互动小问题

第一章

需求明确，变化不大？瀑布过程

需求经常增加？增量过程

无完整的需求说明，只有一些基本需求？原型过程

第一章

把大象装冰箱，面向对象和结构化编程的思想分别是什么？

结构化编程：1.把冰箱门打开；2.把大象装进去；3.把冰箱门关上（出发点：怎么做）。

面向对象：分别找到大象、冰箱和人这三个对象，然后调用它们各自的方法来完成操作（出发点：是什么）。

第二章

- 进行自动类型转换时，会不会造成数据精度丢失呢？

double

float

long

int

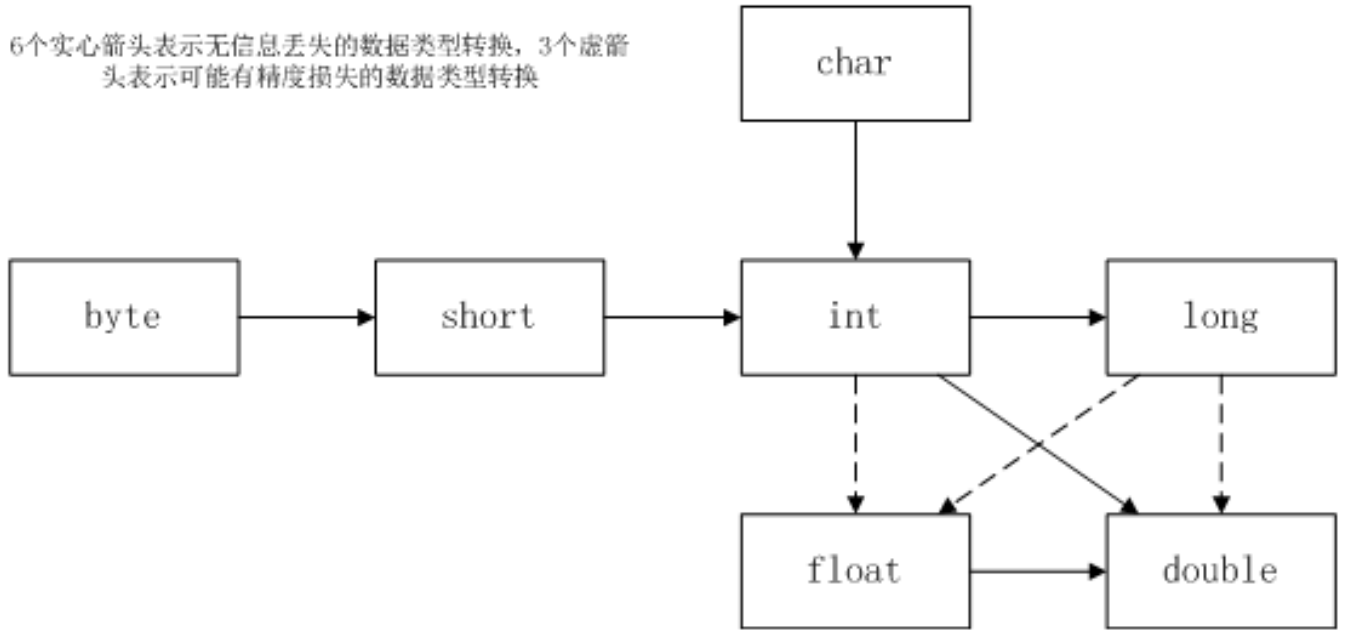
short

byte

高

低

从低到高，可以自动转换，但是不一定能保证精度不丢失。



第二章

请给出4段代码的运行结果

```
int i = 0;
while (i < 10) {
    i++;
    if(i % 2 == 0) {
        continue;   // 如果是偶数跳过当前循环
    }
    System.out.println(i);
} 1, 3, 5, 7, 9
```

```
int i = 0;
while (i < 10) {
    System.out.println(i);
    i++;
    if(i % 2 == 0) {
        continue;   // 如果是偶数跳过当前循环
    }
} 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

