疯狂 Java 面试题（疯狂 Java 讲义精粹附赠）

本大全每个月会定期更新，索取网址：http://www.fkit.org。

Java 核心技术部分

Java 核心技术部分的面试题，可能覆盖 Java 基本语法、面向对象（包括类定义、方法、构造器、递归、继承、抽象类、接口、枚举以及 final、static 等关键字）、Java 常用 API、Java 集合框架（需要重点掌握）、注解（Annotation）、泛型、输入/输出、多线程、网络通信、反射、内存管理等相关内容，这些知识基本都可通过《疯狂 Java 讲义》一书找到详细解答。

这部分面试题大部分从网络收集、整理，也有部分题目来自疯狂软件学员面试之后的反馈。

1．面向对象的特征有哪些？

面向对象的三大特征：

继承：通过继承允许复用已有的类，继承关系是一种“一般到特殊”的关系，比如苹果类继承水果类，这个过程称为类继承。

派生出来的新类称为原有类的子类（派生类），而原有类称为新类的父类（基类）。

子类可以从父类那里继承得到方法和成员变量，而且子类可以修改或增加新的方法使之适合子类的需要。

封装：封装是把对象的状态数据隐藏起来，再通过暴露合适的方法来允许外部程序修改对象的状态数据。Java 的封装主要通过 private、protected、public 等访问控制符来实现。

多态：多态指的是当同一个类型的引用类型的变量在执行相同的方法时，实际上会呈现出多种不同的行为特征。比如程序有 Animal a1 = new Animal (); Animal a2 = new Wolf();，虽然 a1、a2 两个引用变量的类型都是 Animal，但当它们调用同一个 run()方法时，如果 Wolf()类重写过 Animal 的 run()方法，这就会导致 a1、a2 两个变量执行 run()方法时呈现出不同的行为特征，这就是多态。多态增加了编程的灵活性，实际上大量设计模式都是基于多态类实现的。

除此之外，**抽象**也是一个重要的特征，抽象就是忽略与当前目标无关的相关方面，以便更充分地突出与当前目标有关的方面。抽象并不打算了解全部问题，而只是选择其中的一部分，暂时不用部分细节。抽象包括两个方面：一是过程抽象；二是数据抽象。

2．Java 中实现多态的机制是什么？

Java 允许父类或接口定义的引用变量指向子类或具体实现类的实例对象，而程序调用的方法在运行时才动态绑定，就是引用变量所指向的具体实例对象的方法，也就是内存里正在运行的那个对象的方法，而不是引用变量的类型中定义的方法。

正是由于这种机制，两个相同类型的引用变量，由于它们实际引用了不同的对象，因此它们运行时可能呈现出多种不同的行为特征，这就被称多态。

3. 一个.java 源文件中是否可以包括多个类（不是内部类）？有什么限制？

可以有多个类，但只能有一个 public 的类，并且 public 的类名必须与文件的主文件名相同。

包含多个类的 Java 源文件编译之后会生成多个.class 文件，每个类（包括外部类、内部类）都会生成一个对应的.class 文件。

4. String 是基本数据类型吗？

基本数据类型包括 byte、short、int、long、char、float、double 和 boolean。String 不是基本类型。String 是引用类型。

java.lang.String 类是 final 的，因此无法通过 String 类派生子类。

String 也是一个不可变类（它所包含的字符序列是不可改变的），因此如果程序需要使用的字符串所包含的字符序列需要经常改变，建议使用 StringBuffer（线程安全、性能略差）类或 StringBuilder 类。

5. int 和 Integer 有什么区别？

Java 提供两种不同的类型：引用类型和基本数据类型。

int 是基本数据类型，Integer 是 Java 为 int 提供的包装类。

Java 为所有的基本类型提供了对应的包装类。

byte Byte

short Short

int Integer

long Long

char Character

float Float

double Double

boolean Boolean

基本类型的变量只能当成简单的直接量、参与表达式运算，不具备面向对象的特征，基本类型的变量不能被赋为 null；但包装类的变量则完全可以当成对象使用，它具有面向对象的特征，包装类的变量可以被赋为 null。

因为 Integer 具有面向对象的特征，所以 Integer 可以区分出未赋值和值为 0 的区别，int 则无法表达出未赋值的情况。例如，要想表达出没有参加考试和考试成绩为 0 的区别，则只能使用 Integer。在 JSP 开发中，Integer 的默认为 null，所以用 EL 输出为 null 的 Integer 时，将会显示为空白字符串，而 int 的默认值为 0，用 EL 输出将显示 0。所以，int 不适合作为 Web 层的表单数据的类型。

