效率
- 封装性：
    - 如果属性没有封装，那么在本类之外创建对象后，可以直接访问属性

    - private关键字：访问权限修饰符，表示私有的属性或方法，只能在本类中访问。要想在外部访问私有属性，我们需要提供公有的方法来间接访问
    - public关键字：访问权限修饰符，公有的属性或方法，可以被类外部的其他类访问
    - 通常在一类中，属性都私有化，并对外提供setter和getter方法
- 成员变量和局部变量：
    - 在类中的位置不同：
        - 成员变量：在类中定义
        - 局部变量：在方法中定义或者方法的参数
    - 在内存中的位置不同：
        - 成员变量：在堆内存（成员变量属于对象，对象进堆内存）
        - 局部变量：在栈内存（局部变量属于方法，方法进栈内存）
    - 生命周期不同：
        - 成员变量：随着对象的创建而存在，随着对象的销毁而消失
        - 局部变量：随着方法的调用而存在，随着方法的调用完毕而消失
    - 初始化值不同：
        - 成员变量：有默认初始化值，引用类型默认为null
        - 局部变量：没有默认初始化值，必须定义，赋值，然后才能使用
    - 注：局部变量名称和成员变量名称一样，在方法中使用的时候，采用的是就近原则

4.4. 构造方法

```
// Dog类
class Dog {
    private String String;
    private int age;
}
```

- 构造方法的概念：
    - 构造方法就是类构造对象时调用的方法，用于对象的初始化工作
    - 构造方法就是实例化一个类非对象时，也就是new的时候，最先调用的方法
- 构造方法的定义：

```
// 构造方法定义的格式
类名 {
}

// 对象的实例化语法
// new Dog后面有一个括号，带括号表示调用了方法，此时调用的方法就是构造方法
Dog dog = new Dog();
```

  - 构造方法定义的格式：方法名称与类名称相同，无返回值类型的声明

- 无参数的构造方法：

```
// 无参数的构造方法
// 默认的构造方法，通常保留默认的构造方法
public Dog() {
    System.out.println("无参数的构造方法！");
}
```

- 带一个参数的构造方法：

```
// 带一个参数的构造方法
public Dog(String name) {
    this.name = name;
    System.out.println("带一个参数的构造方法！");
}
```

- 带两个参数的构造方法：

```
// 带两个参数的构造方法
public Dog(String name, int age) {
    this.name = name;
    this.age = age;
}
```

- 构造方法之间的调用：

```
// 构造方法之间的调用
// 构造方法调用时要有出口
public Dog(String name, int age) {
    // 在调用其他的构造方法时此语句必须在第一句
    this(name);// 调用带一个参数的构造方法
    this.age = age;
}
```

- 小结：
  - 构造方法名称与类名相同，没有返回值声明（包括void）
  - 构造方法用于初始化数据（属性）
  - 每一个类中都会有一个默认的无参的构造方法

- 如果类中有显式的构造方法，还想保留默认的构造方法，需要显式的写出来
- 构造方法可以有多个，但参数不一样，称为构造方法的重载
- 在构造方法中调用另一个构造方法，使用this(...)，该句代码必须在第一句
- 构造方法之间的调用，必须要有出口
- 给对象初始化数据可以使用构造方法或setter()方法，通常情况下，两者都会保留
- 一个好的编程习惯是要保留默认的构造方法（为了方便一些框架代码使用反射来创建对象）
- private Dog(){}，构造方法私有化，当我们的需求是为了保证该类只有一个对象时
  - 注：工具类（没有属性的类，只有行为）并且该工具对象被频繁使用。权衡只用一个对象与产生多个对象的内存使用，来确定该类是否要定义为只需要一个对象

4.5. this关键字

- this关键字可以完成以下的操作：
  - 调用类中的属性
  - 调用类中的方法或构造方法
  - 表示当前对象
- 表示当前对象：在方法被调用的过程中，那个对象调用了方法，在方法内的this就表示谁
- 在方法中使用this调用类中的其他方法或属性，this可以省略，this前面可以使用当前的类名.this

4.6. 值传递与引用传递

- 值传递

```
// 值传递
public class ValuePassing {
    public static void main(String[] args) {
        int x = 10;
        method(x);
        System.out.println("x = " + x);
    }

    public static void method(int m) {
        m = 20;
    }
```

```
}

// 输出
x = 10
```

- 引用传递

```
 // 引用传递
public class ReferencePassing {
    public static void main(String[] args) {
        Duck d = new Duck();
        method(d);
        System.out.println("Dock age = " + d.age);
    }

    public static void method(Duck duck) {
        duck.age = 5;
    }
}

class Duck {
    int age = 2;// 省略封装
}

// 输出
Dock age = 5
```

- String传递

```
 //  String传递
// 字符串本身就是一个对象
public class ReferencePassing {
    public static void main(String[] args) {
        String name = "小飞";
        method(name);
        System.out.println("name = " + name);
    }

    public static void method(String sname) {
        sname = "小贝";
    }
}

// 输出
name = 小飞
```

- String传递

```
 //  String传递
public class ReferencePassing {
    public static void main(String[] args) {
        Person p = new Person();
        method(p);
        System.out.println("Person person = " + p.name);
```

```
    }

    public static void method(Person person) {
        person.name = "贝贝";
    }
}

class Person {
    String name = "飞飞";// 省略封装
}

// 输出
Perso perso = 贝贝
```