```
int i = 0;
while (i < 10) {
    if(i % 2 == 0) {
        continue;   // 如果是偶数跳过当前循环
    }
    System.out.println(i);
    i++;
} 没有输出
```

```
int i = 0;
while (i < 10) {
    i++;
    if(i % 2 == 0) {
        break;   // 如果是偶数跳过循环
    }
    System.out.println(i);
} 1
```

第二章

```
private String toHanStr(String numStr) {
String result = "";
int numLen = numStr.length();

//依次遍历数字字符串的每一位数字
for (int i = 0 ; i < numLen ; i++ ) {
//把char型数字转换成的int型数字，它们的ASCII码值恰好相差48
    int num = numStr.charAt(i) - 48;

    //如果不是最后一位数字，而且数字不是零，则需要添加单位(千、百、十)
    if ( i != numLen - 1 && num != 0) {
        result += hanArr[num] + unitArr[numLen - 2 - i];
    }
    //否则不要添加单位
    else {
        //上一个数是否为“零”，不为“零”时就添加
        if(result.length()>0 && hanArr[num].equals("零") &&
        result.charAt(result.length()-1)=='零’)
                continue;
        result += hanArr[num];
    }
}
```

完成toHanStr: 将一个整数字符串转换为汉字读法字符串。比如“1123”转换为“一千一百二十三”。

```
// 续
//只有个位数，直接返回
if(result.length()==1)
        return result;

int index=result.length()-1;
while(result.charAt(index)=='零‘){
        index--;
}

if(index!=result.length()-1)
        return
        result.substring(0,index+1
);
else {
        return result;
        }
```

互动小问题

❑异常的捕获处理的例子1：

```
public class Main {
    public static void main(String[] args) {
        try {
            int i = 1 / 0;//发生异常立即跳往catch语句中执行，不执行异常代码下面的代码
            System.out.println("输出结果为：" + i);
        } catch (ArithmeticException e) { //算数异常
            e.printStackTrace(); // 在命令行打印异常信息在程序中出错的位置及原因
            System.out.println("编译报错，除数不能为0");
            return; //是否执行finally? 执行，先finally再return

        }
        finally{
            System.out.println("finally");
        }
        System.out.println(“aaaa”); 写return后这条是否执行？ 不执行,先finally后直接return
    }
}
```

第二章

- 动手实现冒泡排序。输入为int数组。

```
public static void sort(int[] a) {
    //外层循环控制比较的次数
    for (int i = 0; i < a.length - 1; i++) {
      //内层循环控制到达位置
        for (int j = 0; j < a.length - i - 1; j++) {
            //前面的元素比后面大就交换
            if (a[j] > a[j + 1]) {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}
```

第三章

- **思考：** static是否破坏面向对象的特性？
  - 属于类，而非具体对象
  - 初始化加载到内存，被所有对象所共享（一定程度上的全局属性）
  - 保持类的封装性

第三章

- **思考：** 被static修饰的方法有没有this关键字？

不可以使用

- this,指向对象
- static，先于对象而存在，属于类的

第三章

- **思考：**静态成员的值能不能被对象改变？

可以被改变

具体可参照第一章代码Demo.java

第三章

思考：

```
public class Test {
   public Test(){//构造方法
            System.out.println("A的构造方法");
   }
   static {//静态代码块
            System.out.println("A的静态代码块");
   }
   {//初始化块
            System.out.println("A的初始化块");
   }
   public static void main(String[] args) {
   }
}
```

```
public class Test {
   public Test(){//构造方法
            System.out.println("A的构造方法");
   }
   static {//静态代码块
            System.out.println("A的静态代码块");
   }
   {//初始化块
            System.out.println("A的初始化块");
   }
   public static void main(String[] args) {
      Test a = new Test();
   }
}
```

```
public class Test {
   public Test(){//构造方法
            System.out.println("A的构造方法");
   }
   static {//静态代码块
            System.out.println("A的静态代码块");
   }
   {//初始化块
            System.out.println("A的初始化块");
   }
   public static void main(String[] args) {
         Test a = new Test();
         Test b = new Test();
   }
}
```

具体可参照第一章代码Test.java

第三章

思考：