从 Java 5 开始，Java 提供了自动装箱、自动拆箱功能，因此包装类也可以直接参与表达式运算，所以使用起来十分方便。

另外，Integer 提供了多个与整数相关的操作方法，例如，将一个字符串转换成整数，

Integer 中还定义了表示整数的最大值和最小值的常量。

6．Java 有没有 goto？

goto 是 Java 中的保留字，Java 程序的标识符不允许使用 goto。但 Java 也不支持使用 goto 进行跳转。

7．String 和 StringBuffer、StringBuilder 的区别是什么？

Java 提供了 String、StringBuffer 和 StringBuilder，它们都是 CharSequence 的实现类，都可以作为字符串使用。

String 代表字符序列不可变的字符串；而 StringBuffer、StringBuilder 都代表字符序列可变的字符串。

StringBuffer 和 StringBuilder 的区别是 StringBuffer 是线程安全的、性能略低，而 StringBuilder 是线程不安全的，适合单线程环境使用，性能较好。

8．Collection 和 Collections 的区别是什么？

Collection 是集合类（List、Set、Queue）的根接口。

Collections 是针对集合类的一个工具类，它提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

9．说说&和&&的区别。

&和&&都可以用作逻辑与的运算符，当运算符两边的表达式的结果都为 true 时，整个运算结果才为 true；否则，只要有一方为 false，结果就为 false。

&&还具有短路的功能，即如果第一个表达式为 false，则不再计算第二个表达式。例如，对于 if(a >8 && b > 5)，当 a 小于等于 8 时，由于&&之前的表达式已经为 false 了，因此&&之后的表达式根本不会执行。

再例如 if(str != null && !str.equals(""))表达式，当 str 为 null 时，后面的表达式不会执行，因此不会出现 NullPointerException，如果将&&改为&，则可能抛出 NullPointerException 异常。

再例如 if(x > 8 & ++y)与 if(x > 8 && ++y)，当 a 小于等于 8 时，前一个表达式中 y 的值会增长；后一个表达式中 y 的值不会增加。

除此之外，&还可以用作位运算符，当&操作符两边的表达式不是 boolean 类型时，&表示按位与操作，通常使用 0x0f 与一个整数进行&运算，来获取该整数的最低 4 个 bit 位，例如，0x31 & 0x0f 的结果为 0x01。

10．Overload 和 Override 的区别是什么？Overloaded 的方法是否可以改变返回值的类型？

Overload 是方法的重载。

Override 是方法的重写，也叫覆盖。

Overload 要求两个方法具有方法名相同、形参列表不同的要求，返回值类型不能作为重载的条件。

Override 要求子类方法与父类方法具有“两同两小一大”的要求。“两同”指：父类方法、子类方法的方法名相同、形参列表相同；“两小”指：子类方法返回值类型要么是父类方法返回值类型的子类，要么与父类方法返回值类型相同；子类方法声明抛出的异常类型要么是父类方法声明抛出的异常类型的子类，要么与父类声明抛出的异常类型相同；“一大”指：子类方法的访问权限要么与父类方法的访问权限相同，要么比父类方法的访问权限更大。

Overloaded 的方法是可以改变返回值的类型。

11．Java 如何跳出当前的多重嵌套循环？

在 Java 中，要想跳出多重循环，可以在外面的循环语句前定义一个标号，然后在里层循环体的代码中使用带有标号的 break 语句，即可跳出外层循环。例如：

```
outer:
for(int i=0;i<10;i++)
{
    for(int j=0;j<10;j++)
    {
        System.out.println("i=" + i + ",j=" + j);
        if(j == 5) break ouer;
    }
}
```

12．switch 语句能否作用在 byte 上？能否作用在 long 上？能否作用在 String 上？

在 Java 7 以前，在 switch（expr1）中，expr1 只能是一个整数表达式（但不包括 long 和 Long）或者枚举常量，整数表达式可以是 int 基本类型或 Integer 包装类型，byte、short、char 都可以自动转换为 int，它们都可作为 switch 表达式。

从 Java 7 开始，switch 表达式可以使用 String。

13. String s = new String("xyz");创建了几个 String Object？

两个。一个是直接量的"xyz"字符串对象，该字符串将会被缓存在字符串常量池中，以

便以后复用这个字符串；另一个是通过 new String()构造器创建出来的 String 对象，这个 String 对象保存在堆内存中。

通常来说，应该尽量使用直接量的 String 对象，这样具有更好的性能。

14．数组有没有 length()这个方法？String 有没有 length()这个方法？

数组没有 length()这个方法，有 length 属性。String 有 length()方法。

15．short s1 = 1; s1 = s1 + 1;有什么错？short s1 = 1; s1 += 1;有什么错？

对于 short s1 = 1; s1 = s1 + 1;，由于 s1+1 运算时会自动提升表达式的类型，所以结果是 int 类型，再赋值给 short 类型 s1 时，编译器将报告需要强制类型转换的错误。

