1、面向对象的特征有哪些方面？

答：面向对象的特征主要有以下几个方面：

- 抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。

- 继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段（如果不能理解请阅读阎宏博士的《Java 与模式》或《设计模式精解》中关于桥梁模式的部分）。

- 封装：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口（可以想想普通洗衣机和全自动洗衣机的差别，明显全自动洗衣机封装更好因此操作起来更简单；我们现在使用的智能手机也是封装得足够好的，因为几个按键就搞定了所有的事情）。

- 多态性：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当 A 系统访问 B 系统提供的服务时，B 系统有多种提供服务的方式，但一切对 A 系统来说都是透明的（就像电动剃须刀是 A 系统，它的供电系统是 B 系统，B 系统可以使用电池供电或者用交流电，甚至还有可能是太阳能，A 系统只会通过 B 类对象调用供电的方法，但并不知道供电系统的底层实现是什么，究竟通过何种方式获得了动力）。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多

态性(也称为后绑定)。运行时的多态是面向对象最精髓的东西 ,要实现多态需要做两件事：1). 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；2). 对象造型（用父类型引用引用子类型对象 ,这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

2、访问修饰符 public,private,protected,以及不写（默认）时的区别？

答：

修饰符	当前类	同 包	子 类	其他包
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

类的成员不写访问修饰时默认为 default。默认对于同一个包中的其他类相当于公开（public），对于不是同一个包中的其他类相当于私有（private）。受保护（protected）对子类相当于公开，对不是同一包中的没有父子关系的类相当于私有。Java 中，外部类的修饰符只能是 public 或默认，类的成员（包括内部类）的修饰符可以是以上四种。

3、String 是最基本的数据类型吗？

答：不是。Java 中的基本数据类型只有 8 个：byte、short、int、long、float、double、char、boolean；除了基本类型（primitive type）和枚举类型（enumeration type），剩下的都是引用类型（reference type）。

4、float f=3.4;是否正确？

答:不正确。3.4 是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型

（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转换 float f =(float)3.4;或者写成 float f =3.4F;。

5、short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗？

答：对于 short s1 = 1; s1 = s1 + 1;由于 1 是 int 类型，因此 s1+1 运算结果也是 int 型，需要强制转换类型才能赋值给 short 型。而 short s1 = 1; s1 += 1;可以正确编译，因为 s1+= 1;相当于 s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

6、Java 有没有 goto？

答：goto 是 Java 中的保留字，在目前版本的 Java 中没有使用。（根据 James Gosling（Java 之父）编写的《The Java Programming Language》一书的附录中给出了一个 Java 关键字列表，其中有 goto 和 const，但是这两个是目前无法使用的关键字，因此有些地方将其称之为保留字，其实保留字这个词应该有更广泛的意义，因为熟悉 C 语言的程序员都知道，在系统类库中使用过的有特殊意义的单词或单词的组合都被视为保留字）

7、int 和 Integer 有什么区别？

答：Java 是一个近乎纯洁的面向对象编程语言，但是为了编程的方便还是引入了基本数据类型，但是为了能够将这些基本数据类型当成对象操作，Java 为每一个基本数据类型都引入了对应的包装类型（wrapper class），int 的包装类就是 Integer，从 Java 5 开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Java 为每个原始类型提供了包装类型：

- 原始类型: boolean，char，byte，short，int，long，float，double
- 包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

```
class AutoUnboxingTest {
```

```
    public static void main(String[] args) {
        Integer a = new Integer(3);
        Integer b = 3;                 // 将 3 自动装箱成 Integer 类型
        int c = 3;
        System.out.println(a == b);     // false 两个引用没有引用同一对象
        System.out.println(a == c);     // true a 自动拆箱成 int 类型再和 c 比
较
    }
}
```

最近还遇到一个面试题，也是和自动装箱和拆箱有点关系的，代码如下所示：

```
public class Test03 {

    public static void main(String[] args) {
        Integer f1 = 100, f2 = 100, f3 = 150, f4 = 150;

        System.out.println(f1 == f2);
        System.out.println(f3 == f4);
```

```
    }
}
```

如果不明就里很容易认为两个输出要么都是true要么都是false。首先需要注意的是f1、f2、f3、f4四个变量都是Integer对象引用，所以下面的==运算比较的不是值而是引用。装箱的本质是什么呢？当我们给一个Integer对象赋一个int值的时候，会调用Integer类的静态方法valueOf，如果看看valueOf的源代码就知道发生了什么。

```
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }
```

IntegerCache是Integer的内部类，其代码如下所示：

```
/**
     * Cache to support the object identity semantics of autoboxing for
 values between
```

```
     * -128 and 127 (inclusive) as required by JLS.
     *
     * The cache is initialized on first usage.  The size of the cache
     * may be controlled by the {@code -XX:AutoBoxCacheMax=<size>} opti
on.
     * During VM initialization, java.lang.Integer.IntegerCache.high p
roperty
     * may be set and saved in the private system properties in the
     * sun.misc.VM class.
     */

    private static class IntegerCache {
        static final int low = -128;
        static final int high;
        static final Integer cache[];

        static {
            // high value may be configured by property
            int h = 127;
            String integerCacheHighPropValue =
                sun.misc.VM.getSavedProperty("java.lang.Integer.Integer
Cache.high");
            if (integerCacheHighPropValue != null) {
                try {
                    int i = parseInt(integerCacheHighPropValue);
                    i = Math.max(i, 127);
```

```
                    // Maximum array size is Integer.MAX_VALUE
                    h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
                } catch( NumberFormatException nfe) {
                    // If the property cannot be parsed into an int, ignor
e it.
                }
            }
            high = h;

            cache = new Integer[(high - low) + 1];
            int j = low;
            for(int k = 0; k < cache.length; k++)
                cache[k] = new Integer(j++);

            // range [-128, 127] must be interned (JLS7 5.1.7)
            assert IntegerCache.high >= 127;
        }

        private IntegerCache() {}
    }
```

```
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
```

简单的说，如果整型字面量的值在-128 到 127 之间，那么不会 new 新的 Integer 对象，而是直接引用常量池中的 Integer 对象，所以上面的面试题中 f1==f2 的结果是 true，而 f3==f4 的结果是 false。

提醒：越是貌似简单的面试题其中的玄机就越多，需要面试者有相当深厚的功力。

8、&和&&的区别？

答：&运算符有两种用法：(1)按位与；(2)逻辑与。&&运算符是短路与运算。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是 true 整个表达式的值才是 true。&&之所以称为短路运算是因为，如果&&左边的表达式的值是 false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&&而不是&，例如在验证用户登录时判定用户名不是 null 而且不是空字符串 ,应当写为 :username != null &&!username.equals("")，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立 ,根本不能进行字符串的 equals 比较 ,否则会产生 NullPointerException 异常。注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

补充：如果你熟悉 JavaScript，那你可能更能感受到短路运算的强大，想成为 JavaScript 的高手就先从玩转短路运算开始吧。

9、解释内存中的栈(stack)、堆(heap)和静态区(static area)的用法。

答：通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用内存中的栈空间；而通过 new 关键字和构造器创建的对象放在堆空间；程序中的字面量（literal）如直接书写的 100、"hello"和常量都是放在静态区中。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，理论上整个内存没有被其他进程使用的空间甚至硬盘上的虚拟内存都可以被当成堆空间来使用。

```
String str = new String("hello");
```

上面的语句中变量 str 放在栈上，用 new 创建出来的字符串对象放在堆上，而"hello"这个字面量放在静态区。

补充：较新版本的 Java（从 Java 6 的某个更新开始）中使用了一项叫"逃逸分析"的技术，可以将一些局部对象放在栈上以提升对象的操作性能。

10、Math.round(11.5) 等于多少？Math.round(-11.5)等于多少？

答：Math.round(11.5)的返回值是 12，Math.round(-11.5)的返回值是-11。四舍五入的原理是在参数上加 0.5 然后进行下取整。

11、swtich 是否能作用在 byte 上，是否能作用在 long 上，是否能作用在 String 上？

答：在 Java 5 以前，switch(expr)中，expr 只能是 byte、short、char、int。从 Java 5 开始，Java 中引入了枚举类型，expr 也可以是 enum 类型，从 Java 7 开始，expr 还可以是字符串（String），但是长整型（long）在目前所有的版本中都是不可以的。

12、用最有效率的方法计算 2 乘以 8？

答： 2 << 3（左移 3 位相当于乘以 2 的 3 次方，右移 3 位相当于除以 2 的 3 次方）。

补充：我们为编写的类重写 hashCode 方法时，可能会看到如下所示的代码，其实我们不太理解为什么要使用这样的乘法运算来产生哈希码(散列码),而且为什么这个数是个素数，为什么通常选择 31 这个数？前两个问题的答案你可以自己百度一下，选择 31 是因为可以用移位和减法运算来代替乘法，从而得到更好的性能。说到这里你可能已经想到了：31 * num 等价于(num << 5) - num，左移 5 位相当于乘以 2 的 5 次方再减去自身就相当于乘以 31，现在的 VM 都能自动完成这个优化。

```
public class PhoneNumber {
```

```
    private int areaCode;
    private String prefix;
    private String lineNumber;

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + areaCode;
        result = prime * result
                + ((lineNumber == null) ? 0 : lineNumber.hashCode());
        result = prime * result + ((prefix == null) ? 0 : prefix.hashCod
e());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        PhoneNumber other = (PhoneNumber) obj;
```

```
        if (areaCode != other.areaCode)
            return false;
        if (lineNumber == null) {
            if (other.lineNumber != null)
                return false;
        } else if (!lineNumber.equals(other.lineNumber))
            return false;
        if (prefix == null) {
            if (other.prefix != null)
                return false;
        } else if (!prefix.equals(other.prefix))
            return false;
        return true;
    }

}
```

```
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
```

13、数组有没有 length()方法？String 有没有 length()方法？

答：数组没有 length()方法，有 length 的属性。String 有 length()方法。JavaScript 中，获得字符串的长度是通过 length 属性得到的，这一点容易和 Java 混淆。

14、在 Java 中，如何跳出当前的多重嵌套循环？

答：在最外层循环前加一个标记如 A，然后用 break A;可以跳出多重循环。（Java 中支持带标签的 break 和 continue 语句，作用有点类似于 C 和 C++中的 goto 语句，但是就像要避免使用 goto 一样，应该避免使用带标签的 break 和 continue，因为它不会让你的程序变得更优雅，很多时候甚至有相反的作用，所以这种语法其实不知道更好）

15、构造器（constructor）是否可被重写（override）？

答：构造器不能被继承，因此不能被重写，但可以被重载。

16、两个对象值相同(x.equals(y) == true)，但却可有不同的 hash code，这句话对不对？

答：不对，如果两个对象 x 和 y 满足 x.equals(y) == true，它们的哈希码（hash code）应当相同。Java 对于 eqauls 方法和 hashCode 方法是这样规定的：(1)如果两个对象相同（equals 方法返回 true），那么它们的 hashCode 值一定要相同；(2)如果两个对象的 hashCode 相同，它们并不一定相同。当然，你未必要按照要求去做，但是如果你违背了上述原则就会发现在使用容器时，相同的对象可以出现在 Set 集合中，同时增加新元素的效率会大大下降(对于使用哈希存储的系统，如果哈希码频繁的冲突将会造成存取性能急剧下降)。

补充：关于 equals 和 hashCode 方法，很多 Java 程序都知道，但很多人也就是仅仅知道而已，在 Joshua Bloch 的大作《Effective Java》（很多软件公司，《Effective Java》、《Java 编程思想》以及《重构：改善既有代码质量》是 Java 程序员必看书籍，如果你还没看过，那就赶紧去亚马逊买一本吧）中是这样介绍 equals 方法的：首先 equals 方法必须满足自反性（x.equals(x)必须返回 true）、对称性（x.equals(y)返回 true 时，y.equals(x)也必须返回 true）、传递性（x.equals(y)和 y.equals(z)都返回 true 时，x.equals(z)也必须返回 true）和一致性（当 x 和 y 引用的对象信息没有被修改时，多次调用 x.equals(y)应该得到同样的返回值），而且对于任何非 null 值的引用 x，x.equals(null)必须返回 false。实现高质量的 equals 方法的诀窍包括：1. 使用==操作符检查"参数是否为这个对象的引用"；2. 使用 instanceof 操作符检查"参数是否为正确的类型"；3. 对于类中的关键属性，检查参数传入对象的属性是否与之相匹配；4. 编写完 equals 方法后，问自己它是否满足对称性、

传递性、一致性；5. 重写 equals 时总是要重写 hashCode；6. 不要将 equals 方法参数中的 Object 对象替换为其他的类型，在重写时不要忘掉@Override 注解。

17、是否可以继承 String 类？

答：String 类是 final 类，不可以被继承。

补充：继承 String 本身就是一个错误的行为，对 String 类型最好的重用方式是关联关系（Has-A）和依赖关系（Use-A）而不是继承关系（Is-A）。

18、当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递？

答：是值传递。Java 语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。C++和 C#中可以通过传引用或传输出参数来改变传入的参数的值。在 C#中可以编写如下所示的代码，但是在 Java 中却做不到。

```
using System;


namespace CS01 {


    class Program {
        public static void swap(ref int x, ref int y) {
            int temp = x;
            x = y;
            y = temp;
        }
```

```
        public static void Main (string[] args) {
            int a = 5, b = 10;
            swap (ref a, ref b);
            // a = 10, b = 5;
            Console.WriteLine ("a = {0}, b = {1}", a, b);
        }
    }
}
```

说明：Java 中没有传引用实在是非常的不方便，这一点在 Java 8 中仍然没有得到改进，正是如此在 Java 编写的代码中才会出现大量的 Wrapper 类（将需要通过方法调用修改的引用置于一个 Wrapper 类中，再将 Wrapper 对象传入方法），这样的做法只会让代码变得臃肿，尤其是让从 C 和 C++转型为 Java 程序员的开发者无法容忍。

19、String 和 StringBuilder、StringBuffer 的区别？

答：Java 平台提供了两种类型的字符串：String 和 StringBuffer/StringBuilder，它们可以储存和操作字符串。其中 String 是只读字符串，也就意味着 String 引用的字符串内容是不能被改变的。而 StringBuffer/StringBuilder 类表示的字符串对象可以直接进行修改。StringBuilder 是 Java 5 中引入的，它和 StringBuffer 的方法完全相同，区别在于它是在单线程环境下使用的，因为它的所有方面都没有被 synchronized 修饰，因此它的效率也比 StringBuffer 要高。

面试题 1 - 什么情况下用+运算符进行字符串连接比调用 StringBuffer/StringBuilder 对象的 append 方法连接字符串性能更好？

面试题 2 - 请说出下面程序的输出。

```
class StringEqualTest {


    public static void main(String[] args) {

        String s1 = "Programming";

        String s2 = new String("Programming");

        String s3 = "Program" + "ming";

        System.out.println(s1 == s2);

        System.out.println(s1 == s3);

        System.out.println(s1 == s1.intern());

    }

}
```

- 1
- 2

```
3
4
5
6
7
8
9
10
11
```

补充：String 对象的 intern 方法会得到字符串对象在常量池中对应的版本的引用（如果常量池中有一个字符串与 String 对象的 equals 结果是 true），如果常量池中没有对应的字符串，则该字符串将被添加到常量池中，然后返回常量池中字符串的引用。

20、重载（Overload）和重写（Override）的区别。重载的方法能否根据返回类型进行区分？

答：方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。重载发生在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载；重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常(里氏代换原则)。重载对返回类型没有特殊的要求。

面试题：华为的面试题中曾经问过这样一个问题 - "为什么不能根据返回类型来区分重载"，快说出你的答案吧！

21、描述一下 JVM 加载 class 文件的原理机制？

答：JVM 中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java 中的类加载器是一个重要的 Java 运行时系统组件，它负责在运行时查找和装入类文件中的类。