```
public class Test2 {
    public static String staticField = "静态变量";
    public String field = "变量";
    static {
        System.out.println( staticField );
        System.out.println( "静态初始化块" );
    }
    {
        System.out.println( field );
        System.out.println( "初始化块" );
    }
    public Test2()
    {
        System.out.println( "构造方法" );
    }
    public static void main( String[] args )
    {
        new Test2();
    }
}
```

执行顺序：

静态变量

静态初始化块

变量

初始化块

构造方法

具体可参照第一章代码Test2.java

第三章

思考：

```
public class Test3{
    public static void main(String[] args) {
        System.out.println(cnt);
    }
    static int cnt=6;
    static{
        cnt+=9;
    }
    static{
        cnt/=3;
    }
}
```

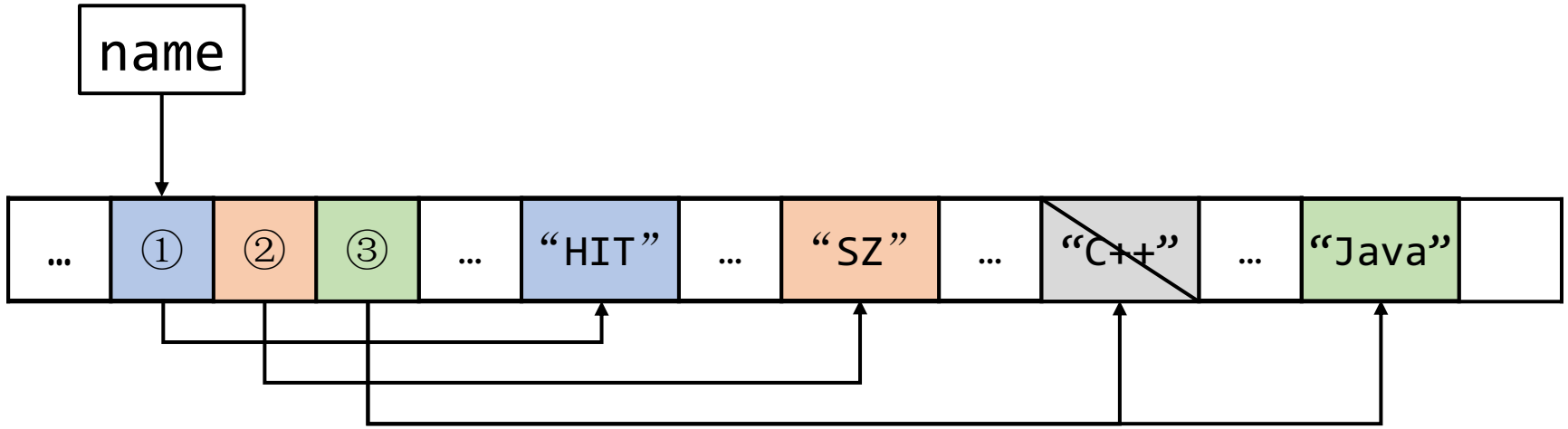
结果是5

执行顺序优先级：静态块,main(), 初始块,构造方法

具体可参照第一章代码Test3.java

- 思考