对于 short s1 = 1; s1 += 1;，由于 +=运算符里已经包括了一个隐式的强制类型转换，因此 Java 会把 s1+=1 计算后的结果进行隐式的强制类型转换，所以它不会有任何错误。

16．Char 类型变量中能不能存储一个中文字符？为什么？

Char 类型变量是用来存储 Unicode 编码的字符的，Unicode 编码字符集中包含了汉字，因此 char 类型变量中可以存储汉字。不过，如果某个特殊的汉字没有被包含在 Unicode 编码字符集中，那么 char 类型变量中就不能存储这个特殊汉字。

char 类型的变量占两个字节，而 Unicode 编码中每个字符也占两个字节，因此 char 类型的变量可以存储任何一个 Unicode 字符。

17．最有效率的方法算出 2 乘以 8 等于几？

2 << 3。

因为将一个数左移 n 位，就相当于乘以 2 的 n 次方，那么一个数乘以 8 只要将其左移 3 位即可，而位运算 CPU 直接支持，效率最高，所以，2 乘以 8 等于几的最有效率的方法是 2 << 3。

但需要注意的是，如果这个数字本身已经很大，比如本身已经是 2 的 30 次方了，此时再用这种位移运算就可能导致“溢出”，这样就得不到正确结果了。

18．使用 final 关键字修饰一个变量时，是引用不能变，还是引用的对象不能变？

使用 final 关键字修饰一个变量时，是指引用变量不能变，引用变量所指向的对象中的

内容还是可以改变的。例如，对于如下语句：

```
final StringBuilder a = new StringBuilder ("immutable");
```

执行如下语句将报告编译错误：

```
a = new StringBuilder ("");
```

但如下语句则是完全正确的：

```
a.append("fkjava.org");
```

有人希望在定义方法的形参时，通过 final 修饰符来阻止方法内部修改传进来的实参：

```
public void method(final StringBuilder param)
{
}
```

实际上这没有用，在该方法内部仍然可以增加如下代码来修改实参对象：

```
param.append("fkjava.org");
```

19．"=="和 equals 方法究竟有什么区别？

==操作符的作用有两种。

A．如果==的两边都是基本类型变量、包装类对象所组成的表达式，==用于比较两边的表达式的值是否相等——只要两边的表达式的值相等，即使数据类型不同，该运算符也会返回 true。例如'a' == 97.0，将会返回 true。

B．如果==的两边是引用类型的变量，==用于判断这两个引用类型的变量是否引用同一块内存，只有当它们引用同一块内存时，==才会返回 true。

而 equals()则是 java.lang.Object 类的一个方法，因此任何 Java 对象都可调用该方法与其他对象进行比较。java.lang.Object 类的 equals 方法的实现代码如下：

```
boolean equals(Object o)
{
    return this==o;
}
```

从上面代码可以看出，如果一个类没有重写 java.lang.Object 的 equals()方法时，此时 equals()方法的比较结果与==的比较结果是相同的。

但 Java 允许任何类重写 equals()方法，重写该方法就是让程序员来自己决定两个对象相等的标准——在极端的情况下，我们完全可以设计出 Person 对象与 Dog 对象 equals()比较返回 true 的情况——当然一般不会这么设计。

实际上重写 equals()方法时通常会按如下格式：

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

```
            if (other.name != null)
                return false;
        }
        else if (!name.equals(other.name))
            return false;
        if (pass == null)
        {
            if (other.pass != null)
                return false;
        }
        else if (!pass.equals(other.pass))
            return false;
        return true;
    }
```

上面重写 equals()方法用于判断两个 Person 对象是否“相等”，程序只要两个 Person 对象的 name、pass 相等，程序就可以把这两个 Person 对象当成相等——这是系统业务决定的。如果业务需要，我们也可以增加更多的参与判断的 Field，当然也可以只根据 name 进行判断——只要两个 Person 的 name 相等，就认为两个 Person 相等，这都是由系统的业务决定的。

总结起来就是一句话：开发者重写 equals()方法就可以根据业务要求来决定两个对象是否“相等”。

20．静态变量和实例变量的区别是什么？

在语法定义上的区别：静态变量前要加 static 关键字，而实例变量前则不加。

在程序运行时的区别：实例变量属于一个对象，必须先创建实例对象，它的实例变量才会被分配空间，才能使用这个实例变量。静态变量则属于类，所以也称为类变量，只要程序加载了类的字节码，不用创建任何实例对象，静态变量就会被分配空间，静态变量就可以被使用了。总之，实例变量必须创建对象后才可以通过这个对象来使用，静态变量则可以直接使用类名来引用。