4.7. 对象的一对一关系

```
// 英雄类
class Hero {
    private String name;
    private int age;
    // 一对一的关系
    private Weapon weapon;

    public Hero() {
    }

    public Hero(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;
    }

    public Weapon getWeapon() {
        return weapon;
    }

    public void setWeapon(Weapon weapon) {
        this.weapon = weapon;
    }

}

```

```java
// 兵器类
class Weapon {
    private String name;
    private int grade;
    private Hero hero;

    public Weapon() {
    }

    public Weapon(String name, int grade) {
        this.name = name;
        this.grade = grade;
    }

    public String getName() {
        return name;
    }

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

    public int getGrade() {
        return grade;
    }

    public void setGrade(int grade) {
        this.grade = grade;
    }

    public Hero getHero() {
        return hero;
    }

    public void setHero(Hero hero) {
        this.hero = hero;
    }
}
```

- 在主函数中进行对象关系的关联：

```java
// 把两个对象关联
hero.setWeapon(weapon);
weapon.setHero(hero);
```

4.8. static关键字

- static关键字的作用：
    - 使用static关键字修饰一个属性：声明为static的变量实质上就是全局变量
    - 使用static关键字修饰一个方法：通常，在一个类中定义一个方法为static，那就是说，无需本类的对象即可调用此方法
    - 使用static关键字修饰一个类（内部类）

- static关键字：
  - 静态变量或方法不属于对象，依赖类
  - 静态变量是全局变量生命周期从类被加载后一直到程序结束
  - 静态变量只会存一份，在静态方法区中存储
  - 静态变量是本来所有对象共享一份
  - 建议不要使用对象名去调用静态数据，直接使用类名调用
  - static修饰一个方法，那么该方法属于类，不属于对象，建议直接用类名调用
  - 静态方法不能访问非静态属性和方法，只能访问静态
  - 不能以任何方式引用this或super
- 所有对象共有的属性或方法时，我们可以定义为静态的

4.9. main方法分析

```
// 主方法
public static void main(String[] args) {
    // 代码块
}

public: 公有的，最大的访问权限
static: 静态的，无需创建对象
void: 表示没有返回值，无需向JVM返回结果
main: 方法名，固定的方法名
String[] args: 表示参数为字符串数组，可以在调用方法时传入参数
```

4.10. 代码块

- 普通代码块：在方法中写的代码块

```
public class OrdinaryCodeBlock {
    public static void main(String[] args) {
        {
            // 普通代码块
            String infoString = "局部变量1";
            System.out.println(infoString);
        }
        String infoString = "局部变量2";
        System.out.println(infoString);
    }
}
```

- 构造代码块：在类中定义的代码块

```
class Demo {
    {
        // 构造块
        System.out.println("构造块");
    }

    public Demo() {
        System.out.println("构造方法.");
    }
}
```

  - 在创建对象时被调用，优于构造方法执行
- 静态代码块：在类中使用static声明的代码块称为静态代码块

```
class Demo {
    {
        System.out.println("构造代码块");
    }
    static {
        System.out.println("静态代码块");
    }

    public Demo() {
        System.out.println("构造方法");
    }
}
```

  - 在第一次使用的时候被调用（创建对象），只会执行一次，静态代码块优于构造代码块执行。我们在项目开发中，通常会使用静态代码块来初始化只调用一次的数据
- 同步代码块：（多线程中讲解）

4.11. 单例设计模式

- 单例设计模式：保证一个类只有一个实例，并提供一个访问它的全局访问点
- 单例实现步骤：
  - 构造方法私有化
  - 声明一个本类对象
  - 给外部提供一个静态方法获取对象实例
- 两种实现方式：
  - 饿汉式：在类被加载后，对象被创建，到程序结束后释放（占用内存的时间长，提高效率）

```
public class SingleCaseDesignPattern {
    public static void main(String[] args) {
        SingleCase singleCase = SingleCase.getInstance();
```

```
        singleCase.print();
    }
}

// 饿汉式
class SingleCase {
    private SingleCase() {}

    private static SingleCase singleCase = new SingleCase();

    public static SingleCase getInstance() {
        return singleCase;
    }

    public void print() {
        System.out.println("饿汉式单例设计模式");
    }
}
```

- 懒汉式：在第一次调用getInstance方法时，对象被创建，到程序结束后释放（占用内存的时间短，效率稍低），也叫作懒加载或延迟加载

```
public class SingleCaseDesignPattern {
    public static void main(String[] args) {
        SingleCase singleCase = SingleCase.getInstance();
        singleCase.print();
    }
}

// 懒汉式
class SingleCase {
    private SingleCase() {}

    private static SingleCase singleCase;

    public static SingleCase getInstance() {
        if (singleCase == null) {
            singleCase = new SingleCase();
        }
        return singleCase;
    }

    public void print() {
        System.out.println("懒汉式单例设计模式");
    }
}
```

- 使用单例设计模式：
  - 在设计一些工具类的时候（通常工具类，只有功能方法没有属性）
  - 工具类可能会被频繁调用

- 单例设计模式的目的：为了节省重复创建对象所带来的内存消耗，从而带来效率
- 使用构造方法私有化+静态方法：

```
class Tools {
    private Tools() {}

    public static void print1() {
    }

    public static void print2() {
    }
}
```

4.12. 对象数组与管理

- 对象数组：数组里的每个元素都是类的对象，赋值时先定义对象，然后将对象直接赋给数组

```
// 使用对象数组实现多个Chicken的管理
Chicken[] chicken = new Chicken[10];
```