```
String[] names = {“HIT”, “SZ”, “C++”};
String s = names[2];
names[2] = “Java”;
System.out.println(s); // s是“C++”还是“Java”
```



结果是***C++***

- 抽象方法的访问修饰符可以有哪些？

可以被子类访问的：public，protected，default（子类在一个包里）

- 抽象方法是否可以被static修饰？

不可以

- static修饰的方法，无需实例化就可调用，类无法调用抽象方法；
- 抽象方法必须被继承实现才可实例化。

- 为什么需要抽象类？
  - 定义了规范，具体实现还依赖于子类
  - 更高层的抽象，以抽象类作为子类的模版，从而避免了子类设计的随意性

第四章

- 接口是特殊的抽象类吗？

不是，二者没有关联

- 接口不能有构造方法，抽象类可以有吗？

可以有，但是构造方法是给子类用的

- 接口可以有方法体，抽象类可以有吗？

可以有

- 接口可以有静态方法，抽象类可以有吗？

可以有，并且可以直接调用

- 接口中变量必须是public static final修饰，抽象类也是吗？

不是

this和super的不同点是什么？

this是调用本类的属性和方法，super是调用父类的属性和方法

this和super的相同点是什么？

只能在首行，都不可以在static环境中使用，因为都指的是对象

互动小问题

- 子类能不能**继承**父类的**私有**属性？
  - 不能，官方文档：https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

Private Members in a Superclass

A subclass does not inherit the `private` members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

- 子类能不能**访问**父类的**私有**属性？
  - 可以，两种方式：1.子类调用父类的构造方法；2.父类定义一个公开的方法来返回自己的私有属性值