例如，对于下面的程序：

```
public class VarTest
{
    public static int staticVar = 0;
    public int instanceVar = 0;
    public VarTest ()
    {
        staticVar++;
        instanceVar++;
        System.out.println("staticVar="    +    staticVar    +
",instanceVar=" + instanceVar);
    }
}
```

上面程序中的 staticVar 变量随着 VarTest 类初始化而分配内存、执行初始化，以后无论创建多少个实例对象，都不会再分配 staticVar 变量，因此永远只有一个 staticVar 变量。

但 instanceVar 变量则是随着 VarTest 对象初始化而分配内存、执行初始化的，因此每创建一个实例对象，就会分配一个 instanceVar，即可以分配多个 instanceVar。因此，上面程序中每创建一个 VarTest 对象，staticVar 的值就会自加 1，但创建每个 VarTest 对象的 instanceVar 都只自加 1。

21．是否可以从一个 static 方法内部调用非 static 方法？

不可以。静态成员不能调用非静态成员。

非 static 方法属于对象，必须创建一个对象后，才可以通过该对象来调用 static 方法。而 static 方法调用时不需要创建对象，通过类就可以调用该方法。也就是说，当一个 static 方法被调用时，可能还没有创建任何实例对象，如果允许从一个 static 方法中调用非 static 方法，那么非 static 方法是没有调用对象的。因此，Java 不允许 static 方法内部调用非 static 方法。

22．Math.round(11.5)等于多少？Math.round(-11.5)等于多少?

Math 类中提供了三个与取整有关的方法：ceil()、floor()、round()，这些方法的作用与它们的英文名称的含义相对应，例如，ceil 的英文意义是天花板，该方法就表示向上取整，所以，Math.ceil(11.3)的结果为 12，Math.ceil(-11.3)的结果是-11；floor 的英文意义是地板，该方法就表示向下取整，所以，Math.floor(11.6)的结果为 11，Math.floor(-11.6)的结果是-12；最难掌握的是 round 方法，它表示“四舍五入”，算法为 Math.floor(x+0.5)，即将原来的数字加上 0.5 后再向下取整，所以，Math.round(11.5)的结果为 12，Math.round(-11.5)的结果为-11。

23．请说出作用域 public、private、protected，以及不写时的区别。

这 4 个作用域的可见范围如下表所示。

作用域	当前类	同一 package	子类	全局
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

说明：如果在修饰的元素上面没有写任何访问修饰符，则表示 default。

只要记住访问权限由小到大依次是 private → default → protected → public，然后再记住 Java 存在的 4 个访问范围，就很容易画出上面的表格了。

24．外部类能用 private、protected 修饰吗？内部类可以用 private、protected 修饰吗？

外部类不能用 private、protected 修饰。内部类能用 private、protected 修饰。

外部类的上一级程序单位是包，因此它只有两个使用范围：包内和包外，所以它只能用 public（表示可以在全局位置使用）和默认修饰符（default，表示只能被同一个包的其他类使用）修饰。

内部类的上一级程序单位是类，因此它有 4 个使用范围：当前类、同一个包内、当前类的子类中、全局范围，因此可以使用 private、默认修饰符、protected、public 的任意一个修饰符修饰。

25．一个类定义多个重载方法，参数分别是 int、char 和 double，然后将 double x = 2 传递进去，会选择哪个方法？

选择参数类型为 double 的方法。

26．说说 has a 与 is a 的区别。

is a 是典型的“一般到特殊”的关系，也就是典型的继承关系。例如 Apple is a Fruit，那么 Apple 是一种特殊的 Fruit，也就是说，Apple 继承了 Fruit。

has a 是典型的“组合”关系。比如 Wolf has a Leg，也就是 Leg 组合成了 Wolf。

需要指出的是，由于继承会造成对父类的破坏，因此有时候可以通过组合来代替继承。使用继承的好处是，程序语义更好理解；坏处是，子类可能重写父类方法，不利于父类封装。使用组合则会造成语义的混淆，但组合类不会重写被组合类的方法，因此更利于被复合类的封装。

27．ClassLoader 如何加载 class？

JVM 里有多个类加载，每个类加载可以负责加载特定位置的类，例如，Bootstrap 类加载器（根类加载器）负责加载 Java 的核心类（jre/lib/rt.jar 中的类）， JDK 常用的 String、Math、HashSet、ArrayList 等类都位于 rt.jar 中；Extension 类加载器负责加载 jar/lib/ext/*.jar 中的类，应用类加载器（App ClassLoader）负责加载 CLASSPATH 指定的目录或 JAR 包中的类。除了 Bootstrap 之外，其他的类加载器本身也都是 Java 类，它们的父类是 ClassLoader。

Bootstrap 类加载器（根类加载器）非常特殊，它并不是 java.lang.ClassLoader 的子类，而是由 JVM 自身实现的。

28．GC 是什么？为什么要有 GC？

GC 是垃圾收集（Gabage Collection）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 的 System 类和 Runtime 类都提供了“通知”程序进行垃圾回收的方法，例如如下代码：

```
Systme.gc();
```