- 举例：

```
import java.util.Arrays;

/**
 * @author xiao儿
 * 动态数组:
 * 1.数组是一种线性数组结构
 * 2.数组不适合进行删除插入等操作，适合添加、查找、遍历操作
 */
public class ArraysOfObjects {
    public static void main(String[] args) {
        ChickenManager chickenManager = new ChickenManager(5);
        // 添加
        chickenManager.add(new Chicken(1, "小小", 10));
        chickenManager.add(new Chicken(2, "小二", 8));
        chickenManager.add(new Chicken(3, "小三", 6));
        chickenManager.add(new Chicken(4, "小四", 4));
        chickenManager.add(new Chicken(5, "小五", 2));

        chickenManager.add(new Chicken(6, "小豆", 1));
        System.out.println("数组的长度是: " + chickenManager.length());

        System.out.println("--------findAll---------");
        chickenManager.printAll();

        System.out.println("--------find-----------");
        Chicken chicken = chickenManager.find(5);
        chicken.print();

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

```
        chickenManager.update(new Chicken(1, "下蛋公鸡", 20));
        chickenManager.printAll();

        System.out.println("--------delete-----------");
        chickenManager.delete(6);
        chickenManager.printAll();
    }
}

// 小鸡管理类
class ChickenManager {
    private Chicken[] chickens = null;
    private int count = 0;// 当前数组的元素个数（下标）

    public ChickenManager(int size) {
        if (size > 0) {
            chickens = new Chicken[size];
        } else {
            chickens = new Chicken[5];
        }
    }

    public int length() {
        return chickens.length;
    }

    // 添加：实现动态数组
    public void add(Chicken chicken) {
        if (count >= chickens.length) {// 数组已满，需要扩充
            // 算法1：扩充原来数组大小的一半 chickens.length * 3 / 2 + 1
            // 算法2：扩充原来数组的一倍 chickens.length * 2
            int newLength = chickens.length * 2;
            chickens = Arrays.copyOf(chickens, newLength);
        }
        chickens[count] = chicken;
        count++;
    }

    // 删除
    public void delete(int id) {
        for (int i = 0; i < count; i++) {
            if (chickens[i].getId() == id) {
                // 找到了要删除的对象，把该对象之后的对象向前移动一起
                for (int j = i; j < count - 1; j++) {
                    chickens[j] = chickens[j + 1];
                }
                // 把最后一个对象赋值为空（删除）
                chickens[count - 1] = null;
                count--;// 下标减一
                break;
            }
        }
    }

    // 更新
    public void update(Chicken chicken) {
        Chicken tempChicken = find(chicken.getId());
        if (tempChicken != null) {
```

```
            tempChicken.setNameString(chicken.getNameString());
            tempChicken.setAge(chicken.getAge());
        }
    }

    // 查找
    public Chicken find(int id) {
        for (int i = 0; i < count; i++) {
            if (chickens[i].getId() == id) {
                return chickens[i];
            }
        }
        return null;
    }

    // 输出所有
    public void printAll() {
        for (int i = 0; i < count; i++) {
            chickens[i].print();
        }
    }
}

// 小鸡类（数据对象）value object (VO)
class Chicken {
    private int id;
    private String nameString;
    private int age;

    public Chicken() {
    }// 一般情况下最好保留默认的构造方法

    public Chicken(int id, String nameString, int age) {
        this.id = id;
        this.nameString = nameString;
        this.age = age;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNameString() {
        return nameString;
    }

    public void setNameString(String nameString) {
        this.nameString = nameString;
    }

    public int getAge() {
        return age;
    }

```

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

    public void print() {
        System.out.println("id = " + id + ", name = " + nameString + ", age = " + age);
    }
}
```

4.13. 继承的基本概念

- 基本概念：继承是从已有的类创建新类的过程
  - 继承是面向对象三大特征之一
  - 被继承的类称为父类（超类），继承父类的类称为子类（派生类）
  - 继承是指一个对象直接使用另一个对象的属性和方法
  - 通过继承可以实现代码重用
- 语法格式：

```
// 语法格式
{访问权限} class 子类名 extends 父类名 {
    类体定义
}

// 示例
public class Dog {
    private String name;
    private String sex;
    public void eat() {
        System.out.println("吃饭");
    }
}

public class HomeDog extends Dog {
    // 类的定义
}

public class HuskyDog extends Dog {
    // 类的定义
}
```

- protected：（受保护的访问权限修饰符，用于修饰属性和方法，使用protected修饰的属性和方法可以被子类继承
- 继承的限制：
  - Java只能实现单继承，也就是一个类只能有一个父类
  - 允许多层继承，即一个子类可以有一个父类，一个父类还可以有其他的父类
  - 继承只能继承非私有的属性和方法

    - 构造方法不能被继承
- 注：子类在创建对象时，都会调用父类默认的无参的构造方法
- 继承小结：
    - 继承是发生在多个类之间
    - 继承使用关键字extends
    - Java只能单继承，允许多层继承
    - 被继承的类叫父类（超类），继承父类的类叫子类（派生类）
    - 在父类中的非私有属性和方法可以被子类继承
    - protected（受保护的访问权限是修饰符），修饰的属性或方法可以被子类继承
    - 构造方法不能被继承
    - 创建对象会调用构造方法，调用构造方法不一定就是创建该类对象
    - 实例化子类对象，会先调用父类的构造方法，如果父类中没有默认的构造方法，那么子类必须显式的通过super(...)来调用父类的带参构造方法，super也只能在子类构造方法中的第一句
- 继承的好处：
    - 提高代码的复用性
    - 提高代码的维护性
    - 让类与类之间产生关系，是多态的前提
- 继承的缺点：
    - 增强了类与类之间的耦合性
- 开发原则：高内聚，低耦合