由于 Java 的跨平台性，经过编译的 Java 源程序并不是一个可执行程序，而是一个或多个类文件。当 Java 程序需要使用某个类时，JVM 会确保这个类已经被加载、连接（验证、准

备和解析）和初始化。类的加载是指把类的.class 文件中的数据读入到内存中，通常是创建一个字节数组读入.class 文件，然后产生与所加载类对应的 Class 对象。加载完成后，Class 对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后 JVM 对类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader 的子类）。从 Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM 更好的保证了 Java 平台的安全性，在该机制中，JVM 自带的 Bootstrap 是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM 不会向 Java 程序提供对 Bootstrap 的引用。下面是关于几个类加载器的说明：

- Bootstrap：一般用本地代码实现，负责加载 JVM 基础核心类库（rt.jar）；
- Extension：从 java.ext.dirs 系统属性所指定的目录中加载类库，它的父加载器是 Bootstrap；
- System：又叫应用类加载器，其父类是 Extension。它是应用最广泛的类加载器。它从环境变量 classpath 或者系统属性 java.class.path 所指定的目录中记载类，是用户自定义加载器的默认父加载器。

22、char 型变量中能不能存贮一个中文汉字，为什么？

答：char 类型可以存储一个中文汉字，因为 Java 中使用的编码是 Unicode（不选择任何

特定的编码，直接使用字符在字符集中的编号，这是统一的唯一方法），一个 char 类型占 2 个字节（16 比特），所以放一个中文是没问题的。

补充：使用 Unicode 意味着字符在 JVM 内部和外部有不同的表现形式，在 JVM 内部都是 Unicode，当这个字符被从 JVM 内部转移到外部时（例如存入文件系统中），需要进行编码转换。所以 Java 中有字节流和字符流，以及在字符流和字节流之间进行转换的转换流，如 InputStreamReader 和 OutputStreamReader，这两个类是字节流和字符流之间的适配器类，承担了编码转换的任务；对于 C 程序员来说，要完成这样的编码转换恐怕要依赖于 union（联合体/共用体）共享内存的特征来实现了。

23、抽象类（abstract class）和接口（interface）有什么异同？

答：抽象类和接口都不能够实例化，但可以定义抽象类和接口类型的引用。一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。接口比抽象类更加抽象，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全部都是抽象方法。抽象类中的成员可以是 private、默认、protected、public 的，而接口中的成员全都是 public 的。抽象类中可以定义成员变量，而接口中定义的成员变量实际上都是常量。有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。

24、静态嵌套类(Static Nested Class)和内部类（Inner Class）的不同？

答：Static Nested Class 是被声明为静态（static）的内部类，它可以不依赖于外部类实例被实例化。而通常的内部类需要在外部类实例化后才能实例化，其语法看起来挺诡异的，如下所示。

```
/**
```

```
 * 扑克类（一副扑克）
 * @author 骆昊
 *
 */
public class Poker {
    private static String[] suites = {"黑桃", "红桃", "草花", "方块"};
    private static int[] faces = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
 13};

    private Card[] cards;

    /**
     * 构造器
     *
     */
    public Poker() {
        cards = new Card[52];
        for(int i = 0; i < suites.length; i++) {
            for(int j = 0; j < faces.length; j++) {
                cards[i * 13 + j] = new Card(suites[i], faces[j]);
            }
        }
    }

    /**
```

```
     * 洗牌 （随机乱序）
     *
     */
    public void shuffle() {
        for(int i = 0, len = cards.length; i < len; i++) {
            int index = (int) (Math.random() * len);
            Card temp = cards[index];
            cards[index] = cards[i];
            cards[i] = temp;
        }
    }

    /**
     * 发牌
     * @param index 发牌的位置
     *
     */
    public Card deal(int index) {
        return cards[index];
    }

    /**
     * 卡片类（一张扑克）
     * [内部类]
```

```
 * @author 骆昊
 *
 */
public class Card {
    private String suite;   // 花色
    private int face;       // 点数

    public Card(String suite, int face) {
        this.suite = suite;
        this.face = face;
    }

    @Override
    public String toString() {
        String faceStr = "";
        switch(face) {
        case 1: faceStr = "A"; break;
        case 11: faceStr = "J"; break;
        case 12: faceStr = "Q"; break;
        case 13: faceStr = "K"; break;
        default: faceStr = String.valueOf(face);
        }
        return suite + faceStr;
    }
```

```
    }

}
```

```
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
```

测试代码：

```
class PokerTest {


    public static void main(String[] args) {

        Poker poker = new Poker();
```

```
        poker.shuffle();                // 洗牌
        Poker.Card c1 = poker.deal(0);  // 发第一张牌
        // 对于非静态内部类 Card
        // 只有通过其外部类 Poker 对象才能创建 Card 对象
        Poker.Card c2 = poker.new Card("红心", 1);    // 自己创建一张牌

        System.out.println(c1);     // 洗牌后的第一张
        System.out.println(c2);     // 打印：红心 A
    }
}
```

面试题 - 下面的代码哪些地方会产生编译错误？

```
class Outer {

    class Inner {}

    public static void foo() { new Inner(); }
```

```
    public void bar() { new Inner(); }


    public static void main(String[] args) {
        new Inner();
    }
}
```

注意：Java 中非静态内部类对象的创建要依赖其外部类对象，上面的面试题中 foo 和 main 方法都是静态方法，静态方法中没有 this，也就是说没有所谓的外部类对象，因此无法创建内部类对象，如果要在静态方法中创建内部类对象，可以这样做：

```
    new Outer().new Inner();
```

25、Java 中会存在内存泄漏吗，请简单描述。

答：理论上 Java 因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是 Java 被广泛使用于服务器端编程的一个重要原因）然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被 GC 回收，因此也会导致内存泄露的发生。例如 Hibernate 的 Session（一

级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象，如果不及时关闭（close）或清空（flush）一级缓存就可能导致内存泄露。下面例子中的代码也会导致内存泄露。

```
import java.util.Arrays;
import java.util.EmptyStackException;

public class MyStack<T> {
    private T[] elements;
    private int size = 0;

    private static final int INIT_CAPACITY = 16;

    public MyStack() {
        elements = (T[]) new Object[INIT_CAPACITY];
    }

    public void push(T elem) {
        ensureCapacity();
        elements[size++] = elem;
    }

    public T pop() {
        if(size == 0)
            throw new EmptyStackException();
```

```
        return elements[--size];
    }

    private void ensureCapacity() {
        if(elements.length == size) {
            elements = Arrays.copyOf(elements, 2 * size + 1);
        }
    }
}
```

```
29
30
```

上面的代码实现了一个栈（先进后出（FILO））结构，乍看之下似乎没有什么明显的问题，它甚至可以通过你编写的各种单元测试。然而其中的 pop 方法却存在内存泄露的问题，当我们用 pop 方法弹出栈中的对象时，该对象不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，因为栈内部维护着对这些对象的过期引用（obsolete reference）。在支持垃圾回收的语言中，内存泄露是很隐蔽的，这种内存泄露其实就是无意识的对象保持。如果一个对象引用被无意识的保留起来了，那么垃圾回收器不会处理这个对象，也不会处理该对象引用的其他对象，即使这样的对象只有少数几个，也可能会导致很多的对象被排除在垃圾回收之外，从而对性能造成重大影响，极端情况下会引发 Disk Paging（物理内存与硬盘的虚拟内存交换数据），甚至造成 OutOfMemoryError。

26、抽象的（abstract）方法是否可同时是静态的（static）,是否可同时是本地方法（native），是否可同时被 synchronized 修饰？

答：都不能。抽象方法需要子类重写，而静态的方法是无法被重写的，因此二者是矛盾的。本地方法是由本地代码（如 C 代码）实现的方法，而抽象方法是没有实现的，也是矛盾的。synchronized 和方法的实现细节有关，抽象方法不涉及实现细节，因此也是相互矛盾的。

27、阐述静态变量和实例变量的区别。

答：静态变量是被 static 修饰符修饰的变量，也称为类变量，它属于类，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个拷贝；实例变量必须依存于某一实例，需要先创建对象然后通过对象才能访问到它。静态变量可以实现让多个对象共享内存。

补充：在 Java 开发中，上下文类和工具类中通常会有大量的静态成员。

28、是否可以从一个静态（static）方法内部发出对非静态（non-static）方法的调用？

答：不可以，静态方法只能访问静态成员，因为非静态方法的调用要先创建对象，在调用静态方法时可能对象并没有被初始化。

29、如何实现对象克隆？

答：有两种方式：

1). 实现 Cloneable 接口并重写 Object 类中的 clone()方法；

2). 实现 Serializable 接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆，代码如下。

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;


public class MyUtil {


    private MyUtil() {
        throw new AssertionError();
    }


    public static <T> T clone(T obj) throws Exception {
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
```

```
        ObjectOutputStream oos = new ObjectOutputStream(bout);

        oos.writeObject(obj);



        ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByte
Array());

        ObjectInputStream ois = new ObjectInputStream(bin);

        return (T) ois.readObject();



        // 说明：调用 ByteArrayInputStream 或 ByteArrayOutputStream 对象的 cl
ose 方法没有任何意义

        // 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于对外
部资源（如文件流）的释放

    }

}
```

```
• 22
• 23
• 24
```

下面是测试代码：

```
import java.io.Serializable;


/**
 * 人类
 * @author 骆昊
 *
 */
class Person implements Serializable {
    private static final long serialVersionUID = -9102017020286042305
L;


    private String name;    // 姓名
    private int age;        // 年龄
    private Car car;        // 座驾


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

```
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 Car getCar() {
    return car;
}

public void setCar(Car car) {
    this.car = car;
}
```

```
    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + ", car=" + car
+ "]";
    }

}
```

```
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
```

```
/**
 * 小汽车类
 * @author 骆昊
 *
 */
class Car implements Serializable {
    private static final long serialVersionUID = -5713945027627603702
L;

    private String brand;       // 品牌
    private int maxSpeed;       // 最高时速

    public Car(String brand, int maxSpeed) {
        this.brand = brand;
```

```
    this.maxSpeed = maxSpeed;
}

public String getBrand() {
    return brand;
}

public void setBrand(String brand) {
    this.brand = brand;
}

public int getMaxSpeed() {
    return maxSpeed;
}

public void setMaxSpeed(int maxSpeed) {
    this.maxSpeed = maxSpeed;
}

@Override
public String toString() {
    return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";
}
```

```
}
```

```
class CloneTest {
```

```
    public static void main(String[] args) {
        try {
            Person p1 = new Person("Hao LUO", 33, new Car("Benz", 300));
            Person p2 = MyUtil.clone(p1);   // 深度克隆
            p2.getCar().setBrand("BYD");
            // 修改克隆的 Person 对象 p2 关联的汽车对象的品牌属性
            // 原来的 Person 对象 p1 关联的汽车不会受到任何影响
            // 因为在克隆 Person 对象时其关联的汽车对象也被克隆了
            System.out.println(p1);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

注意：基于序列化和反序列化实现的克隆不仅仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种是方案明显优于使用 Object 类的 clone 方法克隆对象。让问题在编译的时候暴露出来总是优于把问题留到运行时。

30、GC 是什么？为什么要有 GC？

答：GC 是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。Java 程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc() 或 Runtime.getRuntime().gc() ，但 JVM 可以屏蔽掉显示的垃圾回收调用。

垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。在 Java 诞生初期，垃圾回收是 Java 最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今 Java 的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得 iOS 的系统比 Android 系统有更好的用户体验，其中一个深层次的原因就在于 Android 系统中垃圾回收的不可预知性。

补充：垃圾回收机制有很多种，包括：分代复制垃圾回收、标记垃圾回收、增量垃圾回收等方式。标准的 Java 进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。Java 平台对堆内存回收和再利用的基本算法被称为标记和清除，但是 Java 对其进行了改进，采用“分代式垃圾收集”。这种方法会跟 Java 对象的生命周期将堆内存划分为不同

的区域，在垃圾收集过程中，可能会将对象移动到不同区域：

- 伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。
- 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。
- 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

与垃圾回收相关的 JVM 参数：

- -Xms / -Xmx — 堆的初始大小 / 堆的最大大小
- -Xmn — 堆中年轻代的大小
- -XX:-DisableExplicitGC — 让 System.gc()不产生任何作用
- -XX:+PrintGCDetails — 打印 GC 的细节
- -XX:+PrintGCDateStamps — 打印 GC 操作的时间戳
- -XX:NewSize / XX:MaxNewSize — 设置新生代大小/新生代最大大小
- -XX:NewRatio — 可以设置老生代和新生代的比例
- -XX:PrintTenuringDistribution — 设置每次新生代 GC 后输出幸存者乐园中对象年龄的分布
- -XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值
- -XX:TargetSurvivorRatio：设置幸存区的目标使用率

31、String s = new String("xyz");创建了几个字符串对象？

答：两个对象，一个是静态区的"xyz"，一个是用 new 创建在堆上的对象。

32、接口是否可继承（extends）接口？抽象类是否可实现（implements）接口？抽象类是否可继承具体类（concrete class）？

答：接口可以继承接口，而且支持多重继承。抽象类可以实现(implements)接口，抽象类可继承具体类也可以继承抽象类。

33、一个".java"源文件中是否可以包含多个类（不是内部类）？有什么限制？

答：可以，但一个源文件中最多只能有一个公开类（public class）而且文件名必须和公开类的类名完全保持一致。

34、Anonymous Inner Class(匿名内部类)是否可以继承其它类？是否可以实现接口？

答：可以继承其他类或实现其他接口，在 Swing 编程和 Android 开发中常用此方式来实现事件监听和回调。

35、内部类可以引用它的包含类（外部类）的成员吗？有没有什么限制？

答：一个内部类对象可以访问创建它的外部类对象的成员，包括私有成员。

36、Java 中的 final 关键字有哪些用法？

答：(1)修饰类：表示该类不能被继承；(2)修饰方法：表示方法不能被重写；(3)修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

37、指出下面程序的运行结果。

```
class A {

    static {
        System.out.print("1");
    }

    public A() {
        System.out.print("2");
    }
}

class B extends A{

    static {
        System.out.print("a");
    }

    public B() {
        System.out.print("b");
    }
}

public class Hello {
```

```
    public static void main(String[] args) {

        A ab = new B();

        ab = new B();

    }


}
```

答：执行结果：1a2b2b。创建对象时构造器的调用顺序是：先初始化静态成员，然后调用父类构造器，再初始化非静态成员，最后调用自身构造器。

提示 :如果不能给出此题的正确答案 ,说明之前第 21 题 Java 类加载机制还没有完全理解，赶紧再看看吧。

38、数据类型之间的转换：

- 如何将字符串转换为基本数据类型？

- 如何将基本数据类型转换为字符串？

答：

- 调用基本数据类型对应的包装类中的方法 parseXXX(String)或 valueOf(String)即可返回相应基本类型；

- 一种方法是将基本数据类型与空字符串（""）连接（+）即可获得其所对应的字符串；另一种方法是调用 String 类中的 valueOf()方法返回相应字符串

39、如何实现字符串的反转及替换？

答：方法很多，可以自己写实现也可以使用 String 或 StringBuffer/StringBuilder 中的方法。有一道很常见的面试题是用递归实现字符串反转，代码如下所示：

```
public static String reverse(String originStr) {
    if(originStr == null || originStr.length() <= 1)
        return originStr;
    return reverse(originStr.substring(1)) + originStr.charAt(0);
}
```

```
4
5
```

40、怎样将 GB2312 编码的字符串转换为 ISO-8859-1 编码的字符串？

答：代码如下所示：

```
String s1 = "你好";

String s2 = new String(s1.getBytes("GB2312"), "ISO-8859-1");
```

```
1
2
```

41、日期和时间：

- 如何取得年月日、小时分钟秒？

- 如何取得从 1970 年 1 月 1 日 0 时 0 分 0 秒到现在的毫秒数？

- 如何取得某月的最后一天？

- 如何格式化日期？

答：

问题 1：创建 java.util.Calendar 实例，调用其 get()方法传入不同的参数即可获得参数所对应的值。Java 8 中可以使用 java.time.LocalDateTimel 来获取，代码如下所示。