简单工厂模式 (SimpleFactory Pattern)是否满足开闭原则呢？

不满足，如果新增加了榴莲pizza，需要修改factory

简单工厂模式 (SimpleFactory Pattern)似乎只是把问题搬到另一个对象罢了，问题依然存在，是吗？

虽然**不满足开闭原则**，但是也有好处，目前只看到一个客户类PizzaStore，可能还会有一个新的类PizzaStoreMenu，可利用这个工厂去取得价格和描述；可以有许多客户，跟具体pizza相关的代码需要修改的时候，只改这个类，客户都不需要改，体现了**单一职责原则**的重要性。

工厂模式 (SimpleFactory Pattern)中某个加盟店（NYPizzaStore）想新增加一个口味怎么办？

如果要满足开闭原则，只能新开个加盟店。

(二）判定覆盖

满足判定覆盖是否满足语句覆盖呢？

当然，每个分支都通过了，每个语句当然通过了。

(三）条件覆盖

满足条件覆盖是否满足判定覆盖呢？

满足条件覆盖是否满足语句覆盖呢？

不一定

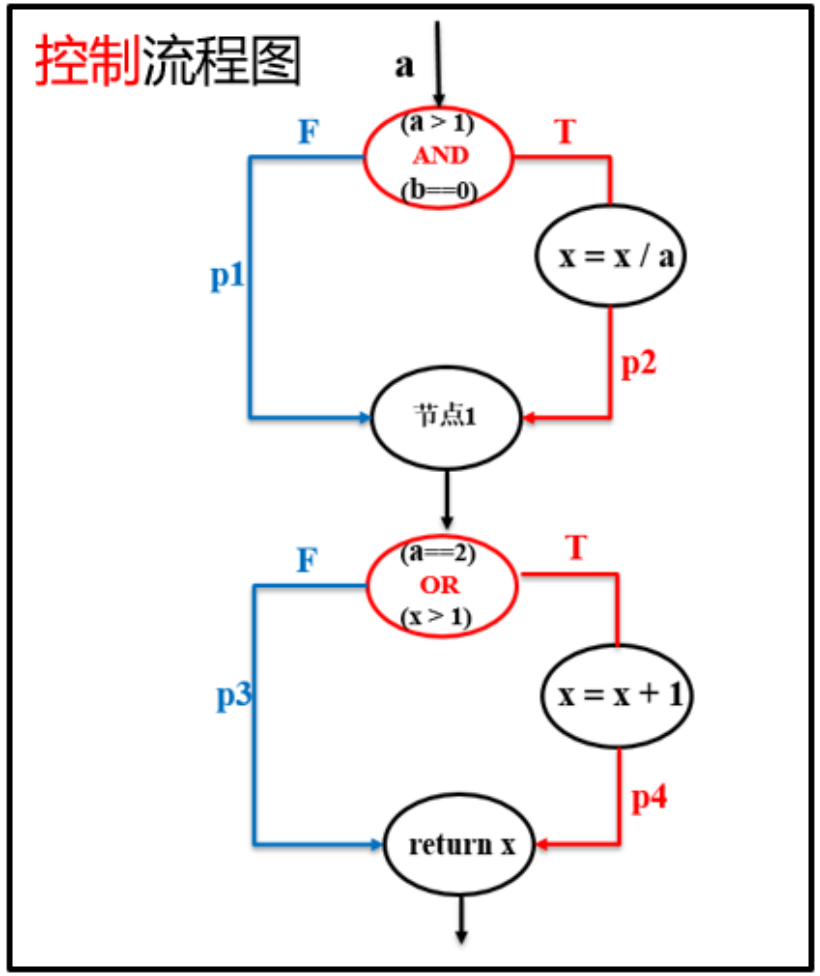
对语句IF(A AND B)THEN S设计测试用例使其满足"条件覆盖", A真B假,以及A假B真,但是它们都未能使语句S得以执行。

不一定，ppt的例子中，就没有覆盖语句x=x/a 。

第六章

(四）判定/条件覆盖

似乎很完美了，但是计算机是怎么对多个条件做出判定的？



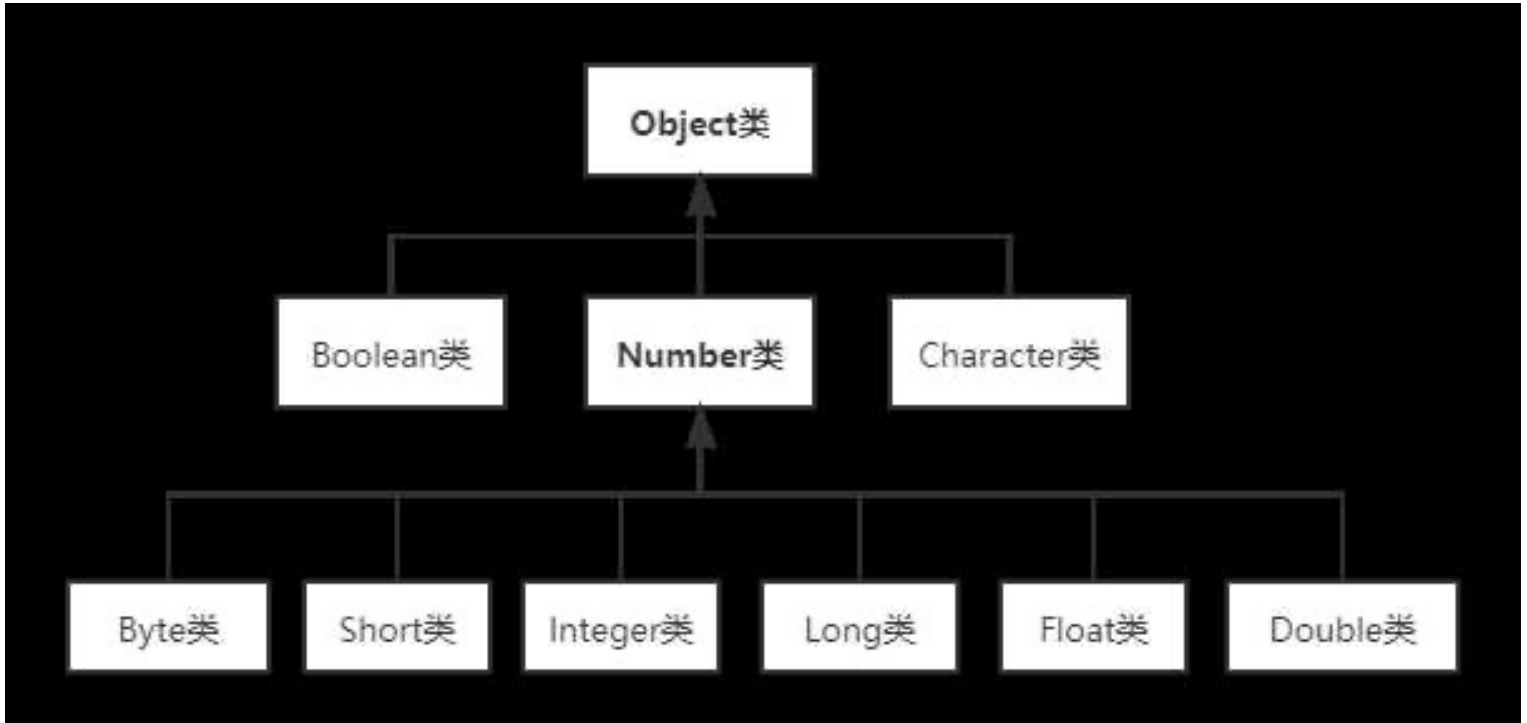
大多数计算机不能用一条指令对多个条件作出判定，而必须将源程序中对多个条件的判定分解成几个简单判定，所以较彻底的测试应使每一个简单判定都真正取到各种可能的结果。

第七章

如果只想放**整数**，下面代码怎么修改？

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<Integer> sites = new ArrayList<>();
        sites.add(1);
        sites.add(23);
        sites.add(456);
        System.out.println(sites);
    }
}
```