4.14. 子类的实例化过程

- 在子类进行实例化的时候，首先会让其父类进行初始化操作，之后子类再自己进行实例化操作
- 实例化过程：
    - 子类实例化时会先调用父类的构造方法
    - 如果父类没有默认的构造方法，在子类的构造方法中必须显式的调用父类的构造方法
- 结论：
    - 构造方法只是用于初始化类中的字段以及执行一些初始化代码
    - 调用构造方法并不代表会生成对象

4.15. 方法的重写

- 方法重写(overriding method)：

- 在Java中，子类可以继承父类中的方法，而不需要重写编写相同的方法。但有时子类并不想原封不动的继承父类的方法，而是想做一定的修改，这就需要采用方法的重写。方法重写又称为方法覆盖。在子类和父类中，重写方法后，在调用时，以创建的对象类型为准，谁的对象调用谁的方法

- 关于方法重写的一些特性：
  - 发生在子父类中，重写的两个方法的返回值、方法名、参数列表必须完全一致（子类重写父类的方法）
  - 子类抛出的异常不能超过父类相应方法抛出的异常（子类异常不能大于父类异常）
  - 子类方法的访问级别不能低于父类相应方法的访问级别（子类访问级别不能低于父类访问级别）
  - 父类中的方法若使用private、static、final任意修饰符修饰，那么不能被子类重写
- 重写方法的目的：
  - 若子类从父类中继承过来的方法，不能满足子类特有的需求时，子类就需要重写父类中相应的方法，方法的重写也是程序扩展的体现
- 面试题：overloading与overriding的区别？
  - overloading：方法的重载，发生在同一个类中，方法名相同，参数列表不同，返回值无关
  - overriding：方法的重写，发生在子父类中，方法名相同，参数列表相同，返回值相同，子类的访问修饰符要大于或等于父类的访问修饰符，子类的一场声明必须要小于或等于父类的异常声明。如果方法被private、static、final修饰，那么不能被重写

4.16. super关键字

- super可以完成的操作：
  - 使用super调用父类中的属性，可以从父类实例处获得信息
  - 使用super调用父类中的方法，可以委托父类对象帮助完成某件事情
  - 使用super调用父类中的构造方法（super(实参)形式），必须在子类构造方法的第一条语句，调用父类中相应的构造方法，若不显式的写出来，默认调用父类的无参数的构造方法
- super和this关键字：
  - this：表示当前对象
  - super：表示父类，可以用来调用父类的属性、方法和构造方法

4.17. 继承的应用示例

```java
import java.util.Arrays;

public class InheritanceExample {
    public static void main(String[] args) {
        ImportCosmeticManager cs = new ImportCosmeticManager();
        cs.add(new Cosmetic("香内儿", "进口", 1400));
        cs.add(new Cosmetic("圣罗兰", "进口", 800));
        cs.add(new Cosmetic("大宝", "国产", 20));
        cs.add(new Cosmetic("万紫千红", "国产", 15));
        cs.print();
    }
}

// 可输出进口化妆品的管理类
class ImportCosmeticManager extends CosmeticManager {
    @Override
    public void print() {
        for (int i = 0; i < count; i++) {
            // 比较连个字符串的值是否相等，不能使用==，要是用equals()
            if ("进口".equals(cosmetics[i].getType())) {
                System.out.println(cosmetics[i].getInfo());
            }
        }
    }
}

// 可按照单价进行排序的化妆品管理类
class SortCosmeticManager extends CosmeticManager {
    // 排序输出所有产品
    @Override
    public void print() {
        Cosmetic[] temp = Arrays.copyOf(cosmetics, count);
        Cosmetic c = null;
        // System.out.println(temp.length);
        for (int i = 0; i < temp.length - 1; i++) {
            for (int j = 0; j < temp.length - i - 1; j++) {
                if (temp[j].getPrice() > temp[j + 1].getPrice()) {
                    c = temp[j];
                    temp[j] = temp[j + 1];
                    temp[j + 1] = c;
                }
            }
        }
        for (Cosmetic cosmetic : temp) {
            System.out.println(cosmetic.getInfo());
        }
    }
}

// 化妆品管理类
class CosmeticManager {
    protected Cosmetic[] cosmetics = new Cosmetic[4];
    protected int count = 0;

    // 进货功能
    public void add(Cosmetic c) {
        int len = cosmetics.length;
```

```
        if (count >= len) {
            int newLen = len * 2;
            cosmetics = Arrays.copyOf(cosmetics, newLen);
        }
        cosmetics[count] = c;
        count++;
    }

    // 输出所有产品
    public void print() {
        for (int i = 0; i < count; i++) {
            System.out.println(cosmetics[i].getInfo());
        }
    }
}

// 化妆品类
class Cosmetic {
    private String name;// 品牌
    private String type;// 进口或国产
    private double price;// 单价

    public Cosmetic() {
    }

    public Cosmetic(String name, String type, double price) {
        this.name = name;
        this.type = type;
        this.price = price;
    }

    public String getName() {
        return name;
    }

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

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public double getPrice() {
        return price;
    }

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

    public String getInfo() {
        return "name=" + name + ",type=" + type + ",price=" + price;
    }
```

```
116 }
```