```
public class DateTimeTest {

    public static void main(String[] args) {

        Calendar cal = Calendar.getInstance();

        System.out.println(cal.get(Calendar.YEAR));

        System.out.println(cal.get(Calendar.MONTH));    // 0 - 11

        System.out.println(cal.get(Calendar.DATE));

        System.out.println(cal.get(Calendar.HOUR_OF_DAY));
```

```
        System.out.println(cal.get(Calendar.MINUTE));

        System.out.println(cal.get(Calendar.SECOND));


        // Java 8

        LocalDateTime dt = LocalDateTime.now();

        System.out.println(dt.getYear());

        System.out.println(dt.getMonthValue());     // 1 - 12

        System.out.println(dt.getDayOfMonth());

        System.out.println(dt.getHour());

        System.out.println(dt.getMinute());

        System.out.println(dt.getSecond());

    }

}
```

问题 2：以下方法均可获得该毫秒数。

```
Calendar.getInstance().getTimeInMillis();

System.currentTimeMillis();

Clock.systemDefaultZone().millis(); // Java 8
```

问题 3：代码如下所示。

```
Calendar time = Calendar.getInstance();

time.getActualMaximum(Calendar.DAY_OF_MONTH);
```

问题 4 利用 java.text.DataFormat 的子类(如 SimpleDateFormat 类)中的 format(Date) 方法可将日期格式化。Java 8 中可以用 java.time.format.DateTimeFormatter 来格式化时间日期，代码如下所示。

```
import java.text.SimpleDateFormat;

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

import java.util.Date;


class DateFormatTest {


    public static void main(String[] args) {
```

```
        SimpleDateFormat oldFormatter = new SimpleDateFormat("yyyy/MM/dd");

        Date date1 = new Date();

        System.out.println(oldFormatter.format(date1));



        // Java 8

        DateTimeFormatter newFormatter = DateTimeFormatter.ofPattern("yyyy/MM/dd");

        LocalDate date2 = LocalDate.now();

        System.out.println(date2.format(newFormatter));

    }

}
```

补充：Java 的时间日期 API 一直以来都是被诟病的东西，为了解决这一问题，Java 8 中引入了新的时间日期 API ,其中包括 LocalDate、LocalTime、LocalDateTime、Clock、Instant

等类，这些的类的设计都使用了不变模式，因此是线程安全的设计。如果不理解这些内容，可以参考我的另一篇文章《关于 Java 并发编程的总结和思考》。

42、打印昨天的当前时刻。

答：

```
import java.util.Calendar;


class YesterdayCurrent {
    public static void main(String[] args){
        Calendar cal = Calendar.getInstance();
        cal.add(Calendar.DATE, -1);
        System.out.println(cal.getTime());
    }
}
```

在 Java 8 中，可以用下面的代码实现相同的功能。

```
import java.time.LocalDateTime;


class YesterdayCurrent {
```

```
    public static void main(String[] args) {
        LocalDateTime today = LocalDateTime.now();
        LocalDateTime yesterday = today.minusDays(1);

        System.out.println(yesterday);
    }
}
```

43、比较一下 Java 和 JavaSciprt。

答 :JavaScript 与 Java 是两个公司开发的不同的两个产品。Java 是原 Sun Microsystems 公司推出的面向对象的程序设计语言，特别适合于互联网应用程序开发；而 JavaScript 是 Netscape 公司的产品，为了扩展 Netscape 浏览器的功能而开发的一种可以嵌入 Web 页面中运行的基于对象和事件驱动的解释性语言。JavaScript 的前身是 LiveScript；而 Java 的前身是 Oak 语言。

下面对两种语言间的异同作如下比较：

- 基于对象和面向对象：Java 是一种真正的面向对象的语言，即使是开发简单的程序，必须设计对象；JavaScript 是种脚本语言，它可以用来制作与网络无关的，与用户交互作用的复杂软件。它是一种基于对象(Object-Based)和事件驱动(Event-Driven)的编程语言，

因而它本身提供了非常丰富的内部对象供设计人员使用。

- 解释和编译：Java 的源代码在执行之前，必须经过编译。JavaScript 是一种解释性编程语言，其源代码不需经过编译，由浏览器解释执行。（目前的浏览器几乎都使用了 JIT（即时编译）技术来提升 JavaScript 的运行效率）

- 强类型变量和类型弱变量：Java 采用强类型变量检查，即所有变量在编译之前必须作声明；JavaScript 中变量是弱类型的，甚至在使用变量前可以不作声明，JavaScript 的解释器在运行时检查推断其数据类型。

- 代码格式不一样。

补充：上面列出的四点是网上流传的所谓的标准答案。其实 Java 和 JavaScript 最重要的区别是一个是静态语言，一个是动态语言。目前的编程语言的发展趋势是函数式语言和动态语言。在 Java 中类（class）是一等公民，而 JavaScript 中函数（function）是一等公民，因此 JavaScript 支持函数式编程，可以使用 Lambda 函数和闭包（closure），当然 Java 8 也开始支持函数式编程，提供了对 Lambda 表达式以及函数式接口的支持。对于这类问题，在面试的时候最好还是用自己的语言回答会更加靠谱，不要背网上所谓的标准答案。

44、什么时候用断言（assert）？

答 断言在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。一般来说，断言用于保证程序最基本、关键的正确性。断言检查通常在开发和测试时开启。为了保证程序的执行效率，在软件发布后断言检查通常是关闭的。断言是一个包含布尔表达式的语句，在执行这个语句时假定该表达式为 true；如果表达式的值为 false，那么系统会报告一个 AssertionError。断言的使用如下面的代码所示：

```
assert(a > 0); // throws an AssertionError if a <= 0
```

断言可以有两种形式：

assert Expression1;

assert Expression1 : Expression2 ;

Expression1 应该总是产生一个布尔值。

Expression2 可以是得出一个值的任意表达式；这个值用于生成显示更多调试信息的字符串消息。

要在运行时启用断言，可以在启动 JVM 时使用-enableassertions 或者-ea 标记。要在运行时选择禁用断言，可以在启动 JVM 时使用-da 或者-disableassertions 标记。要在系统类中启用或禁用断言，可使用-esa 或-dsa 标记。还可以在包的基础上启用或者禁用断言。

注意：断言不应该以任何方式改变程序的状态。简单的说，如果希望在不满足某些条件时阻止代码的执行，就可以考虑用断言来阻止它。

45、Error 和 Exception 有什么区别？

答：Error 表示系统级的错误和程序不必处理的异常，是恢复不是不可能但很困难的情况下的一种严重问题；比如内存溢出，不可能指望程序能处理这样的情况；Exception 表示需要捕捉或者需要程序进行处理的异常，是一种设计或实现问题；也就是说，它表示如果程序运行正常，从不会发生的情况。

面试题：2005 年摩托罗拉的面试中曾经问过这么一个问题“If a process reports a stack overflow run-time error, what’s the most possible cause?”，给了四个选项 a. lack of memory; b. write on an invalid memory space; c. recursive function calling; d. array index out of boundary. Java 程序在运行时也可能会遭遇 StackOverflowError，这是一个

无法恢复的错误，只能重新修改代码了，这个面试题的答案是 c。如果写了不能迅速收敛的递归，则很有可能引发栈溢出的错误，如下所示：

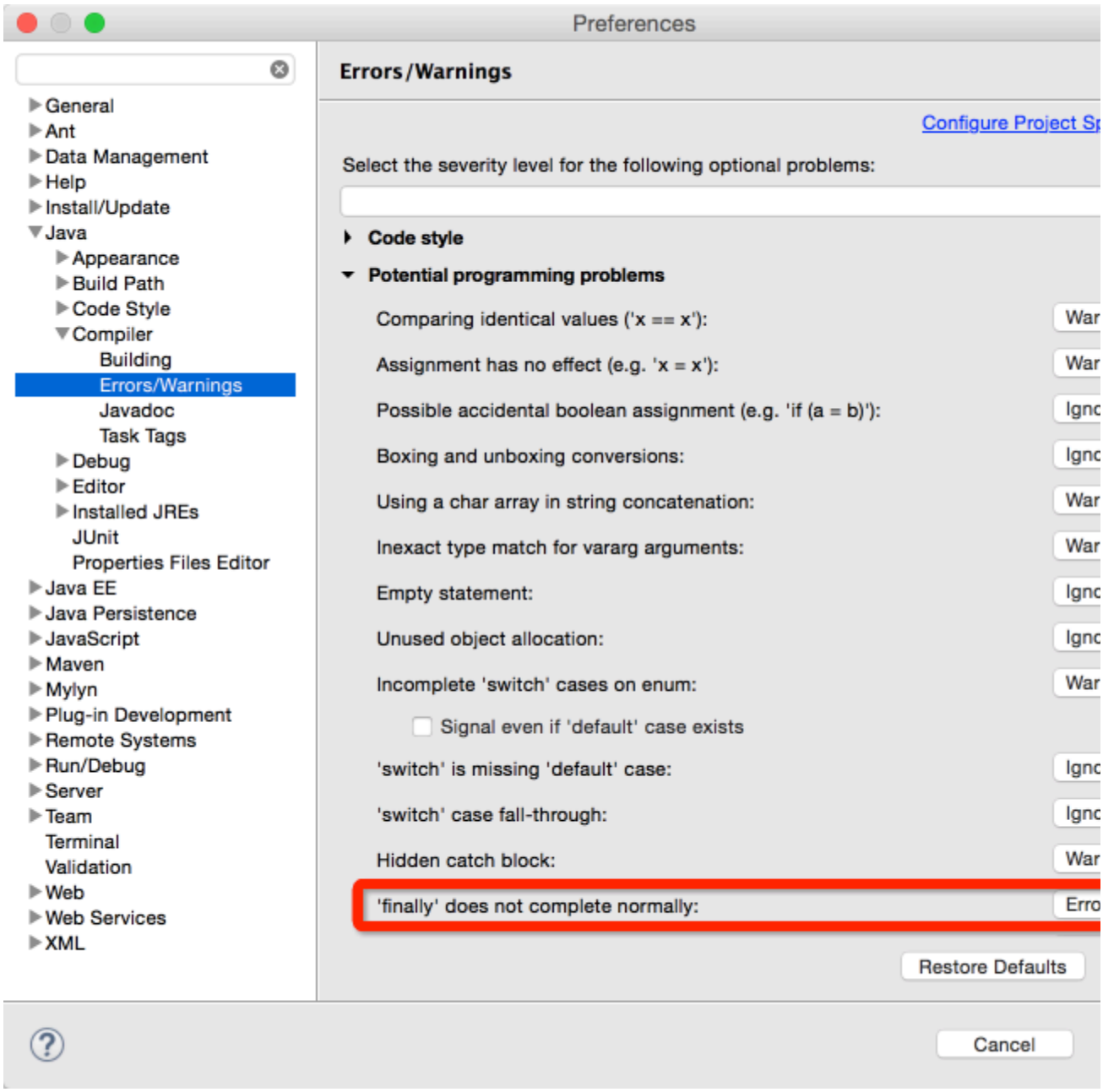
```
class StackOverflowErrorTest {


    public static void main(String[] args) {

        main(null);

    }

}
```

提示：用递归编写程序时一定要牢记两点：1. 递归公式；2. 收敛条件（什么时候就不再继续递归）。

46、try{}里有一个 return 语句，那么紧跟在这个 try 后的 finally{}里的代码会不会被执行，什么时候被执行，在 return 前还是后?

答：会执行，在方法返回调用者前执行。

注意：在 finally 中改变返回值的做法是不好的，因为如果存在 finally 代码块，try 中的 return 语句不会立马返回调用者，而是记录下返回值待 finally 代码块执行完毕之后再向调用者返回其值，然后如果在 finally 中修改了返回值，就会返回修改后的值。显然，在 finally 中返回或者修改返回值会对程序造成很大的困扰，C#中直接用编译错误的方式来阻止程序员干这种龌龊的事情，Java 中也可以通过提升编译器的语法检查级别来产生警告或错误，Eclipse 中可以在如图所示的地方进行设置，强烈建议将此项设置为编译错误。



47、Java 语言如何进行异常处理，关键字：throws、throw、try、catch、finally 分别如何使用？

答：Java 通过面向对象的方法进行异常处理，把各种不同的异常进行分类，并提供了良好的接口。在 Java 中，每个异常都是一个对象，它是 Throwable 类或其子类的实例。当一个方法出现异常后便抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并可以对其进行处理。Java 的异常处理是通过 5 个关键词来实现的：try、

catch、throw、throws 和 finally。一般情况下是用 try 来执行一段程序，如果系统会抛出（throw）一个异常对象，可以通过它的类型来捕获（catch）它，或通过总是执行代码块（finally）来处理；try 用来指定一块预防所有异常的程序；catch 子句紧跟在 try 块后面，用来指定你想要捕获的异常的类型；throw 语句用来明确地抛出一个异常；throws 用来声明一个方法可能抛出的各种异常（当然声明异常时允许无病呻吟）；finally 为确保一段代码不管发生什么异常状况都要被执行；try 语句可以嵌套，每当遇到一个 try 语句，异常的结构就会被放入异常栈中，直到所有的 try 语句都完成。如果下一级的 try 语句没有对某种异常进行处理，异常栈就会执行出栈操作，直到遇到有处理这种异常的 try 语句或者最终将异常抛给 JVM。

48、运行时异常与受检异常有何异同？

答：异常表示程序运行过程中可能出现的非正常状态，运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种常见运行错误，只要程序设计得没有问题通常就不会发生。受检异常跟程序运行的上下文环境有关，即使程序设计无误，仍然可能因使用的问题而引发。Java 编译器要求方法必须声明抛出可能发生的受检异常，但是并不要求必须声明抛出未被捕获的运行时异常。异常和继承一样，是面向对象程序设计中经常被滥用的东西，在 *Effective Java* 中对异常的使用给出了以下指导原则：

- 不要将异常处理用于正常的控制流（设计良好的 API 不应该强迫它的调用者为了正常的控制流而使用异常）
- 对可以恢复的情况使用受检异常，对编程错误使用运行时异常
- 避免不必要的使用受检异常（可以通过一些状态检测手段来避免异常的发生）
- 优先使用标准的异常
- 每个方法抛出的异常都要有文档

- 保持异常的原子性

- 不要在 catch 中忽略掉捕获到的异常

49、列出一些你常见的运行时异常？

答：

- ArithmeticException（算术异常）

- ClassCastException （类转换异常）

- IllegalArgumentException （非法参数异常）

- IndexOutOfBoundsException （下标越界异常）

- NullPointerException （空指针异常）

- SecurityException （安全异常）

50、阐述 final、finally、finalize 的区别。

答：

- final：修饰符（关键字）有三种用法：如果一个类被声明为 final，意味着它不能再派生出新的子类，即不能被继承，因此它和 abstract 是反义词。将变量声明为 final，可以保证它们在使用中不被改变，被声明为 final 的变量必须在声明时给定初值，而在以后的引用中只能读取不可修改。被声明为 final 的方法也同样只能使用，不能在子类中被重写。

- finally：通常放在 try…catch…的后面构造总是执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要 JVM 不关闭都能执行，可以将释放外部资源的代码写在 finally 块中。

- finalize：Object 类中定义的方法，Java 中允许使用 finalize()方法在垃圾收集器将对象

从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写 finalize()方法可以整理系统资源或者执行其他清理工作。

51、类 ExampleA 继承 Exception，类 ExampleB 继承 ExampleA。

有如下代码片断：

```
try {
    throw new ExampleB("b")
} catch（ExampleA e）{
    System.out.println("ExampleA");
} catch（Exception e）{
    System.out.println("Exception");
}
```

请问执行此段代码的输出是什么？

答：输出：ExampleA。（根据里氏代换原则[能使用父类型的地方一定能使用子类型]，抓取 ExampleA 类型异常的 catch 块能够抓住 try 块中抛出的 ExampleB 类型的异常）

面试题 - 说出下面代码的运行结果。（此题的出处是《Java 编程思想》一书）