第七章

数组能不能存放不同类型的数据？

可以，需要声明Object.

第七章

在列表中间添加元素时，是否永远都是LinkedList比ArrayList快？

一般来说，由于ArrayList底层是数组，添加数据需要移动后面的数据，而LinkedList使用的是链表，直接移动指针就行，所以应该是LinkedList更快.

然而插入位置的选取对LinkedList有很大的影响，因为LinkedList在插入时需要**先移动指针到指定节点，才能开始插入，一旦要插入的位置比较远**，LinkedList就需要一步一步的移动指针，直到移动到插入位置，**所以插入节点值越大， LinkedList花费时间越长， 因为指针移动需要时间**。而ArrayList是数据结构， 可以根据下标直接获得位置， 这就省去了查找特定节点的时间，所以对ArrayList的影响不是特别大。

总结： 虽然有上述情况存在， 可是因为ArrayList可以使用下标直接获取数据，所以在使用查询的时候一般选择ArrayList，而进行删除和增加时，LinkedList比较方便，所以一般还是使用LinkedList比较多。

第七章

- 遍历HashSet集合时，能否使用for循环？

不可以

没有index 所以不能用范围for循环

第七章

工厂模式 vs. 策略模式

场景：飞机有很多种，有一个工厂专门负责生产各种需求的飞机。

工厂模式的关注点：
1）根据你给出的目的来生产不同用途的飞机，例如要运人，那么工厂生产客机，要运货就生产货机。
2）即根据你给出一些属性来生产不同行为的一类对象。

策略模式的关注点：
1）用工厂生产的飞机来做对应的事情，例如用货机来运货，用客机来运人。
2）即根据你给出对应的对象来执行对应的方法。

客户给它一个选择，它来帮客户创建一个对象。

客户给它一个对象，它来帮客户做对应的选择。

关注对象的创建：创建型模式

关注行为的选择：行为型模式

第七章

重看面向对象的思想

×可能带来额外的时间和空间开销。

✓面向更高的逻辑抽象层，可减少耦合，可封装变化，尽量避免错误蔓延。

第七章

如果要访问聚合对象中的各个元素，比如ArrayList, 可以用for来遍历，可以用for each 遍历。

1. 这种方式是否满足开闭原则？

不满足，因为更换遍历方法必须修改代码

2. 这种方式是否满足单一职责原则？

不满足，因为Arraylist不仅负责数据的存储，还负责了遍历，做了了两件事

第七章

迭代器模式是什么型模式？为什么？

行为型模式

主要是为了**关注集合类对象和使用者之间的交互**，看集合类对象和使用者之间是如何协作的。

- 一个流能不能既是输出流，又是输入流？

一个流只能是输入流或输出流的一种，在一个数据传输通道中，如果既要输入，又要输出，则要提供两个流。

- 输入和输出是相对哪里定义的？

都是相对内存（程序）来定义的。

第八章

- 读入word文档里的内容是字符流还是字节流？

字节流

Word文档不是纯文本，还有图片等内容。

第八章

- 在输入中，是将字节转为字符，在输出中是怎么转换？

- 输入：文件->字节流->**InputSreamReader**（字节转字符）->程序（字符流），将输入的字节流转换为字符流
- 输出：文件<-字节流<-**OutputSreamWriter**（字符转字节）<-程序（字符流），将输出的字符流转换为字节流

第八章

- 实际使用的时候路径分隔符/或者\\跟操作系统有关吗?