4.18. final关键字

- final关键字可以完成的操作：
  - 使用final关键字声明一个常量
    - 修饰属性或者修饰局部变量（最终变量），也称为常量
    - 常量的命名建议使用全大写，必须在定义时或调用构造方法时完成初始化
  - 使用final关键字声明一个方法
    - 该方法为最终方法，且只能被子类继承，但不能被子类重写
  - 使用final关键字声明一个类
    - 该类就转变为最终类，没有子类的类，final修饰的类无法被继承
  - 在方法参数中使用final，在该方法内部不能修改参数的值（在内部类中详解）

4.19. 抽象类

```java
abstract class Peoople extends Animal {
    // 抽象方法
    public abstract void eat();
    public void sleep() {
        System.out.println("我要睡觉");
    }
}
```

- 基本概念：
  - 很多具有相同特征和行为的对象可以抽象一个类，很多具有相同特征和行为的类可以抽象为一个抽象类
  - 使用abstract关键字声明为抽象类
- 继承抽象类必须实现抽象类的所有抽象方法
- 抽象类的规则：
  - 抽象类可以没有抽象方法，有抽象方法的类必须是抽象类
  - 非抽象类继承抽象类必须实现所有的抽象方法
  - 抽象类可以继承抽象类，可以不实现父类抽象方法
  - 抽象类可以有方法实现和属性
  - 抽象类不能被实例化
  - 抽象类不能声明final
  - 抽象类可以有构造方法

4.20. 接口

```java
// 接口的定义格式
interface 接口名称 {
    全局变量;
    抽象方法;
}

// 示例
interface IEat {
    // public abstract void eat();
    void eat();
    // public static final int NUM = 10;
    int NUM = 10;
}

interface ISleep extends IEat {
    void sleep();
}
```

- 接口的概念：
  - 接口是一组行为的规范、定义，没有实现（JDK1.8后新增默认方法）

```java
// 默认的实现
// JDK1.8之后新特性，可以被所有实现类继承
public default void print() {
    System.out.println("默认的方法");
}
```

  - 使用接口，可以让我们的程序更加利于变化
  - 接口是面向对象编程体系中的思想精髓之一
  - 面向对象设计法则：基于接口编程
- 接口的规则：
  - 定义一个接口，使用interface关键字
  - 在一个接口中，只能定义常量、抽象方法，JDK1.8后可以定义默认的实现方法
  - 接口可以继承多个接口：extends xxx, xxx
  - 一个具体类实现接口使用implements关键字
  - 一个类可以实现多个接口
  - 抽象类实现接口可以不实现接口的方法
  - 在接口中定义的方法没有声明访问修饰符，默认为public
  - 接口中不能有构造方法
  - 接口不能被实例化
- 面向对象设计原则：
  - 对修改关闭，对扩展开放

- 面向接口编程

4.21. 多态性

- 多态的概念：对象在运行过程中的多种形态
- 多态性分类：
  - 方法的重载与重写
  - 对象的多态性

```
// 用父类的引用指向子类对象（用大的类型去接收小的类型，向上转型，自动转型）
Chicken home = new HomeChicken();
```

- 结论：
  - 在编程时针对抽象类型的编写代码，称为面向抽象编程（面向接口编程）
  - 父类通常都定义为抽象类、接口
- 对象的多态性：对象多态性是从继承关系中的多个类而来
- 向上转型：将子类实例转为父类实例
  - 格式：

```
父类 父类对象 = 子类实例; --> 自动转换
// 以基本数据类型操作为例：int i = 'a';
（因为char的容量比int小，所以可以自动完成）
```

- 向下转型：将父类实例转为子类实例
  - 格式：

```
子类 子类对象 = （子类）父类实例; --> 强制转换
// 以基本数据类型操作为例：char c = (char)97;
（因为整型是4个字节比char2个字节要大，所以需要强制完成）
```

- 小结：
  - 方法的重载与重写就是方法的多态性表现
  - 多个子类就是父类中的多种形态
  - 父类引用可以指向子类对象，自动转换
  - 子类对象指向父类引用需要强制转换（注意：类型不对会报异常）
  - 在实际开发中尽量使用父类引用（更利于扩展）

4.22. instanceof关键字

- instanceof是用于检查对象是否为指定的类型，通常把父类引用强制转换为子类引用时要使用，以避免发生类型转换异常（ClassCastException）

- 语法格式：

```
对象 instanceof 类型    -- 返回boolean类型值

// 示例
if (homeChicken instanceof Chicken) {
    //...
}
```

- 该语句一般用于判断一个对象是否为某个类的实例，是返回trur，否则返回false
- 父类的设计法则：
  - 父类通常情况下都设计为抽象类或接口，其中优先考虑接口，如接口不能满足才考虑抽象类
  - 一个具体的类尽可能不去继承另一个具体类，这样的好处是无需要检查对象是否为父类的对象

4.23. 抽象类应用–模板方法模式

- 模板方法模式：定义一个操作中的算法的骨架，而将一些可变部分的实现延迟到了子类中。模板方法模式使得子类可以不改变一个算法的结构即可重新定义该算法的某些特定步骤

```
abstract class BaseManager {
    public void action(String name, String method) {
        if ("admin".equals(name)) {
            execute(method);
        } else {
            System.out.println("你没有操作权限，请联系管理员");
        }
    }

    public abstract void execute(String method);
}

class UserManager extends BaseManager {
    @Override
    public void execute(String method) {
        // 用户是否登录的验证
        // 验证成功后才可以执行以下操作
        if ("add".equals(method)) {
            System.out.println("执行了添加操作");
        } else if ("del".equals(method)) {
            System.out.println("执行了删除操作");
        }
    }
}
```