```
class Annoyance extends Exception {}
class Sneeze extends Annoyance {}
```

```
class Human {


    public static void main(String[] args)
        throws Exception {
        try {
            try {
                throw new Sneeze();
            }
            catch ( Annoyance a ) {
                System.out.println("Caught Annoyance");
                throw a;
            }
        }
        catch ( Sneeze s ) {
            System.out.println("Caught Sneeze");
            return ;
        }
        finally {
            System.out.println("Hello World!");
        }
    }
}
```

- 1
- 2
- 3
- 4

```
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
```

52、List、Set、Map 是否继承自 Collection 接口？

答：List、Set 是，Map 不是。Map 是键值对映射容器，与 List 和 Set 有明显的区别，而 Set 存储的零散的元素且不允许有重复元素（数学中的集合也是如此），List 是线性结构的容器，适用于按数值索引访问元素的情形。

53、阐述 ArrayList、Vector、LinkedList 的存储性能和特性。

答：ArrayList 和 Vector 都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢，Vector 中的方法由于添加了 synchronized 修饰，因此 Vector 是线程安全的容器，但性能上较 ArrayList 差，因此已经是 Java 中的遗留容器。LinkedList 使用双向链表实现存储（将内存中零散的内存单元通过附加的引用关联起来，形成一个可以按序号索引的线性结构，这种链式存储方式与数组的连续存储方式相比，

内存的利用率更高），按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。Vector 属于遗留容器（Java 早期的版本中提供的容器，除此之外，Hashtable、Dictionary、BitSet、Stack、Properties 都是遗留容器），已经不推荐使用，但是由于 ArrayList 和 LinkedListed 都是非线程安全的，如果遇到多个线程操作同一个容器的场景，则可以通过工具类 Collections 中的 synchronizedList 方法将其转换成线程安全的容器后再使用（这是对装潢模式的应用，将已有对象传入另一个类的构造器中创建新的对象来增强实现）。

补充：遗留容器中的 Properties 类和 Stack 类在设计上有严重的问题，Properties 是一个键和值都是字符串的特殊的键值对映射，在设计上应该是关联一个 Hashtable 并将其两个泛型参数设置为 String 类型，但是 Java API 中的 Properties 直接继承了 Hashtable，这很明显是对继承的滥用。这里复用代码的方式应该是 Has-A 关系而不是 Is-A 关系，另一方面容器都属于工具类，继承工具类本身就是一个错误的做法，使用工具类最好的方式是 Has-A 关系（关联）或 Use-A 关系（依赖）。同理，Stack 类继承 Vector 也是不正确的。Sun 公司的工程师们也会犯这种低级错误，让人唏嘘不已。

54、Collection 和 Collections 的区别？

答：Collection 是一个接口，它是 Set、List 等容器的父接口；Collections 是个一个工具类，提供了一系列的静态方法来辅助容器操作，这些方法包括对容器的搜索、排序、线程安全化等等。

55、List、Map、Set 三个接口存取元素时，各有什么特点？

答：List 以特定索引来存取元素，可以有重复元素。Set 不能存放重复元素（用对象的 equals() 方法来区分元素是否重复）。Map 保存键值对（key-value pair）映射，映射关系可以是

一对一或多对一。Set 和 Map 容器都有基于哈希存储和排序树的两种实现版本，基于哈希存储的版本理论存取时间复杂度为 O(1)，而基于排序树版本的实现在插入或删除元素时会按照元素或元素的键（key）构成排序树从而达到排序和去重的效果。

56、TreeMap 和 TreeSet 在排序时如何比较元素？Collections 工具类中的 sort()方法如何比较元素？

答：TreeSet 要求存放的对象所属的类必须实现 Comparable 接口，该接口提供了比较元素的 compareTo()方法，当插入元素时会回调该方法比较元素的大小。TreeMap 要求存放的键值对映射的键必须实现 Comparable 接口从而根据键对元素进行排序。Collections 工具类的 sort 方法有两种重载的形式，第一种要求传入的待排序容器中存放的对象比较实现 Comparable 接口以实现元素的比较；第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是 Comparator 接口的子类型（需要重写 compare 方法实现元素的比较），相当于一个临时定义的排序规则，其实就是通过接口注入比较元素大小的算法，也是对回调模式的应用（Java 中对函数式编程的支持）。

例子 1：

```
public class Student implements Comparable<Student> {
    private String name;        // 姓名
    private int age;            // 年龄

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
```

```
    }

    @Override
    public String toString() {
        return "Student [name=" + name + ", age=" + age + "]";
    }

    @Override
    public int compareTo(Student o) {
        return this.age - o.age; // 比较年龄(年龄的升序)
    }

}
```

```
• 21
```

```
import java.util.Set;

import java.util.TreeSet;


class Test01 {


    public static void main(String[] args) {

        Set<Student> set = new TreeSet<>();     // Java 7 的钻石语法(构造器后面的尖括号中不需要写类型)

        set.add(new Student("Hao LUO", 33));

        set.add(new Student("XJ WANG", 32));

        set.add(new Student("Bruce LEE", 60));

        set.add(new Student("Bob YANG", 22));


        for(Student stu : set) {

            System.out.println(stu);

        }
//      输出结果:
//      Student [name=Bob YANG, age=22]
//      Student [name=XJ WANG, age=32]
//      Student [name=Hao LUO, age=33]
//      Student [name=Bruce LEE, age=60]

    }

}
```

```
• 1
```

```
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

例子 2：

```
public class Student {

    private String name;    // 姓名

    private int age;        // 年龄


    public Student(String name, int age) {

        this.name = name;

        this.age = age;

    }


    /**

     * 获取学生姓名
```

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

    /**
     * 获取学生年龄
     */
    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Student [name=" + name + ", age=" + age + "]";
    }

}
```

```
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
```

```
import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;


class Test02 {


    public static void main(String[] args) {

        List<Student> list = new ArrayList<>();      // Java 7 的钻石语法(构造器后面的尖括号中不需要写类型)

        list.add(new Student("Hao LUO", 33));

        list.add(new Student("XJ WANG", 32));

        list.add(new Student("Bruce LEE", 60));

        list.add(new Student("Bob YANG", 22));
```

```
        // 通过 sort 方法的第二个参数传入一个 Comparator 接口对象
        // 相当于是传入一个比较对象大小的算法到 sort 方法中
        // 由于 Java 中没有函数指针、仿函数、委托这样的概念
        // 因此要将一个算法传入一个方法中唯一的选择就是通过接口回调
        Collections.sort(list, new Comparator<Student> () {

            @Override
            public int compare(Student o1, Student o2) {
                return o1.getName().compareTo(o2.getName());    // 比较学生姓名
            }
        });

        for(Student stu : list) {
            System.out.println(stu);
        }
//      输出结果:
//      Student [name=Bob YANG, age=22]
//      Student [name=Bruce LEE, age=60]
//      Student [name=Hao LUO, age=33]
//      Student [name=XJ WANG, age=32]
    }
}
```

```
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

57、Thread 类的 sleep()方法和对象的 wait()方法都可以让线程暂停执行，它们有什么区别?

答：sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠

时间结束后会自动恢复（线程回到就绪状态，请参考第 66 题中的线程状态转换图）。wait()是 Object 类的方法，调用对象的 wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的 notify()方法（或 notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

补充：可能不少人对什么是进程，什么是线程还比较模糊，对于为什么需要多线程编程也不是特别理解。简单的说：进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，是操作系统进行资源分配和调度的一个独立单位；线程是进程的一个实体，是 CPU 调度和分派的基本单位，是比进程更小的能独立运行的基本单位。线程的划分尺度小于进程，这使得多线程程序的并发性高；进程在执行时通常拥有独立的内存单元，而线程之间可以共享内存。使用多线程的编程通常能够带来更好的性能和用户体验，但是多线程的程序对于其他程序是不友好的，因为它可能占用了更多的 CPU 资源。当然，也不是线程越多，程序的性能就越好，因为线程之间的调度和切换也会浪费 CPU 时间。时下很时髦的 Node.js 就采用了单线程异步 I/O 的工作模式。

58、线程的 sleep()方法和 yield()方法有什么区别？

答：

① sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；

② 线程执行 sleep()方法后转入阻塞（blocked）状态，而执行 yield()方法后转入就绪（ready）状态；

③ sleep()方法声明抛出 InterruptedException，而 yield()方法没有声明任何异常；

④ sleep()方法比 yield()方法（跟操作系统 CPU 调度相关）具有更好的可移植性。

59、当一个线程进入一个对象的 synchronized 方法 A 之后，其它线程是否可进入此对象的 synchronized 方法 B？

答：不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的 synchronized 修饰符要求执行方法时要获得对象的锁，如果已经进入 A 方法说明对象锁已经被取走，那么试图进入 B 方法的线程就只能在等锁池（**注意不是等待池哦**）中等待对象的锁。

60、请说出与线程同步以及线程调度相关的方法。

答：

- wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；

- sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理 InterruptedException 异常；

- notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由 JVM 确定唤醒哪个线程，而且与优先级无关；

- notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

提示：关于 Java 多线程和并发编程的问题，建议大家看我的另一篇文章《关于 Java 并发编程的总结和思考》。

补充：Java 5 通过 Lock 接口提供了显式的锁机制（explicit lock），增强了灵活性以及对线程的协调。Lock 接口中定义了加锁（lock()）和解锁（unlock()）的方法，同时还提供了 newCondition()方法来产生用于线程之间通信的 Condition 对象；此外，Java 5 还提供了信号量机制（semaphore），信号量可以用来限制对某个共享资源进行访问的线程的数量。

在对资源进行访问之前，线程必须得到信号量的许可（调用 Semaphore 对象的 acquire()方法）；在完成对资源的访问后，线程必须向信号量归还许可（调用 Semaphore 对象的 release()方法）。

下面的例子演示了 100 个线程同时向一个银行账户中存入 1 元钱，在没有使用同步机制和使用同步机制情况下的执行情况。

- 银行账户类：

```
/**
 * 银行账户
 * @author 骆昊
 *
 */
public class Account {
    private double balance;     // 账户余额

    /**
     * 存款
     * @param money 存入金额
     */
    public void deposit(double money) {
        double newBalance = balance + money;
        try {
            Thread.sleep(10);   // 模拟此业务需要一段处理时间
        }
```

```
            catch(InterruptedException ex) {
                ex.printStackTrace();
            }
            balance = newBalance;
        }

        /**
         * 获得账户余额
         */
        public double getBalance() {
            return balance;
        }
    }
```

```
21
22
23
24
25
26
27
28
29
30
```

- 存钱线程类：

```
/**
 * 存钱线程
 * @author 骆昊
 *
 */
public class AddMoneyThread implements Runnable {
    private Account account;    // 存入账户
    private double money;       // 存入金额

    public AddMoneyThread(Account account, double money) {
        this.account = account;
        this.money = money;
    }

    @Override
    public void run() {
        account.deposit(money);
```

```
    }


}
```

- 测试类：

```
import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;


public class Test01 {


    public static void main(String[] args) {

        Account account = new Account();

        ExecutorService service = Executors.newFixedThreadPool(100);
```

```
        for(int i = 1; i <= 100; i++) {
            service.execute(new AddMoneyThread(account, 1));
        }

        service.shutdown();

        while(!service.isTerminated()) {}

        System.out.println("账户余额: " + account.getBalance());
    }
}
```

在没有同步的情况下 ,执行结果通常是显示账户余额在 10 元以下 ,出现这种状况的原因是，当一个线程 A 试图存入 1 元的时候，另外一个线程 B 也能够进入存款的方法中，线程 B 读取到的账户余额仍然是线程 A 存入 1 元钱之前的账户余额，因此也是在原来的余额 0 上面做了加 1 元的操作，同理线程 C 也会做类似的事情，所以最后 100 个线程执行结束时，本来期望账户余额为 100 元，但实际得到的通常在 10 元以下（很可能是 1 元哦）。解决这个问题的办法就是同步，当一个线程对银行账户存钱时，需要将此账户锁定，待其操作完成后才允许其他的线程进行操作，代码有如下几种调整方案：

- 在银行账户的存款（deposit）方法上同步（synchronized）关键字

```
/**
 * 银行账户
 * @author 骆昊
 *
 */
public class Account {
    private double balance;     // 账户余额

    /**
     * 存款
     * @param money 存入金额
     */
    public synchronized void deposit(double money) {
        double newBalance = balance + money;
        try {
```

```
            Thread.sleep(10);   // 模拟此业务需要一段处理时间
        }
        catch(InterruptedException ex) {
            ex.printStackTrace();
        }
        balance = newBalance;
    }

    /**
     * 获得账户余额
     */
    public double getBalance() {
        return balance;
    }
}
```

```
18
19
20
21
22
23
24
25
26
27
28
29
30
```

- 在线程调用存款方法时对银行账户进行同步

```
/**
 * 存钱线程
 * @author 骆昊
 *
 */
public class AddMoneyThread implements Runnable {
    private Account account;    // 存入账户
    private double money;       // 存入金额

    public AddMoneyThread(Account account, double money) {
        this.account = account;
        this.money = money;
    }

    @Override
```

```
    public void run() {
        synchronized (account) {
            account.deposit(money);
        }
    }

}
```

- 通过 Java 5 显示的锁机制，为每个银行账户创建一个锁对象，在存款操作进行加锁和解锁的操作

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
```

```java
/**
 * 银行账户
 *
 * @author 骆昊
 *
 */
public class Account {
    private Lock accountLock = new ReentrantLock();
    private double balance; // 账户余额

    /**
     * 存款
     *
     * @param money
     *            存入金额
     */
    public void deposit(double money) {
        accountLock.lock();
        try {
            double newBalance = balance + money;
            try {
                Thread.sleep(10); // 模拟此业务需要一段处理时间
            }
```

```
            catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            balance = newBalance;
        }
        finally {
            accountLock.unlock();
        }
    }

    /**
     * 获得账户余额
     */
    public double getBalance() {
        return balance;
    }
}
```

```
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
```

按照上述三种方式对代码进行修改后，重写执行测试代码 Test01，将看到最终的账户余额为 100 元。当然也可以使用 Semaphore 或 CountdownLatch 来实现同步。

61、编写多线程程序有几种实现方式？

答：Java 5 以前实现多线程有两种实现方法：一种是继承 Thread 类；另一种是实现 Runnable 接口。两种方式都要通过重写 run()方法来定义线程的行为，推荐使用后者，因

为 Java 中的继承是单继承，一个类有一个父类，如果继承了 Thread 类就无法再继承其他类了，显然使用 Runnable 接口更为灵活。

补充：Java 5 以后创建线程还有第三种方式：实现 Callable 接口，该接口中的 call 方法可以在线程执行结束时产生一个返回值，代码如下所示：

```
import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;



class MyTask implements Callable<Integer> {

    private int upperBounds;


    public MyTask(int upperBounds) {

        this.upperBounds = upperBounds;

    }


    @Override

    public Integer call() throws Exception {

        int sum = 0;

        for(int i = 1; i <= upperBounds; i++) {
```

```
            sum += i;
        }
        return sum;
    }

}

class Test {

    public static void main(String[] args) throws Exception {
        List<Future<Integer>> list = new ArrayList<>();
        ExecutorService service = Executors.newFixedThreadPool(10);
        for(int i = 0; i < 10; i++) {
            list.add(service.submit(new MyTask((int) (Math.random() * 1
00))));
        }

        int sum = 0;
        for(Future<Integer> future : list) {
            // while(!future.isDone()) ;
            sum += future.get();
        }