或

```
Runtime.getInstance().gc();
```

但这两个方法只是“通知”Java 进行垃圾回收，但实际上 JVM 何时进行垃圾回收，还是由 JVM 自己决定。

29．垃圾回收的优点和原理，并考虑两种回收机制。

传统的 C/C++等编程语言，需要程序员负责回收已经分配的内存。显式进行垃圾回收是一件比较困难的事情，因为程序员并不总是知道内存应该何时被释放。如果一些分配出去的内存得不到及时回收，就会引起系统运行速度下降，甚至导致程序瘫痪，这种现象被称为内存泄漏。总体而言，显式进行垃圾回收主要有如下两个缺点：

A．程序忘记及时回收无用内存，从而导致内存泄漏，降低系统性能。

B．程序错误地回收程序核心类库的内存，从而导致系统崩溃。

与 C/C++语言不同，Java 语言不需要程序员直接控制内存回收，Java 程序的内存分配和回收都是由 JRE 在后台自动进行的。JRE 会负责回收那些不再使用的内存，这种机制被称为垃圾回收（Garbage Collection， GC）。通常 JRE 会提供一条后台线程来进行检测和控制，一般都是在 CPU 空闲或内存不足时自动进行垃圾回收，而程序员无法精确控制垃圾回收的时间和顺序等。

实际上，垃圾回收机制不可能实时检测到每个 Java 对象的状态，当一个对象失去引用后，它也不会被立即回收，只有等垃圾回收器运行时才会被回收。

对于一个垃圾回收器的设计算法来说，大致有如下可供选择的设计。

A．串行回收（Serial）和并行回收（Parallel）：串行回收就是不管系统有多少个 CPU，始终只用一个 CPU 来执行垃圾回收操作；而并行回收就是把整个回收工作拆分成多个部分，每个部分由一个 CPU 负责，从而让多个 CPU 并行回收。并行回收的执行效率很高，但复杂度增加，另外也有其他一些副作用，比如内存碎片会增加。

B．并发执行（Concurrent）和应用程序停止（Stop-the-world）：Stop-the-world 的垃圾回收方式在执行垃圾回收的同时会导致应用程序的暂停。并发执行的垃圾回收虽然不会导致应用程序的暂停，但由于并发执行垃圾回收需要解决和应用程序的执行冲突（应用程序可能会在垃圾回收的过称中修改对象），因此并发执行垃圾回收的系统开销比 Stop-the-world 更好，而且执行时也需要更多的堆内存。

C．压缩（Compacting）、不压缩（Non-compacting）和复制（Copying）：为了减少内存碎片，支持压缩的垃圾回收器会把所有的活对象搬迁到一起，然后将之前占用的内存全部回收。不压缩式的垃圾回收器只是回收内存，这样回收回来的内存不可能是连续的，因此将会有较多的内存碎片。较之压缩式的垃圾回收，不压缩式的垃圾回收回收内存快了，而分配内存时就会更慢，而且无法解决内存碎片的问题。复制式的垃圾回收会将所有的可达对象复制到另一块相同的内存中，这种方式的优点是垃圾及回收过程不会产生内存碎片，但缺点也很明显，需要拷贝数据和额外的内存。

30．垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？

对于 Java 程序中的对象而言，如果这个对象没有任何引用变量引用它，那么这个对象将不可能被程序访问，因此可认为它是垃圾；只要有一个以上的引用变量引用该对象，该对象就不会被垃圾回收。

对于 Java 的垃圾回收器来说，它使用有向图来记录和管理堆内存中的所有对象，通过这个有向图就可以识别哪些对象是“可达的”（有引用变量引用它就是可达的），哪些对象是“不可达的”（没有引用变量引用它就是不可达的），所有的“不可达”对象都是可被垃圾回收的。

但对于如下程序：

```
class A
{
    B b;
}
class B
{
    A a;
}
public class Test
{
    public static void main(String[] args)
    {
        A a = new A();
        a.b = new B();
        a.b.a = a;
        a = null;
    }
}
```

上面程序中的 A 对象、B 对象它们“相互”引用，A 对象的 b 属性引用 B 对象，而 B 对象的 a 属性引用 A 对象，但实际上没有引用变量引用 A 对象、B 对象，因此它们在有向图中依然是不可达的，所以也会被当成垃圾处理。

程序员可以手动执行 System.gc()，通知 GC 运行，但这只是一个通知，而 JVM 依然有权决定何时进行垃圾回收。

31．什么时候用 assert？

assert（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。在实现中，assert 就是程序中的一条语句，它对一个 boolean 表达式进行检查，一个正确程序必须保证这个 boolean 表达式的值为 true；如果该值为 false，说明程序已经处于不正确的状态下，assert 将给出警告或退出。