4.24. 接口应用–策略模式

- 策略模式：定义了一系列的算法，将每一种算法封装起来并可以相互替换使用，策略模式让算法独立于使用它的客户应用而独立变化
- OO设计原则：
  - 面向接口编程（面向抽象编程）
  - 封装变化
  - 多用组合，少用继承

```java
public class StrategyMode {
    public static void main(String[] args) {
        BaseService user = new UserService();
        // 可在此处进行相互替换
        user.setISave(new FileSave());
        user.add("user");
    }
}

// 把可变的行为抽象出来，定义一系列的算法
interface ISave {
    public void save(String data);
}

class FileSave implements ISave {
    @Override
    public void save(String data) {
        System.out.println("把数据保存在文件中。" + data);
    }
}

class NetSave implements ISave {
    @Override
    public void save(String data) {
        System.out.println("把数据保存在网络上。" + data);
    }
}

abstract class BaseService {
    private ISave iSave;

    public void setISave(ISave iSave) {
        this.iSave = iSave;
    }

    public void add(String data) {
        System.out.println("检查数据合法性");
        iSave.save(data);
        System.out.println("数据保存完毕");
    }
}

class UserService extends BaseService {

}
```

4.25. Object类

- Object类是类层次结构的根类
  - 每个类都使用Object作为超类。所有对象（包括数组）都实现这个类的方法
  - 所有类都是Object类的子类
- 常用的方法：
  - public String toString()方法：返回该对象的字符串表示
    - toString方法会返回一个“以文本方式表示”此对象的字符串，结果也是一个简明且易于读懂的表达式。建议所有子类都重写此方法

```
// 重写Object类中的toString方法
@Override
public String toString() {
    return "did = " + did + ", name = " + name + ", age = " +
age;
}
```

  - public boolean equals(Object obj)方法：
    - 指明其他某个对象是否与此对象“相等”，equals方法在非空对象应用上实现相等关系，具有自反性、对称性、传递性、一致性

```
// 重写toString类中的equals方法
@Override
public boolean equals(Object obj) {
    if(this == obj) {
        return true;
    }
    if (obj instanceof Doctor) {
        Doctor doc = (Doctor) obj;
        if (!this.name.equals(doc.name)) {
            return false;
        } else if (this.did != doc.did) {
            return false;
        } else if (this.age != doc.age) {
            return false;
        }
        return true;
    }
    return false;
}
```

  - protected void finalize() throws Throwable方法：

        - 当垃圾回收器确定不存在该对象的更多引用时，由对象的垃圾回收器调用此方法。子类重写finalize方法，以配置系统资源或执行其他清除
    - public final Class<?> getClass()方法：
        - 返回此Object的运行类

4.26. 简单工厂模式

- 简单工厂模式：由一个工厂对象决定创建出哪一种产品类的实例。简单工厂模式是工厂模式家族中最简单使用的模式

```
public class FactoryMode {
    public static void main(String[] args) {
        // 使用者和被使用者两者之间，耦合，产生了依赖，当被使用者改变时，会影响使用者
        // 使用工厂模式，降低两者之间的依赖
        // Product phone = new Phone();
        Product phone = ProductFactory.getProduct("phone");
        phone.work();
    }
}

// 工厂类
class ProductFactory {
    public static Product getProduct(String name) {
        if ("phone".equals(name)) {
            return new Phone();
        } else if ("computer".equals(name)) {
            return new Computer();
        } else {
            return null;
        }
    }
}

interface Product {
    public void work();
}

class Phone implements Product {
    @Override
    public void work() {
        System.out.println("手机开始工作......");
    }
}

class Computer implements Product {
    @Override
    public void work() {
        System.out.println("电脑开始工作......");
    }
}
```

4.27. 静态代理模式

- 代理模式（proxy）：为其他对象提供一种代理以控制对这个对象的访问
- 代理模式说白了就是“真实对象”的代表，在访问对象时引入一定程度的间接性，因为这种间接性可以附加多种用途

```java
public class ProxyMode {
    public static void main(String[] args) {
        Action userAction = new UserAction();
        ActionProxy proxy = new ActionProxy(userAction);
        proxy.doAction();
    }
}

interface Action {
    public void doAction();
}

class ActionProxy implements Action {

    private Action target;// 被代理的对象

    public ActionProxy(Action target) {
        this.target = target;
    }

    // 执行操作
    @Override
    public void doAction() {
        long startTime = System.currentTimeMillis();
        target.doAction();
        long endTime = System.currentTimeMillis();
        System.out.println("共耗时: " + (endTime - startTime));
    }
}

class UserAction implements Action {
    @Override
    public void doAction() {
        for (int i = 0; i < 100; i++) {
            System.out.println("用户开始工作");
        }
    }
}
```

4.28. 适配器模式

- 适配器模式：将一个类的接口转换成客户希望的另一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作
- OO设计原则：
    - 面向接口编程（面向抽象编程）

- 封装变化
- 多用组合，少用继承
- 对修饰关闭，对扩展开放

```java
public class AdapterMode {
    public static void main(String[] args) {
        PowerA powerA = new PowerAImpl();
        work(powerA);

        PowerB powerB = new PowerBImpl();
        // work(powerB);
        Adapter adapter = new Adapter(powerB);
        work(adapter);
    }

    public static void work(PowerA a) {
        System.out.println("正在连接...");
        a.insert();
        System.out.println("工作结束");
    }
}

class Adapter implements PowerA {
    private PowerB PowerB;

    public Adapter(PowerB powerB) {
        this.PowerB = powerB;
    }

    @Override
    public void insert() {
        PowerB.connect();
    }
}

interface PowerA {
    public void insert();
}

interface PowerB {
    public void connect();
}

class PowerAImpl implements PowerA {
    @Override
    public void insert() {
        System.out.println("电源A开始工作");
    }
}

class PowerBImpl implements PowerB {
    @Override
    public void connect() {
        System.out.println("电源B开始工作");
    }
}
```