        System.out.println(sum);
    }
```

```
}
```

```
43
44
```

62、synchronized 关键字的用法？

答：synchronized 关键字可以将对象或者方法标记为同步，以实现对对象和方法的互斥访问，可以用 synchronized(对象) { … }定义同步代码块，或者在声明方法时将 synchronized 作为方法的修饰符。在第 60 题的例子中已经展示了 synchronized 关键字的用法。

63、举例说明同步和异步。

答：如果系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。事实上，所谓的同步就是指阻塞式操作，而异步就是非阻塞式操作。

64、启动一个线程是调用 run()还是 start()方法？

答：启动一个线程是调用 start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由 JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

65、什么是线程池（thread pool）？

答：在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在 Java 中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的

次数，特别是一些很耗资源的对象创建和销毁，这就是”池化资源”技术产生的原因。线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

Java 5+中的 Executor 接口定义一个执行线程的工具。它的子类型即线程池接口是 ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类 Executors 面提供了一些静态工厂方法，生成一些常用的线程池，如下所示：

- newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

- newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

- newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60 秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。

- newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

- newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

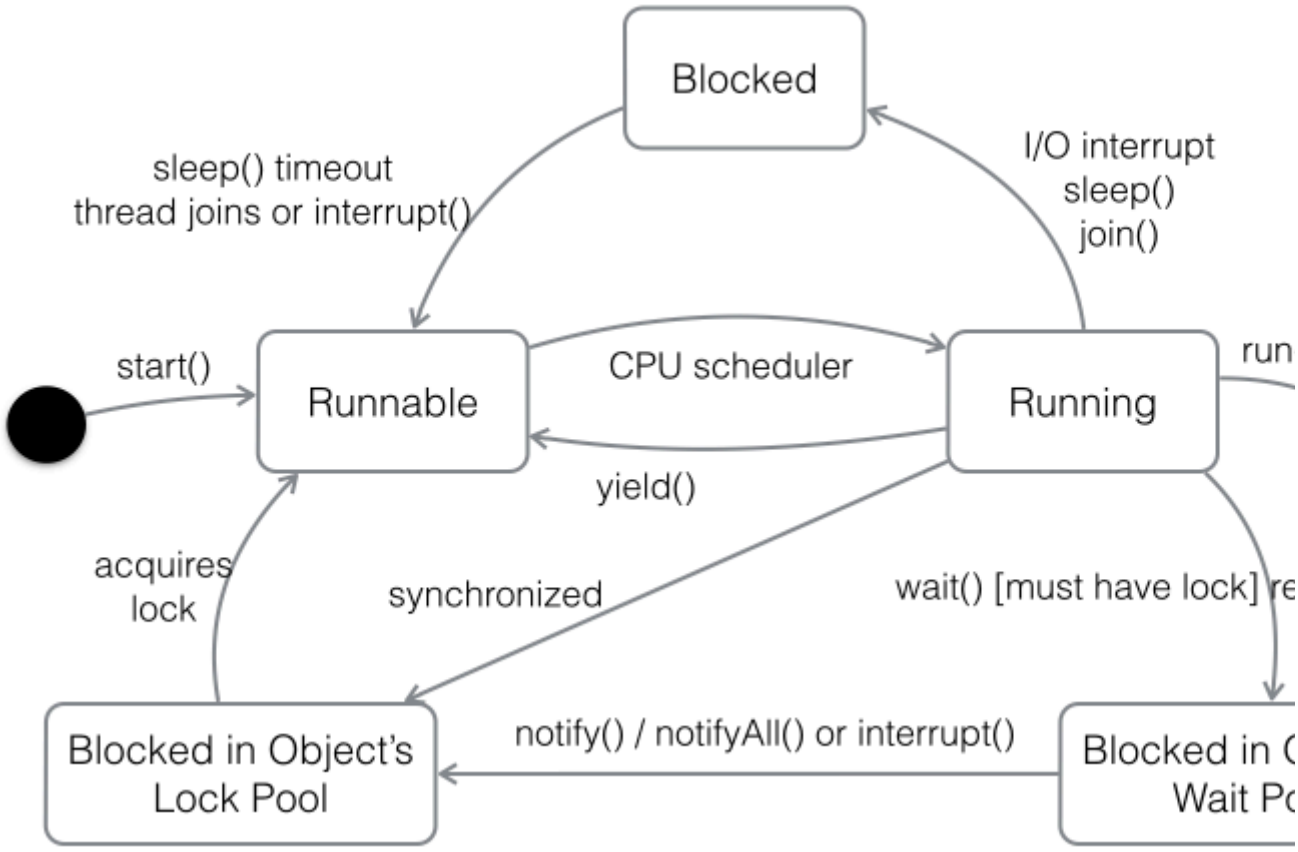
第 60 题的例子中演示了通过 Executors 工具类创建线程池并使用线程池执行线程的代码。

如果希望在服务器上使用线程池 ,强烈建议使用 newFixedThreadPool 方法来创建线程池，

这样能获得更好的性能。

66、线程的基本状态以及状态之间的关系？

答：



说明：其中 Running 表示运行状态，Runnable 表示就绪状态（万事俱备，只欠 CPU），

Blocked 表示阻塞状态，阻塞状态又有多种情况，可能是因为调用 wait()方法进入等待池，

也可能是执行同步方法或同步代码块进入等锁池，或者是调用了 sleep()方法或 join()方法

等待休眠或其他线程结束，或是因为发生了 I/O 中断。

67、简述 synchronized 和 java.util.concurrent.locks.Lock 的异同？

答：Lock 是 Java 5 以后引入的新的 API，和关键字 synchronized 相比主要相同点：Lock

能完成 synchronized 所实现的所有功能；主要不同点：Lock 有比 synchronized 更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁。synchronized 会自动释放锁，而 Lock 一定要求程序员手工释放，并且最好在 finally 块中释放（这是释放外部资源的最好的地方）。

68、Java 中如何实现序列化，有什么意义？

答：序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。

要实现序列化，需要让一个类实现 Serializable 接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过 writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过 readObject 方法从流中读取对象。序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆（可以参考第 29 题）。

69、Java 中有几种类型的流？

答：字节流和字符流。字节流继承于 InputStream、OutputStream，字符流继承于 Reader、Writer。在 java.io 包中还有许多其他的流，主要是为了提高性能和使用方便。关于 Java 的 I/O 需要注意的有两点：一是两种对称性(输入和输出的对称性，字节和字符的对称性)；二是两种设计模式（适配器模式和装潢模式）。另外 Java 中的流不同于 C#的是它只有一个维度一个方向。

面试题 - 编程实现文件拷贝。（这个题目在笔试的时候经常出现，下面的代码给出了两种实现方案）

```
import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;


public final class MyUtil {


    private MyUtil() {

        throw new AssertionError();

    }


    public static void fileCopy(String source, String target) throws IO
Exception {

        try (InputStream in = new FileInputStream(source)) {

            try (OutputStream out = new FileOutputStream(target)) {

                byte[] buffer = new byte[4096];

                int bytesToRead;

                while((bytesToRead = in.read(buffer)) != -1) {

                    out.write(buffer, 0, bytesToRead);

                }

            }

        }
```

```
    }


    public static void fileCopyNIO(String source, String target) throw
s IOException {
        try (FileInputStream in = new FileInputStream(source)) {
            try (FileOutputStream out = new FileOutputStream(target)) {
                FileChannel inChannel = in.getChannel();
                FileChannel outChannel = out.getChannel();
                ByteBuffer buffer = ByteBuffer.allocate(4096);
                while(inChannel.read(buffer) != -1) {
                    buffer.flip();
                    outChannel.write(buffer);
                    buffer.clear();
                }
            }
        }
    }
}
```

```
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
```

注意：上面用到 Java 7 的 TWR，使用 TWR 后可以不用在 finally 中释放外部资源 ，从而让代码更加优雅。

70、写一个方法，输入一个文件名和一个字符串，统计这个字符串在这个文件中出现的次数。

答：代码如下：

```
import java.io.BufferedReader;

import java.io.FileReader;
```

```
public final class MyUtil {

    // 工具类中的方法都是静态方式访问的因此将构造器私有不允许创建对象(绝对好习惯)
    private MyUtil() {
        throw new AssertionError();
    }

    /**
     * 统计给定文件中给定字符串的出现次数
     *
     * @param filename  文件名
     * @param word 字符串
     * @return 字符串在文件中出现的次数
     */
    public static int countWordInFile(String filename, String word) {
        int counter = 0;
        try (FileReader fr = new FileReader(filename)) {
            try (BufferedReader br = new BufferedReader(fr)) {
                String line = null;
                while ((line = br.readLine()) != null) {
                    int index = -1;
                    while (line.length() >= word.length() && (index = lin
e.indexOf(word)) >= 0) {
                        counter++;
```

```
                    line = line.substring(index + word.length());
                }
            }
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return counter;
  }

}
```

```
25
26
27
28
29
30
31
32
33
34
35
36
37
```

71、如何用 Java 代码列出一个目录下所有的文件？

答：

如果只要求列出当前文件夹下的文件，代码如下所示：

```
import java.io.File;


class Test12 {


    public static void main(String[] args) {
        File f = new File("/Users/Hao/Downloads");
        for(File temp : f.listFiles()) {
            if(temp.isFile()) {
                System.out.println(temp.getName());
            }
        }
    }
}
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

如果需要对文件夹继续展开，代码如下所示：

```
import java.io.File;


class Test12 {


    public static void main(String[] args) {
        showDirectory(new File("/Users/Hao/Downloads"));
    }


    public static void showDirectory(File f) {
        _walkDirectory(f, 0);
    }


    private static void _walkDirectory(File f, int level) {
        if(f.isDirectory()) {
            for(File temp : f.listFiles()) {
```

```
                _walkDirectory(temp, level + 1);
            }
        }
        else {
            for(int i = 0; i < level - 1; i++) {
                System.out.print("\t");
            }
            System.out.println(f.getName());
        }
    }
}
```

```
25
26
```

在 Java 7 中可以使用 NIO.2 的 API 来做同样的事情，代码如下所示：

```
class ShowFileTest {


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

        Path initPath = Paths.get("/Users/Hao/Downloads");

        Files.walkFileTree(initPath, new SimpleFileVisitor<Path>() {


            @Override

            public FileVisitResult visitFile(Path file, BasicFileAttrib
utes attrs)

                    throws IOException {

                System.out.println(file.getFileName().toString());

                return FileVisitResult.CONTINUE;

            }


        });

    }

}
```

```
1
2
3
4
5
6
7
8
```

```
9
10
11
12
13
14
15
16
```

72、用 Java 的套接字编程实现一个多线程的回显（echo）服务器。

答：

```
import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.ServerSocket;

import java.net.Socket;


public class EchoServer {


    private static final int ECHO_SERVER_PORT = 6789;


    public static void main(String[] args) {

        try(ServerSocket server = new ServerSocket(ECHO_SERVER_PORT)) {

            System.out.println("服务器已经启动...");

            while(true) {

                Socket client = server.accept();

                new Thread(new ClientHandler(client)).start();
```

```
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static class ClientHandler implements Runnable {
        private Socket client;

        public ClientHandler(Socket client) {
            this.client = client;
        }

        @Override
        public void run() {
            try(BufferedReader br = new BufferedReader(new InputStreamR
eader(client.getInputStream()));
                    PrintWriter pw = new PrintWriter(client.getOutputStre
am())) {
                String msg = br.readLine();
                System.out.println("收到" + client.getInetAddress() + "发
送的: " + msg);
                pw.println(msg);
                pw.flush();
            } catch(Exception ex) {
                ex.printStackTrace();
```

```
            } finally {
                try {
                    client.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

}
```

```
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
```

注意：上面的代码使用了 Java 7 的 TWR 语法，由于很多外部资源类都间接的实现了 AutoCloseable 接口（单方法回调接口），因此可以利用 TWR 语法在 try 结束的时候通过回调的方式自动调用外部资源类的 close()方法，避免书写冗长的 finally 代码块。此外，上面的代码用一个静态内部类实现线程的功能，使用多线程可以避免一个用户 I/O 操作所产生的中断影响其他用户对服务器的访问 ,简单的说就是一个用户的输入操作不会造成其他用户的阻塞。当然，上面的代码使用线程池可以获得更好的性能，因为频繁的创建和销毁线程所造成的开销也是不可忽视的。

下面是一段回显客户端测试代码：

```
import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.Socket;

import java.util.Scanner;


public class EchoClient {


    public static void main(String[] args) throws Exception {

        Socket client = new Socket("localhost", 6789);

        Scanner sc = new Scanner(System.in);

        System.out.print("请输入内容: ");

        String msg = sc.nextLine();

        sc.close();

        PrintWriter pw = new PrintWriter(client.getOutputStream());

        pw.println(msg);

        pw.flush();

        BufferedReader br = new BufferedReader(new InputStreamReader(cl
ient.getInputStream()));

        System.out.println(br.readLine());

        client.close();

    }

}
```

- 1
- 2
- 3

```
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

如果希望用 NIO 的多路复用套接字实现服务器，代码如下所示。NIO 的操作虽然带来了更好的性能，但是有些操作是比较底层的，对于初学者来说还是有些难于理解。

```
import java.io.IOException;

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.CharBuffer;

import java.nio.channels.SelectionKey;

import java.nio.channels.Selector;

import java.nio.channels.ServerSocketChannel;

import java.nio.channels.SocketChannel;

import java.util.Iterator;


public class EchoServerNIO {
```

```
    private static final int ECHO_SERVER_PORT = 6789;
    private static final int ECHO_SERVER_TIMEOUT = 5000;
    private static final int BUFFER_SIZE = 1024;

    private static ServerSocketChannel serverChannel = null;
    private static Selector selector = null;    // 多路复用选择器
    private static ByteBuffer buffer = null;    // 缓冲区

    public static void main(String[] args) {
        init();
        listen();
    }

    private static void init() {
        try {
            serverChannel = ServerSocketChannel.open();
            buffer = ByteBuffer.allocate(BUFFER_SIZE);
            serverChannel.socket().bind(new InetSocketAddress(ECHO_SERV
ER_PORT));
            serverChannel.configureBlocking(false);
            selector = Selector.open();
            serverChannel.register(selector, SelectionKey.OP_ACCEPT);
        } catch (Exception e) {
            throw new RuntimeException(e);
```

```
        }
    }

    private static void listen() {
        while (true) {
            try {
                if (selector.select(ECHO_SERVER_TIMEOUT) != 0) {
                    Iterator<SelectionKey> it = selector.selectedKeys().
iterator();
                    while (it.hasNext()) {
                        SelectionKey key = it.next();
                        it.remove();
                        handleKey(key);
                    }
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    private static void handleKey(SelectionKey key) throws IOException
 {
        SocketChannel channel = null;
```

```
        try {
            if (key.isAcceptable()) {
                ServerSocketChannel serverChannel = (ServerSocketChannel)
 key.channel();
                channel = serverChannel.accept();
                channel.configureBlocking(false);
                channel.register(selector, SelectionKey.OP_READ);
            } else if (key.isReadable()) {
                channel = (SocketChannel) key.channel();
                buffer.clear();
                if (channel.read(buffer) > 0) {
                    buffer.flip();
                    CharBuffer charBuffer = CharsetHelper.decode(buffe
r);
                    String msg = charBuffer.toString();
                    System.out.println("收到" + channel.getRemoteAddress()
 + "的消息：" + msg);
                    channel.write(CharsetHelper.encode(CharBuffer.wrap(m
sg)));
                } else {
                    channel.close();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
            if (channel != null) {
```

```
                channel.close();