Java 的 assert 是关键字。

```
public class TestAssert
{
    public static void main(String[] args)
    {
        int a = 5;
        // 断言 a>3
        assert a > 3;
        // 断言 a<3，否则显示 a 不小于 3，且 a 的值为：" + a
        assert a < 3 : "a 不小于 3，且 a 的值为：" + a;
    }
}
```

从上面代码可以看出，assert 的两个基本用法如下：

```
assert logicExp;
asert logicExp : expr;
```

A．第一个直接进行断言。

B．第二个也是进行断言，但当断言失败时显示特定信息。

最后要指出的是，虽然 assert 是 JDK 1.4 新增的关键字，但有一点非常重要：Java 命令默认不启动断言。

为了启动用户断言，应该在运行 java 命令时增加-ea（Enable Assert）选项。

为了启动系统断言，应该在运行 java 命令时增加-esa（Enable System Assert）选项。

32．Java 中会存在内存泄漏吗？请简单描述。

为了搞清楚 Java 程序是否有内存泄漏存在，首先要了解什么是内存泄漏。在程序运行过程中会不断地分配内存空间；那些不再使用的内存空间应该即时回收它们，从而保证系统可以再次使用这些内存。如果存在无用的内存没有被回收回来，那就是内存泄漏。

对于 Java 程序而言，只要 Java 对象一直处于可达状态，垃圾回收机制就不会回收它们——即使它们对于程序来说已经变成了垃圾（程序再也不需要它们了）；但对于垃圾回收机制来说，它们还不是垃圾（还处于可达状态），因此不能回收。

看 ArrayList 中 remove(int index)方法的源代码，程序如下：

```
public E remove(int index)
{
    // 检查 index 索引是否越界
    RangeCheck(index);
    // 使修改次数加 1
    modCount++;
    // 获取被删除的元素
    E oldValue = (E)elementData[index];
    int numMoved = size - index - 1;
    // 整体搬家
    if (numMoved > 0)
        System.arraycopy(elementData, index+1
            , elementData, index, numMoved);
    //将 ArrayList 的 size 减 1
```

```
        // 并将最后一个数组赋为 null，让垃圾回收机制回收最后一个元素
        elementData[--size] = null;
        return oldValue;
    }
```

上面程序中的粗体字代码 elementData[--size] = null;就是为了避免垃圾回收机制而编写的，如果没有这行代码，这个方法就会产生内存泄漏——每删除一个对象，但该对象所占用的内存空间却不会释放。

33．能不能自己写个类，也叫 java.lang.String？

可以，但在应用的时候，需要用自己的类加载器去加载；否则，系统的类加载器永远只是去加载 rt.jar 包中的那个 java.lang.String。

但在 Tomcat 的 Web 应用程序中，都是由 webapp 自己的类加载器先自己加载 WEB-INF/classess 目录中的类，然后才委托上级的类加载器加载，如果我们在 Tomcat 的 Web 应用程序中写一个 java.lang.String，这时候 Servlet 程序加载的就是我们自己写的 java.lang.String，但是这么做就会出现很多潜在的问题，原来所有用了 java.lang.String 类的都将出现问题。

34．ArrayList 如何实现插入的数据按自定义的方式有序存放？

编程思路是：实现一个类对 ArrayList 进行包装，当程序试图向 ArrayList 中放入数据时，程序将先检查该元素与 ArrayList 集合中其他元素的大小，然后将该元素插入到指定位置。

```
class MyBean implements Comparable{
    public int compareTo(Object obj){
        if(! obj instanceof MyBean)
            throw new ClassCastException()。
        MyBean other = (MyBean) obj;
        return age > other.age?1:age== other.age?0:-1;
    }
}
class MyTreeSet {
    private ArrayList  datas = new ArrayList();

    public void add(Object obj){
        for(int i=0;i<datas.size();i++){
            if(obj.compareTo(datas.get(i) != 1){
                datas.add(i,obj);
            }
        }
    }
}
```

35．序列化接口的版本号（id）有什么用？

反序列化 Java 对象时必须提供该对象的 class 文件，现在的问题是随着项目的升级，系统的 class 文件也会升级，Java 如何保证两个 class 文件的兼容性？

Java 序列化机制允许为序列化类提供一个 private static final long 类型的 serialVersionUID 值，该 Field 值用于标识该 Java 类的序列化版本。也就是说，如果一个类升级后，只要它的 serialVersionUID 值保持不变，序列化机制也会把它们当成同一个序列化版本。