4.29. 内部类

- 内部类就是在一个类的内部定义的类
- 成员内部类：

```
// 成员内部类的格式如下:
class Outer {
    class Inner{}
}

// 编译上述代码会产生两个文件:
Outer.class和Outer$Inner.class
```

  - 在外部创建内部类对象：
    - 内部类除了可以在外部类中产生实例化对象，也可以在外部类的外部来实例化。那么，根据内部类生成的*.class文件：Outer$Inner.class
    - “$”符号在程序运行时将替换成“.”
    - 内部类的访问：通过“外部内.内部类”的形式表示

```
Outer outer = new Outer();// 产生外部类实例
Outer.Inner inner = outer.new Inner();// 实例化内部类对象
```

- 方法内部类：

```
// 方法内部类的格式如下:
class Outer {
    public void doSomething() {
        class Inner {
            public void seeOuter(){}
        }
    }
}
```

  - 内部类可以作为一个类的成员外，还可以把类放在方法内定义
  - 注：
    - 方法内部类只能在定义该内部类的方法内实例化，不可以在此方法外对其实例化
    - 方法内部类对象不能使用该内部类所在方法的非final局部变量
- 静态内部类：

```java
// 静态内部类的格式
class Outer {
    static class Inner{}
}

// 实例化
Outer.Inner inner = new Outer.Inner();
```

    - 在一个类内部定义一个静态内部类：静态的含义是该内部类可以像其他静态成员一样，没有外部类对象时，也能够访问它。静态嵌套类仅能访问外部类的静态成员和方法

- 匿名内部类：
    - 匿名内部类就是没有名字的内部类
    - 匿名内部类的三种情况：
        - 继承式的匿名内部类
        - 接口式的匿名内部类
        - 参数式的匿名内部类
    - 在使用匿名内部类时，要记住以下几个原则：
        - 不能有构造方法，只能有一个实例
        - 不能定义任何静态成员、静态方法
        - 不能是public、protected、private、static
        - 一定是在new的后面，用其隐含实现一个接口或继承一个类
        - 匿名内部类为局部的，所以局部内部类的所有显示都对其生效
- 问题：局部内部类访问局部变量必须使用final修饰，为什么？
    - 当调用这个方法时，局部变量如果没有用final修饰，它的生命周期和方法的生命周期是一样的，当方法被调用时入栈，方法结束后即弹栈，这个局部变量已经被销毁，如果局部内部类的对象还没有马上消失想使用这个局部变量，显然已经无法使用了，如果用final修饰会在类加载的时候进入常量池，即使方法弹栈，常量池的常量还在，也可以继续使用。
    - 注：JDK1.8中取消了在局部内部类中使用的变量必须显式的使用final修饰

4.30. 数据结构之链表

- 递归算法：一种直接或间接调用自身算法的过程
    - 注意：
        - 递归必须要有出口
        - 递归对内存消耗大，容易发生内存溢出

    - 层次调用越多，越危险
- 链表：一种常见的基础数据结构，是一种线性表，但是不会按现行的顺序存储数据，而是在每一个节点里存的是下一个节点的指针

```java
package day04_面向对象;

public class LinkedList {
    public static void main(String[] args) {
        NodeManager nodeManager = new NodeManager();
        nodeManager.add(5);
        nodeManager.add(4);
        nodeManager.add(3);
        nodeManager.add(2);
        nodeManager.add(1);
        nodeManager.print();

        nodeManager.del(3);
        nodeManager.print();

        boolean findFlag = nodeManager.find(1);
        if (findFlag == true) {
            System.out.println("该数据在链表中");
        } else {
            System.out.println("该数据不在链表中");
        }

        boolean updateFlag = nodeManager.update(1, 10);
        if (updateFlag == true) {
            System.out.println("更新成功");
        } else {
            System.out.println("更新失败");
        }
        nodeManager.print();

        nodeManager.insert(2, 6);
        nodeManager.print();
    }
}

class NodeManager {
    private Node rootNode;// 根节点
    private int currentIndex = 0;// 节点的序号，每次操作从0开始

    // 添加节点
    public void add(int data) {
        if (rootNode == null) {
            rootNode = new Node(data);
        } else {
            rootNode.addNode(data);
        }
    }

    // 删除节点
    public void del(int data) {
        if (rootNode == null) {
            return;
```

```
        }
        if (rootNode.getData() == data) {
            rootNode = rootNode.nextNode;
        } else {
            rootNode.delNode(data);
        }
    }

    // 打印所有节点
    public void print() {
        if (rootNode != null) {
            System.out.print(rootNode.getData() + "->");
            rootNode.printNode();
            System.out.println();
        }
    }

    // 查找节点是否存在
    public boolean find(int data) {
        if (rootNode == null) {
            return false;
        }
        if (rootNode.data == data) {
            return true;
        } else {
            return rootNode.findNode(data);
        }
    }

    // 更新节点
    public boolean update(int oldData, int newData) {
        if (rootNode == null) {
            return false;
        }
        if (rootNode.getData() == oldData) {
            rootNode.setData(newData);
            return true;
        } else {
            return rootNode.updateNode(oldData, newData);
        }
    }

    public void insert(int index, int data) {
        if (index < 0) {
            return;
        }
        currentIndex = 0;
        if (index == currentIndex) {
            Node newNode = new Node(data);
            rootNode.nextNode = newNode;
            rootNode = newNode;
        } else {
            rootNode.insertNode(index, data);
        }
    }

    private class Node {
        private int data;
```

```
111         private Node nextNode;// 把当前类型作为属性
112 
113         public Node(int data) {
114             this.data = data;
115         }
116 
117         public int getData() {
118             return data;
119         }
120 
121         public void setData(int data) {
122             this.data = data;
123         }
124 
125         // 添加节点
126         public void addNode(int data) {
127             if (this.nextNode == null) {
128                 this.nextNode = new Node(data);
129             } else {
130                 this.nextNode.addNode(data);
131             }
132         }
133 
134         // 删除节点
135         public void delNode(int data) {
136             if (this.nextNode != null) {
137                 if (this.nextNode.data == data) {
138                     this.nextNode = this.nextNode.nextNode;
139                 } else {
140                     this.nextNode.delNode(data);
141                 }
142             }
143         }
144 
145         // 输出所有节点
146         public void printNode() {
147             if (this.nextNode != null) {
148                 System.out.print(this.nextNode.data + "->");
149                 this.nextNode.printNode();
150             }
151         }
152 
153         // 查找节点是否存在
154         public boolean findNode(int data) {
155             if (this.nextNode != null) {
156                 if (this.nextNode.data == data) {
157                     return true;
158                 } else {
159                     return this.nextNode.findNode(data);
160                 }
161             }
162             return false;
163         }
164 
165         // 修改节点
166         public boolean updateNode(int oldData, int newData) {
167             if (this.nextNode != null) {
168                 if (this.nextNode.data == oldData) {
```

```
                    this.nextNode.data = newData;
                    return true;
                } else {
                    return this.nextNode.updateNode(oldData, newData);
                }
            }
            return false;
        }

        // 插入节点
        public void insertNode(int index, int data) {
            currentIndex++;
            if (index == currentIndex) {
                Node newNode = new Node(data);
                newNode.nextNode = this.nextNode;
                this.nextNode = newNode;
            } else {
                this.insertNode(index, data);
            }
        }
    }
}
```