            }

        }

    }


}
```

```
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
```

```
78
79
80
81
82
83
84
85
86
```

```
import java.nio.ByteBuffer;

import java.nio.CharBuffer;

import java.nio.charset.CharacterCodingException;

import java.nio.charset.Charset;

import java.nio.charset.CharsetDecoder;

import java.nio.charset.CharsetEncoder;



public final class CharsetHelper {

    private static final String UTF_8 = "UTF-8";

    private static CharsetEncoder encoder = Charset.forName(UTF_8).new
Encoder();

    private static CharsetDecoder decoder = Charset.forName(UTF_8).new
Decoder();



    private CharsetHelper() {

    }



    public static ByteBuffer encode(CharBuffer in) throws CharacterCod
ingException{

        return encoder.encode(in);
```

```
    }


    public static CharBuffer decode(ByteBuffer in) throws CharacterCod
ingException{
        return decoder.decode(in);
    }
}
```

73、XML 文档定义有几种形式？它们之间有何本质区别？解析 XML 文档有哪几种方式？

答：XML 文档定义分为 DTD 和 Schema 两种形式，二者都是对 XML 语法的约束，其本质区别在于 Schema 本身也是一个 XML 文件，可以被 XML 解析器解析，而且可以为 XML

承载的数据定义类型，约束能力较之 DTD 更强大。对 XML 的解析主要有 DOM（文档对象模型，Document Object Model）、SAX（Simple API for XML）和 StAX（Java 6 中引入的新的解析 XML 的方式，Streaming API for XML），其中 DOM 处理大型文件时其性能下降的非常厉害，这个问题是由 DOM 树结构占用的内存较多造成的，而且 DOM 解析方式必须在解析文件之前把整个文档装入内存，适合对 XML 的随机访问（典型的用空间换取时间的策略）；SAX 是事件驱动型的 XML 解析方式，它顺序读取 XML 文件，不需要一次全部装载整个文件。当遇到像文件开头，文档结束，或者标签开头与标签结束时，它会触发一个事件，用户通过事件回调代码来处理 XML 文件，适合对 XML 的顺序访问；顾名思义，StAX 把重点放在流上，实际上 StAX 与其他解析方式的本质区别就在于应用程序能够把 XML 作为一个事件流来处理。将 XML 作为一组事件来处理的想法并不新颖（SAX 就是这样做的），但不同之处在于 StAX 允许应用程序代码把这些事件逐个拉出来，而不用提供在解析器方便时从解析器中接收事件的处理程序。

74、你在项目中哪些地方用到了 XML？

答：XML 的主要作用有两个方面：数据交换和信息配置。在做数据交换时，XML 将数据用标签组装成起来，然后压缩打包加密后通过网络传送给接收者，接收解密与解压缩后再从 XML 文件中还原相关信息进行处理，XML 曾经是异构系统间交换数据的事实标准，但此项功能几乎已经被 JSON（JavaScript Object Notation）取而代之。当然，目前很多软件仍然使用 XML 来存储配置信息，我们在很多项目中通常也会将作为配置信息的硬代码写在 XML 文件中，Java 的很多框架也是这么做的，而且这些框架都选择了 dom4j 作为处理 XML 的工具，因为 Sun 公司的官方 API 实在不怎么好用。

补充：现在有很多时髦的软件（如 Sublime）已经开始将配置文件书写成 JSON 格式，我们已经强烈的感受到 XML 的另一项功能也将逐渐被业界抛弃。

75、阐述 JDBC 操作数据库的步骤。

答：下面的代码以连接本机的 Oracle 数据库为例，演示 JDBC 操作数据库的步骤。

- 加载驱动。

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- 创建连接。

```
Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger");
```

- 创建语句。

```
PreparedStatement ps = con.prepareStatement("select * from emp where sal between ? and ?");
ps.setInt(1, 1000);
ps.setInt(2, 3000);
```

- 执行语句。

```
ResultSet rs = ps.executeQuery();
```

- 处理结果。

```
while(rs.next()) {
    System.out.println(rs.getInt("empno") + " - " + rs.getString("ename"));
}
```

- 关闭资源。

```
finally {
    if(con != null) {
        try {
            con.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

提示：关闭外部资源的顺序应该和打开的顺序相反，也就是说先关闭 ResultSet、再关闭 Statement、在关闭 Connection。上面的代码只关闭了 Connection（连接），虽然通常情况下在关闭连接时，连接上创建的语句和打开的游标也会关闭，但不能保证总是如此，因此应该按照刚才说的顺序分别关闭。此外，第一步加载驱动在 JDBC 4.0 中是可以省略的（自动从类路径中加载驱动），但是我们建议保留。

76、Statement 和 PreparedStatement 有什么区别？哪个性能更好？

答：与 Statement 相比，①PreparedStatement 接口代表预编译的语句，它主要的优势在于可以减少 SQL 的编译错误并增加 SQL 的安全性（减少 SQL 注射攻击的可能性）；②PreparedStatement 中的 SQL 语句是可以带参数的，避免了用字符串连接拼接 SQL 语句的麻烦和不安全；③当批量处理 SQL 或频繁执行相同的查询时，PreparedStatement 有明显的性能上的优势，由于数据库可以将编译优化后的 SQL 语句缓存起来，下次执行相同结构的语句时就会很快（不用再次编译和生成执行计划）。

补充：为了提供对存储过程的调用，JDBC API 中还提供了 CallableStatement 接口。存储过程（Stored Procedure）是数据库中一组为了完成特定功能的 SQL 语句的集合，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。虽然调用存储过程会在网络开销、安全性、性能上获得很多好处，但是存在如果底层数据库发生迁移时就会有很多麻烦，因为每种数据库的存储过程在书写上存在不少的差别。

77、使用 JDBC 操作数据库时，如何提升读取数据的性能？如何提升更新数据的性能？

答：要提升读取数据的性能，可以指定通过结果集（ResultSet）对象的 setFetchSize()方

法指定每次抓取的记录数（典型的空间换时间策略）；要提升更新数据的性能可以使用 PreparedStatement 语句构建批处理，将若干 SQL 语句置于一个批处理中执行。

78、在进行数据库编程时，连接池有什么作用？

答：由于创建连接和释放连接都有很大的开销（尤其是数据库服务器不在本地时，每次建立连接都需要进行 TCP 的三次握手，释放连接需要进行 TCP 四次握手，造成的开销是不可忽视的），为了提升系统访问数据库的性能，可以事先创建若干连接置于连接池中，需要时直接从连接池获取，使用结束时归还连接池而不必关闭连接，从而避免频繁创建和释放连接所造成的开销，这是典型的用空间换取时间的策略（浪费了空间存储连接，但节省了创建和释放连接的时间）。池化技术在 Java 开发中是很常见的，在使用线程时创建线程池的道理与此相同。基于 Java 的开源数据库连接池主要有：C3P0、Proxool、DBCP、BoneCP、Druid 等。

补充：在计算机系统中时间和空间是不可调和的矛盾，理解这一点对设计满足性能要求的算法是至关重要的。大型网站性能优化的一个关键就是使用缓存，而缓存跟上面讲的连接池道理非常类似，也是使用空间换时间的策略。可以将热点数据置于缓存中，当用户查询这些数据时可以直接从缓存中得到，这无论如何也快过去数据库中查询。当然，缓存的置换策略等也会对系统性能产生重要影响，对于这个问题的讨论已经超出了这里要阐述的范围。

79、什么是 DAO 模式？

答：DAO（Data Access Object）顾名思义是一个为数据库或其他持久化机制提供了抽象接口的对象，在不暴露底层持久化方案实现细节的前提下提供了各种数据访问操作。在实际的开发中，应该将所有对数据源的访问操作进行抽象化后封装在一个公共 API 中。用程序设计语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。

在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口，在逻辑上该类对应一个特定的数据存储。DAO 模式实际上包含了两个模式，一是 Data Accessor（数据访问器），二是 Data Object（数据对象），前者要解决如何访问数据的问题，而后者要解决的是如何用对象封装数据。

80、事务的 ACID 是指什么？

答：

- 原子性(Atomic)：事务中各项操作，要么全做要么全不做，任何一项操作的失败都会导致整个事务的失败；
- 一致性(Consistent)：事务结束后系统状态是一致的；
- 隔离性(Isolated)：并发执行的事务彼此无法看到对方的中间状态；
- 持久性(Durable)：事务完成后所做的改动都会被持久化，即使发生灾难性的失败。通过日志和同步备份可以在故障发生后重建数据。

补充：关于事务，在面试中被问到的概率是很高的，可以问的问题也是很多的。首先需要知道的是，只有存在并发数据访问时才需要事务。当多个事务访问同一数据时，可能会存在 5 类问题，包括 3 类数据读取问题（脏读、不可重复读和幻读）和 2 类数据更新问题（第 1 类丢失更新和第 2 类丢失更新）。

脏读（Dirty Read）：A 事务读取 B 事务尚未提交的数据并在此基础上操作，而 B 事务执行回滚，那么 A 读取到的数据就是脏数据。

时间	转账事务 A	取款事务 B
T1		开始事务
T2	开始事务	

时间	转账事务 A	取款事务 B
T3		查询账户余额为 1000 元
T4		取出 500 元余额修改为 500 元
T5	查询账户余额为 500 元（脏读）	
T6		撤销事务余额恢复为 1000 元
T7	汇入 100 元把余额修改为 600 元	
T8	提交事务	

不可重复读（Unrepeatable Read）：事务 A 重新读取前面读取过的数据，发现该数据已经被另一个已提交的事务 B 修改过了。

时间	转账事务 A	取款事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元
T4	查询账户余额为 1000 元	
T5		取出 100 元修改余额为 900
T6		提交事务
T7	查询账户余额为 900 元（不可重复读）	

幻读（Phantom Read）：事务 A 重新执行一个查询，返回一系列符合查询条件的行，发现其中插入了被事务 B 提交的行。

时间	统计金额事务 A	转账事务 B
T1		开始事务
T2	开始事务	
T3	统计总存款为 10000 元	
T4		新增一个存款账户存入 100 元
T5		提交事务
T6	再次统计总存款为 10100 元（幻读）	

第 1 类丢失更新：事务 A 撤销时，把已经提交的事务 B 的更新数据覆盖了。

时间	取款事务 A	转账事务 B
T1	开始事务	
T2		开始事务
T3	查询账户余额为 1000 元	
T4		查询账户余额为 1000 元
T5		汇入 100 元修改余额为 1100 元
T6		提交事务
T7	取出 100 元将余额修改为 900 元	
T8	撤销事务	
T9	余额恢复为 1000 元（丢失更新）	

第 2 类丢失更新：事务 A 覆盖事务 B 已经提交的数据，造成事务 B 所做的操作丢失。

时间	转账事务 A	取款事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元
T4	查询账户余额为 1000 元	
T5		取出 100 元将余额修改为 900
T6		提交事务
T7	汇入 100 元将余额修改为 1100 元	
T8	提交事务	
T9	查询账户余额为 1100 元（丢失更新）	

数据并发访问所产生的问题，在有些场景下可能是允许的，但是有些场景下可能就是致命的，数据库通常会通过锁机制来解决数据并发访问问题，按锁定对象不同可以分为表级锁和行级锁；按并发事务锁定关系可以分为共享锁和独占锁，具体的内容大家可以自行查阅资料进行了解。

直接使用锁是非常麻烦的，为此数据库为用户提供了自动锁机制，只要用户指定会话的事务隔离级别，数据库就会通过分析 SQL 语句然后为事务访问的资源加上合适的锁，此外，数据库还会维护这些锁通过各种手段提高系统的性能，这些对用户来说都是透明的（就是说你不用理解，事实上我确实也不知道）。ANSI/ISO SQL 92 标准定义了 4 个等级的事务隔离级别，如下表所示：

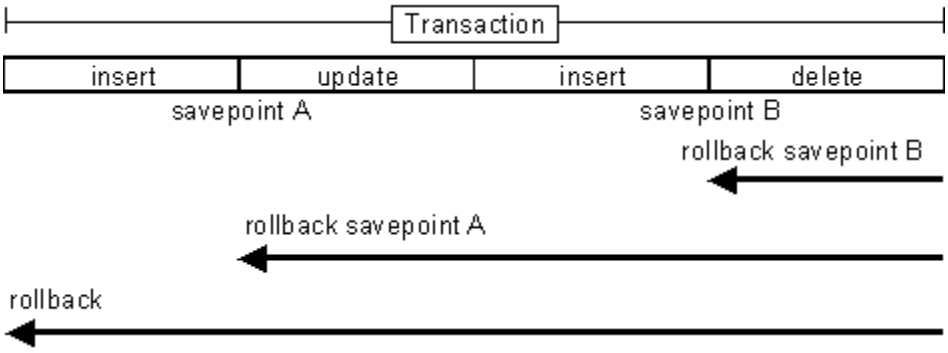
隔离级别	脏读	不可重复读	幻读	第一类丢失更新	第二类丢失

隔离级别	脏读	不可重复读	幻读	第一类丢失更新	第二类丢失
READ UNCOMMITED	允许	允许	允许	不允许	允许
READ COMMITTED	不允许	允许	允许	不允许	允许
REPEATABLE READ	不允许	不允许	允许	不允许	不允许
SERIALIZABLE	不允许	不允许	不允许	不允许	不允许

需要说明的是，事务隔离级别和数据访问的并发性是对立的，事务隔离级别越高并发性就越差。所以要根据具体的应用来确定合适的事务隔离级别，这个地方没有万能的原则。

81、JDBC 中如何进行事务处理？

答：Connection 提供了事务处理的方法，通过调用 setAutoCommit(false)可以设置手动提交事务；当事务完成后用 commit()显式提交事务；如果在事务处理过程中发生异常则通过 rollback()进行事务回滚。除此之外，从 JDBC 3.0 中还引入了 Savepoint（保存点）的概念，允许通过代码设置保存点并让事务回滚到指定的保存点。



82、JDBC 能否处理 Blob 和 Clob？

答：Blob 是指二进制大对象(Binary Large Object)，而 Clob 是指大字符对象(Character Large Objec)，因此其中 Blob 是为存储大的二进制数据而设计的，而 Clob 是为存储大的文本数据而设计的。JDBC 的 PreparedStatement 和 ResultSet 都提供了相应的方法来支持 Blob 和 Clob 操作。下面的代码展示了如何使用 JDBC 操作 LOB：

下面以 MySQL 数据库为例 ,创建一个张有三个字段的用户表 ,包括编号(id)、姓名(name ）和照片（ photo ），建表语句如下：

```
create table tb_user
(
id int primary key auto_increment,
name varchar(20) unique not null,
photo longblob
);
```

下面的 Java 代码向数据库中插入一条记录：

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

class JdbcLobTest {
```

```
    public static void main(String[] args) {
        Connection con = null;
        try {
            // 1. 加载驱动（Java6 以上版本可以省略）
            Class.forName("com.mysql.jdbc.Driver");
            // 2. 建立连接
            con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "123456");
            // 3. 创建语句对象
            PreparedStatement ps = con.prepareStatement("insert into tb_user values (default, ?, ?)");
            ps.setString(1, "骆昊");              // 将 SQL 语句中第一个占位符换成字符串
            try (InputStream in = new FileInputStream("test.jpg")) {    // Java 7 的 TWR
                ps.setBinaryStream(2, in);      // 将 SQL 语句中第二个占位符换成二进制流
                // 4. 发出 SQL 语句获得受影响行数
                System.out.println(ps.executeUpdate() == 1 ? "插入成功" : "插入失败");
            } catch(IOException e) {
                System.out.println("读取照片失败!");
            }
        } catch (ClassNotFoundException | SQLException e) {     // Java 7 的多异常捕获
            e.printStackTrace();
        } finally { // 释放外部资源的代码都应当放在 finally 中保证其能够得到执行
```