与平台无关。

第八章

- 只是单纯的把对象转成字节序列吗？
  - 把对象转成字节序列的时候需要制定一种**规则**（序列化），帮助还原对象
  - 把字节序列转成对象的时候再以这种**规则**（反序列化）把对象还原回来

第八章

- 被static修饰的字段可以被序列化吗？

不可以。

- **被static修饰的字段是针对类的**
- **序列化针对的是对象**

第八章

- 用户想更改删除数据，需要对数据库进行直接操作吗？
  - 不需要数据库的操作,因为DAO层就是**封装了数据库的操作**，用户不必为操作数据库感到苦恼。
  - DAO模式提供了**简单而统一的操作接口**，方便了对前端存储的管理。
- 想换一个数据库的话在哪里改？
  - 只会改DAO层而不会影响到服务层或者实体对象，**减少**了服务层与数据存储设备层之间的**耦合度**。

- 什么时候可以有多种监听器的情况？

比如鼠标可以单击、双击、右击、左击，就需要不同的监听器。

第九章

- 有没有生活中的MVC例子？

比如一家商场，完全可以分成三部分。

一部分是仓库，负责操作商品，这是**model**；

一部分是铺面，负责销售商品，这是**view**；

两者之间有个**controller**。

比如客户要提货了，所谓的**controller**就会通知仓库就提货，然后仓库告诉**controller**货已备好，**controller**通知铺面可以销售了，然后客户就可以提货了。

第九章

MVC优缺点？

- 优点：封装变化，减少耦合。
- 缺点：增加额外的时间和空间开销。

堆是线程共享的吗？

是的
堆放的是new出来的对象

在刚才买票的例子中(TicketDemo1.java), 怎么做到多线程共享资源?

```
private static int num = 10;
```

静态变量被所有对象共享

第十章

静态变量是线程安全的吗？

为所有对象共享，共享一份内存，一旦静态变量被修改，
其他对象均对修改可见，故线程不安全。

实例变量是线程安全的吗？

每个线程都在修改同一个对象的实例变量，则线程不安全；
如果每个线程执行都是在不同的对象中，那对象与对象之间的实例变量的修改互不影响，故线程安全。

局部变量是线程安全的吗？

每个线程执行时将会把局部变量放在各自的栈中，线程间不共享，
故不存在线程安全问题。

第十章

- sleep和wait方法的异同点？
  - 相同点：
    - ✓ 都可以让线程处于等待状态。
  - 不同点：
    - ✓ sleep方法必须指定时间；wait方法可以指定时间，也可以不指定。
    - ✓ sleep方法时间到，线程处于临时阻塞或者运行；wait如果没有指定时间，必须要通过notify或者 notifyAll()唤醒。
    - ✓ sleep方法是让当前线程休眠，让出cpu，但是不释放锁，这是Thread的静态方法；wait方法是让当前线程等待，释放锁，这是Object的方法。

第十章

```
public class DaemonTest {
    public static void main(String[] args){
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();}
                int sum = 0;
                for (int i = 1; i  <= 100; i++) {
                    sum = sum + i; }
                System.out.println("守护线程，最终求和的值为： " + sum);
            }
        });
        thread.setDaemon(true); //设置thread为守护线程
        thread. start();
        System.out.println("main 函数线程执行完毕， JVM 退出。");
    }
```

结果是什么？

System.out.println("main 函数线程执行完毕，
JVM 退出。");