通常建议程序员为序列化类的 serialVersionUID 指定值！如果程序员没有为序列化类的 serialVersionUID 指定值，系统会为该序列化类的 serialVersionUID 自动分配一个值。无论程序员对该类进行了怎样的修改（即使该修改对序列化没有任何影响），系统也会自动修改 serialVersionUID 值；如果程序员主动为序列化类的 serialVersionUID 分配值，则可以控制只有对该类的修改影响了序列化机制才去修改 serialVersionUID 值。

36．hashCode()方法的作用是什么？

hashCode()方法与 equals()方法相似，都是来自 java.lang.Object 类的方法，都允许用户定义的子类重写这两个方法。

一般来说，equals()方法是给用户调用的，如果你想根据自己的业务规则来判断两个对象是否相等，你可以重写 equals()方法。简单来讲，equals()方法主要是用来判断从表面上看或者从内容上看，两个对象是不是相等。

而 hashCode()方法通常是给其他类来调用的，比如当我们要把两个对象放入 HashSet 时，由于 HashSet 要求两个对象不能相等，而 HashSet 判断两个对象是否相等的标准是通过 equals()比较返回 false，或者两个对象的 hashCode()方法的返回值不相等——只要满足任意一个条件都可认为两个对象不相等。

从这个角度来看，我们可以把 hashCode()方法的返回值当成这个对象的“标识符”，如果两个对象的 hashCode()相等，即可认为这两个对象是相等的。因此，当我们重写一个类的 equals()方法时，也应该重写它的 hashCode()方法，而且这两个方法判断两个对象相等的标准也应该是一样的。

37．编写一个函数将一个十六进制数的字符串参数转换成整数返回。

```
String str = "13abf";
int len = str.length();
int sum = 0;
for(int i = 0 ; i < len ; i++)
{
   char c = str.charAt(len - 1 - i);
   int n = Character.digit(c ,16);
   sum += n * (1 << (4 * i));
}
```

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

其实，也可以用 Integer.parseInt(str,16)，但面试官很可能是想考我们的编码基本功。

38．银行还款问题。

银行贷款的还款方式中最常用的是“等额本息”还款法，即借款人在约定还款期限内的每一期（月）归还的金额（产生的利息+部分本金）都是相等的。现有一笔总额为 T 元的 N 年期住房贷款，年利率为 R，要求算出每一期还款的本金和利息总额。请写出解决思路和任意一种编程语言实现的主要代码。

思路：既然是按月还款，那么就要将 N 年按月来计算，即要还 N*12 个月，这样就可以求出每月要还的本金。由于每月要还的那部分本金所欠的时间不同，所以，它们所产生的利息是不同的，该部分本金的利息为：部分本金额*所欠月数*月利率。应该是这样的算法，如果利息还计利息，如果月还款不按年利率来算，那么老百姓是算不明白的。

```
int monthMoney = T/N/12;
float monthRate = R/12;
int totalMonth = N * 12;
float totalRate = 0;
for(int i = 1 ; i <= totalMonth ; i++)
{
    totalRate += monthMoney * monthRate * i;
}
int result = monthMoney + totalRate/N/12;
```

39．假设给定任意字符序列“123456”之类，输出它们所有的排列组合。

```
String str = "fkjavx";
char[] arr = str.toCharArray();
String[] result = {""};
for (int i = 0 ; i < arr.length ; i++ )
{
    String[] tmp = new String[result.length * (arr.length - i)];
    int counter = 0;
    for (int j = 0 ; j < result.length; j++ )
    {
        for (int k = 0 ; k < arr.length ; k++ )
        {
            System.out.println(j + " ----" + result[j]);
            if(!result[j].contains(arr[k] + ""))
            {
                tmp[counter++] = result[j] + arr[k];
            }
```

```
            }
        }
        result = tmp;
        System.out.println(java.util.Arrays.toString(result));
    }
    System.out.println(java.util.Arrays.toString(result));
```

40．构造器 Constructor 是否可被 Override？

构造器 Constructor 不能被继承，因此不能重写（Override），但可以被重载（Overload）。

41．接口是否可继承接口？抽象类是否可实现（implements）接口？抽象类是否可继承具体类（concrete class）？抽象类中是否可以有静态的 main 方法？

接口可以继承接口。抽象类可以实现（implements）接口，抽象类也可以继承具体类。抽象类中可以有静态的 main 方法。

只要记住《疯狂 Java 讲义》中的归纳：抽象类的特征是有得有失，得到的功能是抽象类可以拥有抽象方法（当然也可以没有）；失去的功能是抽象类不能创建实例。至于其他的，抽象类与普通类在语法上大致是一样的。