```
            try {
                if(con != null && !con.isClosed()) {
                    con.close();    // 5. 释放数据库连接
                    con = null;     // 指示垃圾回收器可以回收该对象
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
```

83、简述正则表达式及其用途。

答：在编写处理字符串的程序时，经常会有查找符合某些复杂规则的字符串的需要。正则表达式就是用于描述这些规则的工具。换句话说，正则表达式就是记录文本规则的代码。

说明：计算机诞生初期处理的信息几乎都是数值，但是时过境迁，今天我们使用计算机处理的信息更多的时候不是数值而是字符串，正则表达式就是在进行字符串匹配和处理的时候最为强大的工具，绝大多数语言都提供了对正则表达式的支持。

84、Java 中是如何支持正则表达式操作的？

答：Java 中的 String 类提供了支持正则表达式操作的方法，包括：matches()、replaceAll()、replaceFirst()、split()。此外，Java 中可以用 Pattern 类表示正则表达式对象，它提供了丰富的 API 进行各种正则表达式操作，请参考下面面试题的代码。

面试题：- 如果要从字符串中截取第一个英文左括号之前的字符串，例如：北京市(朝阳区)(西城区)(海淀区)，截取结果为：北京市，那么正则表达式怎么写？

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;


class RegExpTest {


    public static void main(String[] args) {

        String str = "北京市(朝阳区)(西城区)(海淀区)";

        Pattern p = Pattern.compile(".*?(?=\\()");

        Matcher m = p.matcher(str);

        if(m.find()) {

            System.out.println(m.group());

        }

    }

}
```

说明：上面的正则表达式中使用了懒惰匹配和前瞻，如果不清楚这些内容，推荐读一下网上很有名的《正则表达式 30 分钟入门教程》。

85、获得一个类的类对象有哪些方式？

答：

- 方法 1：类型.class，例如：String.class

- 方法 2：对象.getClass()，例如："hello".getClass()

- 方法 3：Class.forName()，例如：Class.forName("java.lang.String")

86、如何通过反射创建对象？

答：

- 方法 1：通过类对象调用 newInstance()方法，例如：String.class.newInstance()

- 方法 2：通过类对象的 getConstructor()或 getDeclaredConstructor()方法获得构造器（Constructor）对象并调用其 newInstance()方法创建对象，例如：String.class.getConstructor(String.class).newInstance("Hello");

87、如何通过反射获取和设置对象私有字段的值？

答：可以通过类对象的 getDeclaredField()方法字段（Field）对象，然后再通过字段对象的 setAccessible(true)将其设置为可以访问，接下来就可以通过 get/set 方法来获取/设置字段的值了。下面的代码实现了一个反射的工具类，其中的两个静态方法分别用于获取和设置私有字段的值，字段可以是基本类型也可以是对象类型且支持多级对象操作，例如 ReflectionUtil.get(dog, "owner.car.engine.id");可以获得 dog 对象的主人的汽车的引擎的 ID 号。

```
import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Modifier;
```

```java
import java.util.ArrayList;
import java.util.List;

/**
 * 反射工具类
 * @author 骆昊
 *
 */
public class ReflectionUtil {

    private ReflectionUtil() {
        throw new AssertionError();
    }

    /**
     * 通过反射取对象指定字段(属性)的值
     * @param target 目标对象
     * @param fieldName 字段的名字
     * @throws 如果取不到对象指定字段的值则抛出异常
     * @return 字段的值
     */
    public static Object getValue(Object target, String fieldName) {
        Class<?> clazz = target.getClass();
        String[] fs = fieldName.split("\\.");
```

```
        try {
            for(int i = 0; i < fs.length - 1; i++) {
                Field f = clazz.getDeclaredField(fs[i]);
                f.setAccessible(true);
                target = f.get(target);
                clazz = target.getClass();
            }

            Field f = clazz.getDeclaredField(fs[fs.length - 1]);
            f.setAccessible(true);
            return f.get(target);
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * 通过反射给对象的指定字段赋值
     * @param target 目标对象
     * @param fieldName 字段的名称
     * @param value 值
     */
```

```
    public static void setValue(Object target, String fieldName, Objec
t value) {

        Class<?> clazz = target.getClass();

        String[] fs = fieldName.split("\\.");

        try {

            for(int i = 0; i < fs.length - 1; i++) {

                Field f = clazz.getDeclaredField(fs[i]);

                f.setAccessible(true);

                Object val = f.get(target);

                if(val == null) {

                    Constructor<?> c = f.getType().getDeclaredConstructo
r();


                    c.setAccessible(true);

                    val = c.newInstance();

                    f.set(target, val);

                }

                target = val;

                clazz = target.getClass();

            }


            Field f = clazz.getDeclaredField(fs[fs.length - 1]);

            f.setAccessible(true);

            f.set(target, value);

        }

        catch (Exception e) {
```

```
            throw new RuntimeException(e);
        }
    }


}
```

- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74
- 75
- 76
- 77
- 78
- 79

88、如何通过反射调用对象的方法？

答：请看下面的代码：

```
import java.lang.reflect.Method;


class MethodInvokeTest {


    public static void main(String[] args) throws Exception {
        String str = "hello";
        Method m = str.getClass().getMethod("toUpperCase");
        System.out.println(m.invoke(str));  // HELLO
    }
}
```

89、简述一下面向对象的"六原则一法则"。

答：

- 单一职责原则：一个类只做它该做的事情。（单一职责原则想表达的就是"高内聚"，写代码最终极的原则只有六个字"高内聚、低耦合"，就如同葵花宝典或辟邪剑谱的中心思想就八个字"欲练此功必先自宫"，所谓的高内聚就是一个代码模块只完成一项功能，在面向对象中，

如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则，这个类就只有单一职责。我们都知道一句话叫"因为专注，所以专业"，一个对象如果承担太多的职责 那么注定它什么都做不好。这个世界上任何好的东西都有两个特征，一个是功能单一，好的相机绝对不是电视购物里面卖的那种一个机器有一百多种功能的，它基本上只能照相；另一个是模块化，好的自行车是组装车，从减震叉、刹车到变速器，所有的部件都是可以拆卸和重新组装的，好的乒乓球拍也不是成品拍，一定是底板和胶皮可以拆分和自行组装的，一个好的软件系统，它里面的每个功能模块也应该是可以轻易的拿到其他系统中使用的，这样才能实现软件复用的目标。）

- 开闭原则：软件实体应当对扩展开放，对修改关闭。（在理想的状态下，当我们需要为一个软件系统增加新功能时，只需要从原来的系统派生出一些新类就可以，不需要修改原来的任何一行代码。要做到开闭有两个要点：①抽象是关键，一个系统中如果没有抽象类或接口系统就没有扩展点；②封装可变性，将系统中的各种可变因素封装到一个继承结构中，如果多个可变因素混杂在一起，系统将变得复杂而换乱，如果不清楚如何封装可变性，可以参考《设计模式精解》一书中对桥梁模式的讲解的章节。）
- 依赖倒转原则：面向接口编程。（该原则说得直白和具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代，请参考下面的里氏替换原则。）

里氏替换原则：任何时候都可以用子类型替换掉父类型。（关于里氏替换原则的描述，Barbara Liskov 女士的描述比这个要复杂得多，但简单的说就是能用父类型的地方就一定能使用子类型。里氏替换原则可以检查继承关系是否合理，如果一个继承关系违背了里氏替换原则，那么这个继承关系一定是错误的，需要对代码进行重构。例如让猫继承狗，或者狗继承猫，又或者让正方形继承长方形都是错误的继承关系，因为你很容易找到违反里氏替换

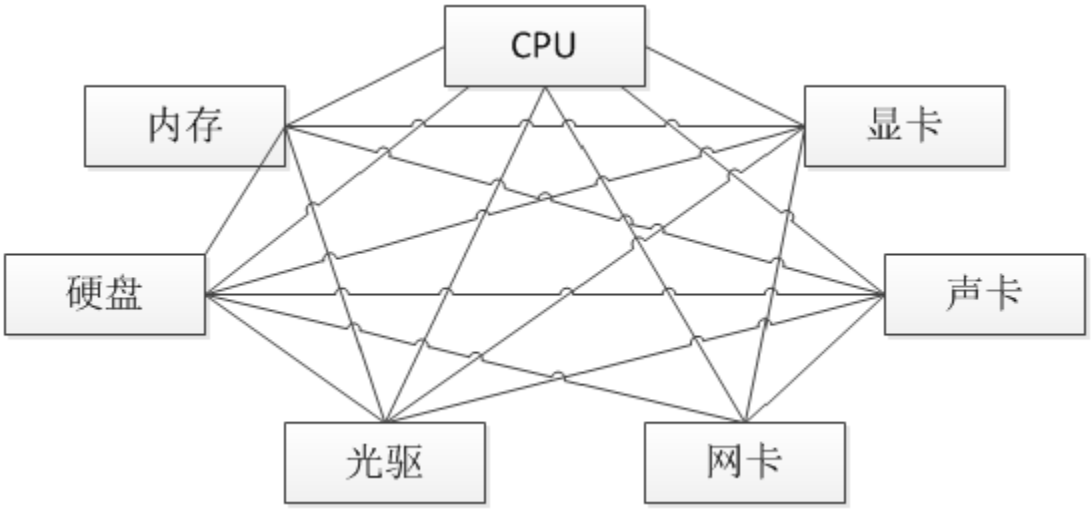
原则的场景。需要注意的是：子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。）

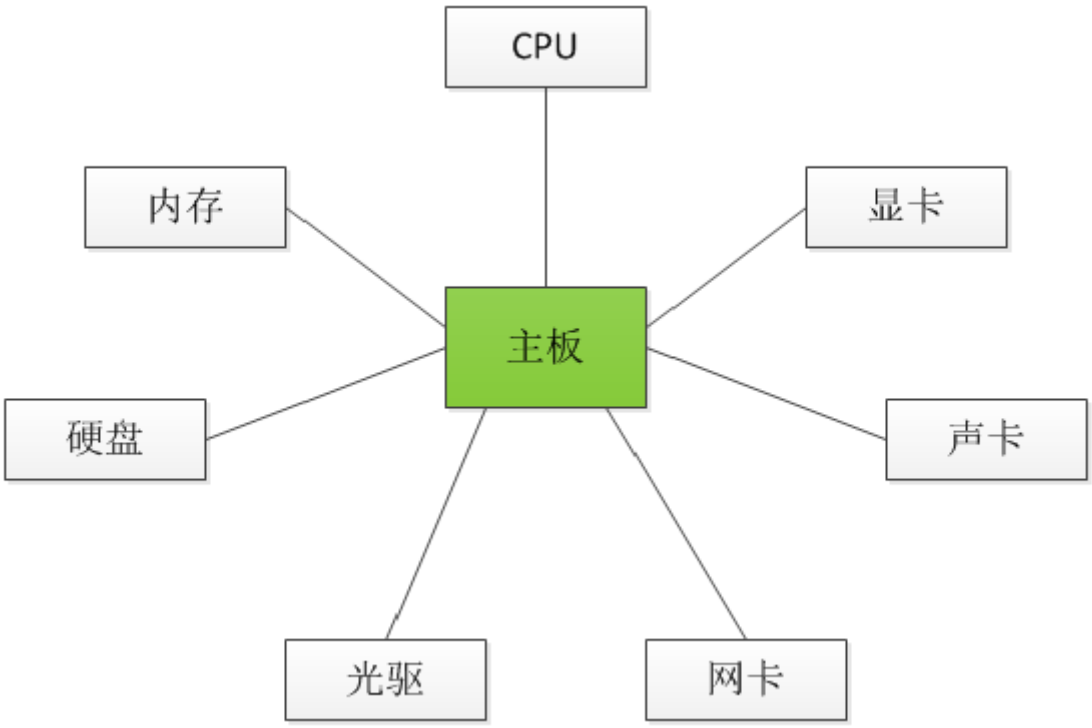
- 接口隔离原则：接口要小而专，绝不能大而全。（臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。例如，琴棋书画就应该分别设计为四个接口，而不应设计成一个接口中的四个方法，因为如果设计成一个接口中的四个方法，那么这个接口很难用，毕竟琴棋书画四样都精通的人还是少数，而如果设计成四个接口，会几项就实现几个接口，这样的话每个接口被复用的可能性是很高的。Java中的接口代表能力、代表约定、代表角色，能否正确的使用接口一定是编程水平高低的重要标识。）

- 合成聚合复用原则：优先使用聚合或合成关系复用代码。（通过继承来复用代码是面向对象程序设计中被滥用得最多的东西，因为所有的教科书都无一例外的对继承进行了鼓吹从而误导了初学者，类与类之间简单的说有三种关系，Is-A 关系、Has-A 关系、Use-A 关系，分别代表继承、关联和依赖。其中，关联关系根据其关联的强度又可以进一步划分为关联、聚合和合成，但说白了都是 Has-A 关系，合成聚合复用原则想表达的是优先考虑 Has-A 关系而不是 Is-A 关系复用代码，原因嘛可以自己从百度上找到一万个理由，需要说明的是，即使在 Java 的 API 中也有不少滥用继承的例子，例如 Properties 类继承了 Hashtable 类，Stack 类继承了 Vector 类，这些继承明显就是错误的，更好的做法是在 Properties 类中放置一个 Hashtable 类型的成员并且将其键和值都设置为字符串来存储数据，而 Stack 类的设计也应该是在 Stack 类中放一个 Vector 对象来存储数据。记住：任何时候都不要继承工具类，工具是可以拥有并可以使用的，而不是拿来继承的。）

- 迪米特法则 迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。（迪米特法则简单的说就是如何做到"低耦合"，门面模式和调停者模式就是对迪米特法则的

践行。对于门面模式可以举一个简单的例子，你去一家公司洽谈业务，你不需要了解这个公司内部是如何运作的，你甚至可以对这个公司一无所知，去的时候只需要找到公司入口处的前台美女，告诉她们你要做什么，她们会找到合适的人跟你接洽，前台的美女就是公司这个系统的门面。再复杂的系统都可以为用户提供一个简单的门面，Java Web 开发中作为前端控制器的 Servlet 或 Filter 不就是一个门面吗，浏览器对服务器的运作方式一无所知，但是通过前端控制器就能够根据你的请求得到相应的服务。调停者模式也可以举一个简单的例子来说明，例如一台计算机，CPU、内存、硬盘、显卡、声卡各种设备需要相互配合才能很好的工作，但是如果这些东西都直接连接到一起，计算机的布线将异常复杂，在这种情况下，主板作为一个调停者的身份出现，它将各个设备连接在一起而不需要每个设备之间直接交换数据，这样就减小了系统的耦合度和复杂度，如下图所示。迪米特法则用通俗的话来将就是不要和陌生人打交道，如果真的需要，找一个自己的朋友，让他替你和陌生人打交道。）





90、简述一下你了解的设计模式。

答：所谓设计模式，就是一套被反复使用的代码设计经验的总结（情境中一个问题经过证实的一个解决方案）。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使人们可以更加简单方便的复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。

在 GoF 的《Design Patterns: Elements of Reusable Object-Oriented Software》中给出了三类（创建型[对类的实例化过程的抽象化]、结构型[描述如何将类或对象结合在一起形成更大的结构]、行为型[对在不同的对象之间划分责任和算法的抽象化]）共 23 种设计模式，包括：Abstract Factory（抽象工厂模式），Builder（建造者模式），Factory Method（工厂方法模式），Prototype（原始模型模式），Singleton（单例模式）；Facade（门面模式），Adapter（适配器模式），Bridge（桥梁模式），Composite（合成模式），Decorator（装饰模式），Flyweight（享元模式），Proxy（代理模式）；Command（命令模式），Interpreter（解释器模式），Visitor（访问者模式），Iterator（迭代子模式），Mediator（调停者模式），Memento（备忘录模式），Observer（观察者模式），State（状态模式），Strategy（策略模式），Template Method（模板方法模式），Chain Of Responsibility（责任链模式）。

面试被问到关于设计模式的知识时，可以拣最常用的作答，例如：

- 工厂模式：工厂类可以根据条件生成不同的子类实例，这些子类有一个公共的抽象父类并且实现了相同的方法，但是这些方法针对不同的数据进行了不同的操作（多态方法）。当得到子类的实例后，开发人员可以调用基类中的方法而不必考虑到底返回的是哪一个子类的实例。
- 代理模式：给一个对象提供一个代理对象，并由代理对象控制原对象的引用。实际开发中，按照使用目的的不同，代理可以分为：远程代理、虚拟代理、保护代理、Cache 代理、防火墙代理、同步化代理、智能引用代理。
- 适配器模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起使用的类能够一起工作。
- 模板方法模式：提供一个抽象类，将部分逻辑以具体方法或构造器的形式实现，然后声明