守护线程在主线程执行完自动结束。

第十章

- Runnable和Callable的区别是什么？
  - ✓Runnable执行方法是run(),Callable是call()。
  - ✓Runnable无返回值且不会抛异常；Callable有返回值且可以向外抛异常。
  - ✓Runnable实现类创建的对象可以直接传入Thread启动线程；Callable实现类创建的对象不可以直接传入Thread启动线程。

第十一章

- 集合不强制转型是否可以取数据？

当然可以，声明Object 即可。

第十一章

- 如果泛型方法的泛型与泛型类声明的泛型名称一致，则泛型方法中的泛型会覆盖类的泛型，为什么？

方法中的泛型可以看做是独立于类定义的泛型而存在的，因为，就算不使用泛型类，也是可以直接使用泛型方法的。

第十一章

- Object是所有类型的父类，但是List<Object>确并不是List<String>的父类，为什么？

List<Object>里可能有String、Integer以及Double等类型的数据，与List<String>不构成继承关系。

第十一章

- 上例（无限定通配符）中，如果想改成类型参数T的形式应该怎么改？

```
public static <T> void printAllObject(List<T> list)
```

第十一章

- 上两个例子（上界和下界通配符）中，是否能改成类型参数T的形式？

不可以，类型参数T不能规定控制指定的类型范围。

第十一章

- 对于模板方法，怎么才能知道什么时候使用抽象方法，什么使用钩子呢？

当你的子类必须提供算法中某个方法或步骤的实现时，就使用抽象方法；如果时可选的，就用钩子。对于钩子，子类可以选择实现，也可以不实现。

第十一章

- 似乎应该保持抽象方法的数目越少越好，否则，在子类中实现这些方法将会很麻烦，是吗？

确实，算法内的步骤不要切割的太细，但是步骤太少也会没有弹性，所有要看情况折衷。一定要注意，擅用钩子，可选的就实现成钩子，而不是抽象方法，这样就能减轻子类的负荷。

第十一章

- 模版方法模式是类模式还是对象模式呢？

类模式，因为主要用于处理类与子类之间的关系。

第十一章

- 模版方法模式的优缺点是什么？

优点：减少耦合（**模板方法本身和这两个方法的具体实现之间被解耦了**）；封装变化（**子类封装了变化**）。

缺点：**带来额外的**时间和空间开销。

- 类的类型对象放在哪里呢？

Java中的Class也是一个类，所以Class对象也存放在堆当中。

- date1和date2是否相等？

不相等，因为通过反射调用构造函数又创建了一个新的对象，和之前的对象不相等。

第十二章

- 如果是传输文件，用哪个协议？
  - **TCP**
- 如果是视频/语音聊天，用哪个协议?
  - **UDP**

第十二章

- 域名和URL是什么关系？
  - URL包含了域名

第十二章

- **观察者模式经常应用在游戏开发领域，请寻找以下场景中的观察者和观察目标。**
  - ✓ 游戏主角移动到怪物的有效范围，怪物会袭击主角；
  - ✓ 游戏主角移动到陷阱的有效范围，陷阱会困住主角；
  - ✓ 游戏主角移动到道具的有效范围，道具会为主角加血。

观察者（怪物，陷阱，道具），目标（游戏主角）

第十二章

- **观察者模式和MVC模式之间是什么关系（都将显示逻辑和数据逻辑分离）？**
  - ✓ MVC模式应用了观察者模式，MVC中的模型（Model）可对应观察目标(Subject)， MVC中的视图（View）可对应观察者(Observer)，**模型的变化会引来视图的更新**。
  - ✓ 但是这里有两个问题：
    1）模型处理数据复杂的逻辑，又要通知视图，违背了单一职责原则；
    2）视图和模型耦合在一起，不利用复用。
  - ✓ 综上所述，就引入了控制器，专门负责监听并作为视图和模型之间的中介来使它俩解耦。

互动小问题

- **我们学到的哪些设计模式引入了“中介”来封装变化和减少耦合呢?**
  - ✓ 策略模式中的环境类
  - ✓ 数据访问对象模式中的DAO层
  - ✓ MVC模式中的控制器
  - ✓ 生产者消费者模式的buffer