- 链表和数组：线性数据结构
- 数组适合查找、遍历，是固定长度的
- 链表适合插入、删除，不宜过长，否则会导致遍历性能下降

4.31. 基本数据类型包装类

- 八种包装类型：

基本数据类型	包装类
int	Integer
char	Character
float	Float
double	Double
boolean	Boolean
byte	Byte
short	Short
long	Long

  - **Number**：Integer、Short、Long、Double、Float、Byte都是Number的子类表示是一个数字
  - **Object**：Character、Boolean都是Object的直接子类
- 装箱和拆箱操作：

方法	描述
byteValue()	Byte
doubleValue()	Double→double
folatValue()	Float→float
intValue()	Integer→int
longValue()	Long→long
shortValue()	Short→short

    - 将一个基本数据类型转换为包装类，这样的操作称为装箱操作。将一个包装类转换为一个基本数据类型，这样的操作称为拆箱操作
- 转型操作：
    - 在包装类中，可以将一个字符串变为指定的基本数据类型，一般在输入数据时使用较多
        - 在Integer类中将String变为int类型数据：public static int parseInt(String s)
        - 在Float类中将String变为float类型数据：public static float parseFloat(String s)
    - 注意：转型操作时，字符串必须由数字组成，否则会出现错误
- 面试题：

```
// 面试题
Integer x1 = new Integer(10);
Integer x2 = new Integer(10);
System.out.println(x1 == x2);// false
System.out.println(x1.equals(x2));// true

Integer x3 = new Integer(128);
Integer x4 = new Integer(128);
System.out.println(x3 == x4);// false
System.out.println(x3.equals(x4));// true

Integer x5 = 10;
Integer x6 = 10;
System.out.println(x5 == x6);// true
System.out.println(x5.equals(x6));// true

Integer x7 = 128;
Integer x8 = 128;
System.out.println(x7 == x8);// false
System.out.println(x7.equals(x8));// true

// 享元模式
// 它使用共享对象，用来尽可能减少内存使用量以及分享资讯给尽可能多的相似对象；它适合用于当大量对象只是重复因而无法令人接受的使用大量内存。通常对象中的部分状态是可以分享。常见做法是把它们放在外部数据结构，当需要使用时再将它们传递给享元
```

```
24
25 // 运用共享技术有效的支持大量细粒度的对象
```

4.32. 包与访问修饰符

- 包：
  - 包对于多个java源文件的管理，就像我们的文件目录一样
  - 包的定义格式：

```
1 package com.vince;
2 // 该句只能出现在代码中的第一句
```

- 访问修饰符：

访问修饰符	同一个类	同一个包	不同包的子类	不同包的非子类
public	是	是	是	是
protected	是	是	是	
默认	是	是		
private	是			

4.33. OO原则总结

- **开闭原则**
  - 一个软件实体，如类、模块和函数应该对外扩展开放，对修改关闭
- **合成/聚用复用原则**
  - 新对象的某些功能在已创建好的对象里已实现，那么尽量用已有对象提供的功能，使之成为新对象的一部分，而不需要再创建
- **依赖倒置原则**
  - 高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象
- **接口隔离原则**
  - 客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上
- **迪米特法则**
  - 一个对象应该对其他对象保持最少的了解
- **里氏替换原则**
  - 所有引用基类的地方必须能透明的使用其子类的对象
- **单一职责原则**
  - 不要存在多于一个导致类变更的原因，即一个类只负责一项职责