一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法（多态实现），从而实现不同的业务逻辑。

除此之外，还可以讲讲上面提到的门面模式、桥梁模式、单例模式、装潢模式（Collections 工具类和 I/O 系统中都使用装潢模式）等，反正基本原则就是拣自己最熟悉的、用得最多的作答，以免言多必失。

91、用 Java 写一个单例类。

答：

- 饿汉式单例

```
public class Singleton {

    private Singleton(){}

    private static Singleton instance = new Singleton();

    public static Singleton getInstance(){

        return instance;

    }

}
```

- 懒汉式单例

```
public class Singleton {

    private static Singleton instance = null;
```

```
    private Singleton() {}
    public static synchronized Singleton getInstance(){
        if (instance == null) instance = new Singleton();
        return instance;
    }
}
```

注意：实现一个单例有两点注意事项，①将构造器私有，不允许外界通过构造器创建对象；②通过公开的静态方法向外界返回类的唯一实例。这里有一个问题可以思考：Spring 的 IoC 容器可以为普通的类创建单例，它是怎么做到的呢？

92、什么是 UML？

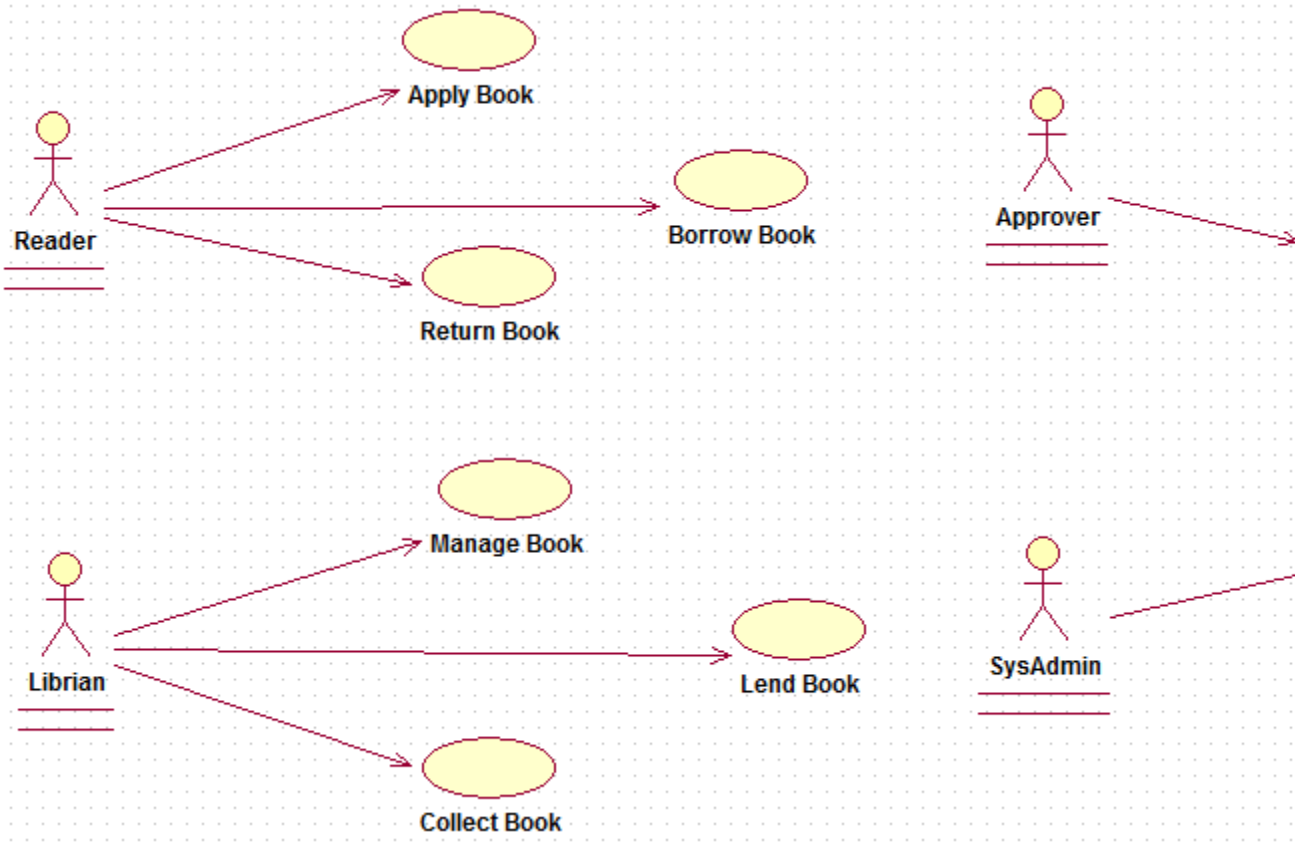
答：UML 是统一建模语言（Unified Modeling Language）的缩写，它发表于 1997 年，综合了当时已经存在的面向对象的建模语言、方法和过程，是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持。使用 UML 可以帮助沟通与交流，辅助应用设计和文档的生成，还能够阐释系统的结构和行为。

93、UML 中有哪些常用的图？

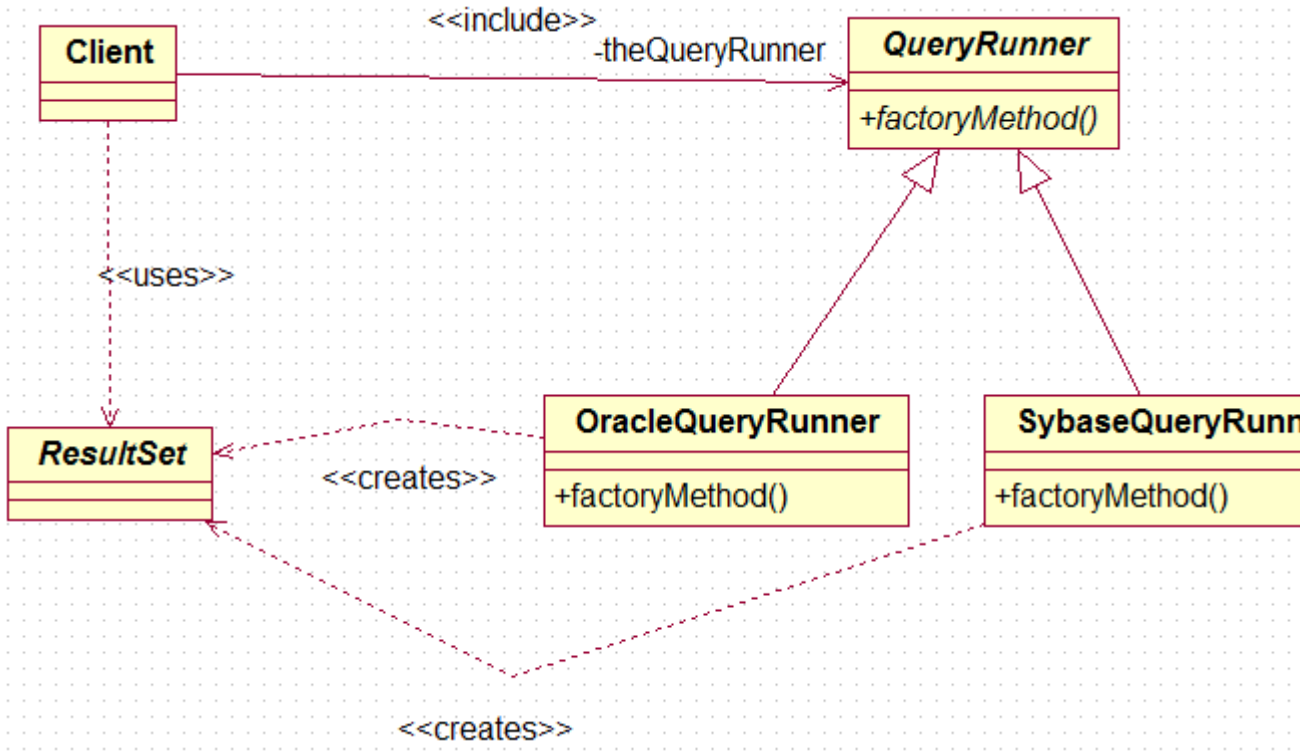
答：UML 定义了多种图形化的符号来描述软件系统部分或全部的静态结构和动态结构，包括：用例图（use case diagram）、类图（class diagram）、时序图（sequence diagram）、协作图（collaboration diagram）、状态图（statechart diagram）、活动图（activity

diagram）、构件图（component diagram）、部署图（deployment diagram）等。在这些图形化符号中，有三种图最为重要，分别是：用例图（用来捕获需求，描述系统的功能，通过该图可以迅速的了解系统的功能模块及其关系）、类图（描述类以及类与类之间的关系，通过该图可以快速了解系统）、时序图（描述执行特定任务时对象之间的交互关系以及执行顺序，通过该图可以了解对象能接收的消息也就是说对象能够向外界提供的服务）。

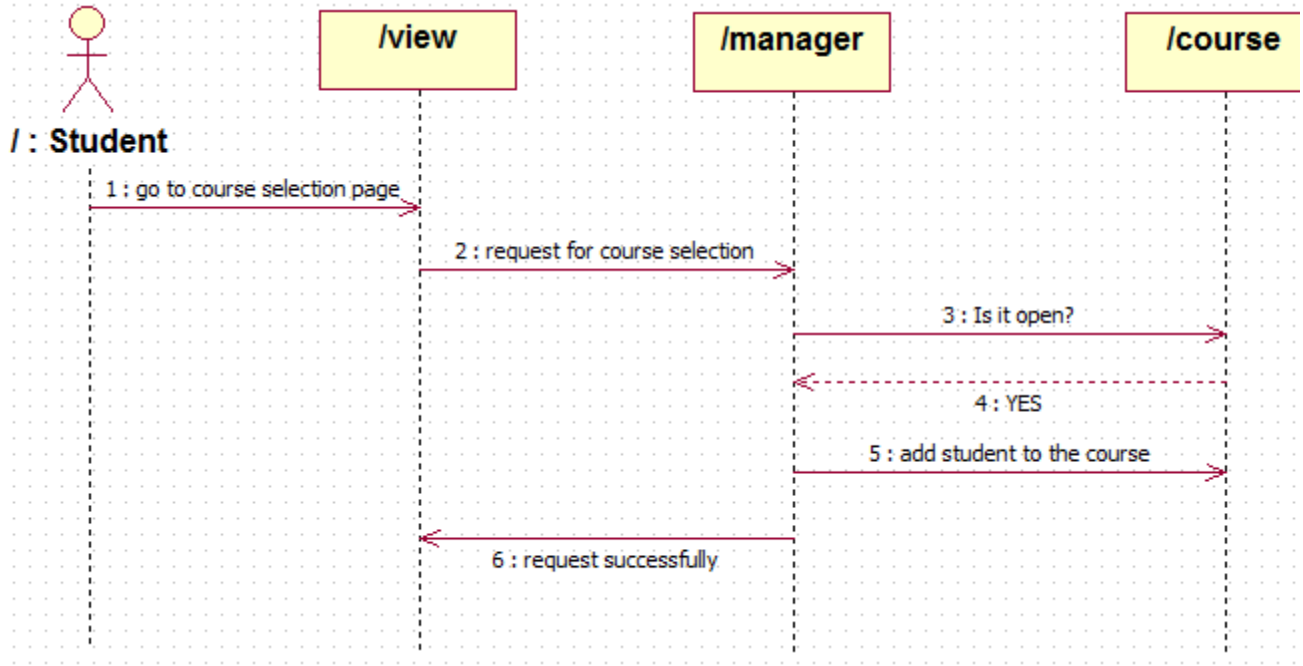
用例图：



类图：


Client
<<include>>
-theQueryRunner
QueryRunner
+factoryMethod()
<<uses>>
ResultSet
<<creates>>
OracleQueryRunner
+factoryMethod()
SybaseQueryRunn
+factoryMethod()
<<creates>>

时序图：


/ : Student
/view
/manager
/course
1 : go to course selection page
2 : request for course selection
3 : Is it open?
4 : YES
5 : add student to the course
6 : request successfully

94、用 Java 写一个冒泡排序。

答：冒泡排序几乎是个程序员都写得出来，但是面试的时候如何写一个逼格高的冒泡排序却不是每个人都能做到，下面提供一个参考代码：

```
import java.util.Comparator;


/**
 * 排序器接口(策略模式：将算法封装到具有共同接口的独立的类中使得它们可以相互替换)
 * @author 骆昊
 *
 */
public interface Sorter {

    /**
     * 排序
     * @param list 待排序的数组
     */
    public <T extends Comparable<T>> void sort(T[] list);


    /**
     * 排序
     * @param list 待排序的数组
     * @param comp 比较两个对象的比较器
     */
    public <T> void sort(T[] list, Comparator<T> comp);
```

```
}
```

```
import java.util.Comparator;

/**
 * 冒泡排序
 *
 * @author 骆昊
 *
 */
public class BubbleSorter implements Sorter {

    @Override
```

```
public <T extends Comparable<T>> void sort(T[] list) {
    boolean swapped = true;
    for (int i = 1, len = list.length; i < len && swapped; ++i) {
        swapped = false;
        for (int j = 0; j < len - i; ++j) {
            if (list[j].compareTo(list[j + 1]) > 0) {
                T temp = list[j];
                list[j] = list[j + 1];
                list[j + 1] = temp;
                swapped = true;
            }
        }
    }
}


@Override
public <T> void sort(T[] list, Comparator<T> comp) {
    boolean swapped = true;
    for (int i = 1, len = list.length; i < len && swapped; ++i) {
        swapped = false;
        for (int j = 0; j < len - i; ++j) {
            if (comp.compare(list[j], list[j + 1]) > 0) {
                T temp = list[j];
                list[j] = list[j + 1];
```

```
                    list[j + 1] = temp;
                    swapped = true;
                }
            }
        }
    }
}
```

```
32
33
34
35
36
37
38
39
40
41
42
```

95、用 Java 写一个折半查找。

答：折半查找，也称二分查找、二分搜索，是一种在有序数组中查找某一特定元素的搜索算法。搜索过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组已经为空，则表示找不到指定的元素。这种搜索算法每一次比较都使搜索范围缩小一半，其时间复杂度是 O(logN)。

```
import java.util.Comparator;


public class MyUtil {


   public static <T extends Comparable<T>> int binarySearch(T[] x, T ke
y) {

      return binarySearch(x, 0, x.length- 1, key);

   }


   // 使用循环实现的二分查找

   public static <T> int binarySearch(T[] x, T key, Comparator<T> comp)
 {
```

```
    int low = 0;
    int high = x.length - 1;
    while (low <= high) {
        int mid = (low + high) >>> 1;
        int cmp = comp.compare(x[mid], key);
        if (cmp < 0) {
          low= mid + 1;
        }
        else if (cmp > 0) {
          high= mid - 1;
        }
        else {
          return mid;
        }
    }
    return -1;
  }


  // 使用递归实现的二分查找
  private static<T extends Comparable<T>> int binarySearch(T[] x, int
low, int high, T key) {
    if(low <= high) {
      int mid = low + ((high -low) >> 1);
      if(key.compareTo(x[mid])== 0) {
         return mid;
```

```
        }
        else if(key.compareTo(x[mid])< 0) {
           return binarySearch(x,low, mid - 1, key);
        }
        else {
           return binarySearch(x,mid + 1, high, key);
        }
      }
      return -1;
   }
}
```

```
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
```

说明：上面的代码中给出了折半查找的两个版本，一个用递归实现，一个用循环实现。需要注意的是计算中间位置时不应该使用(high+ low) / 2 的方式，因为加法运算可能导致整数越界，这里应该使用以下三种方式之一 :low + (high - low) / 2 或 low + (high − low) >> 1 或(low + high) >>> 1（ >>>是逻辑右移，是不带符号位的右移）