42．写 clone()方法时，通常都有一行代码，是什么？

clone()有默认行为：super.clone();，因为首先要把父类中的成员复制到位，然后才是复制自己的成员。

43．abstract class 和 interface 有什么区别？

含有 abstract 修饰符的 class 即为抽象类，abstract 类不能创建实例对象。含有 abstract 方法的类必须定义为 abstract class，abstract class 类中的方法不必是抽象的。abstract class 类中定义抽象方法必须在具体（Concrete）子类中实现，所以不能有抽象构造方法或抽象静态方法。如果子类没有实现抽象父类中的所有抽象方法，那么子类也必须定义为 abstract 类型。

接口（interface）可以说成是抽象类的一种特例，接口中的所有方法都必须是抽象的。接口中的方法定义默认为 public abstract 类型，接口中的成员变量类型默认为 public static final。

需要说明的是，Java 8 增强后的接口可以定义默认方法（使用 default 修饰的方法）和类方法（使用 static 修饰的方法），接口中的默认方法和类方法都不再是抽象方法，都需要提供方法体。

下面比较一下两者的语法区别。

（1）抽象类可以有构造器，接口中不能有构造器。

（2）抽象类中可以有普通成员变量，接口中没有普通成员变量。

（3）抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。Java 8 增强的接口可拥有默认方法和类方法，接口中的默认方法和类方法都不再是抽象方法，都需要提供方法体。

（4）抽象类中的抽象方法的访问类型可以是 public、protected 和默认访问权限。但接口中的方法只能是 public 的。如果是普通实例方法，则必须是抽象方法；如果是默认方法，则必须使用 default 修饰；如果是类方法，则必须使用 static 修饰。

（5）抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是 public static final 类型，并且默认为 public static final 类型。

（6）一个类可以实现多个接口，但只能继承一个抽象类。

下面再说说两者在应用上的区别。

接口更多的是在系统架构设计方面发挥作用，接口体现的是一种规范。而抽象类在代码实现方面发挥作用，可以实现代码的重用。例如，模板模式是抽象类的一个典型应用，假设项目中需要使用大量的 DAO 组件，这些 DAO 组件通常都具有增、删、改、查等基本方法，因此我们就可以定义一个抽象的 DAO 基类，然后让其他 DAO 组件来继承这个 DAO 基类，把这个 DAO 基类当成模板使用。

44．abstract 的 method 是否可同时是 static？是否可同时是 native？是否可同时是 synchronized？

abstract 的 method 不可以是 static 的，因为抽象的方法是要被子类实现的，而 static 与子类扯不上关系！

native 方法表示该方法是用另外一种依赖平台的编程语言实现的，不存在被子类实现的问题，所以它也不能是抽象的，不能与 abstract 混用。

关于 synchronized 与 abstract 不能同时使用。因为 synchronized 修饰一个方法时，表明将会使用该方法的调用者作为同步监视器，但对于一个 abstract 方法而言，它所在类是一个抽象类，抽象类也无法创建实例，因此也就无法确定 synchronized 修饰方法时的同步监视器了，所以 synchronized 与 abstract 不能同时使用。

45．什么是内部类？ Static Nested Class 和 Inner Class 有什么不同？

内部类就是在一个类的内部定义的类，在非静态内部类中不能定义静态成员。静态内部类不能访问外部类的静态成员。

内部类作为其外部类的一个成员，因此内部类可以直接访问外部类的成员。但有一点需要指出：静态成员不能访问非静态成员，因此静态内部类不能访问外部类的非静态成员。

如果内部类使用了 static 修饰，那么这个内部类就是静态内部类，也就是所谓的 static Nested Class；如果内部类没有使用 static 修饰，那么它就是 Inner Class。除此之外，还有一种局部内部类：在方法中定义的内部类就是局部内部类，局部内部类只在方法中有效。

对于 Static Nested Class 来说，它使用了 static 修饰，因此它属于类成员，Static Nested Class

的实例只要寄生在外部类中即可。因此，使用 Static Nested Class 十分方便，开发者可以把外部类当成 Static Nested Class 的一个包。

对于 Inner Class 而言，它属于实例成员，因此 Inner Class 的实例必须寄生在外部类的实例中，所以程序在创建 Inner Class 实例之前，必须先获得一个它所寄生的外部类的实例；否则，程序无法创建 Inner Class 实例。例如如下代码：

```
class Outer
{
    class Inner
    {
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Outer.Inner inner;
        Outer outer = new Outer();
        // 必须先获得外部类的实例，然后才能调用构造器
        inner = outer.new Inner();
    }
}
```

46．内部类可以引用它的外部类的成员吗？有没有什么限制？

内部类可以访问所在外部类的成员。

但有一点需要注意：静态成员不能访问非静态成员，因此静态内部类（属于静态成员）就不能访问外部类的非静态成员。

47．Anonymous Inner Class（匿名内部类）是否可以 extends（继承）其他类？是否可以 implements（实现）interface（接口）？

匿名内部类必须显式继承某个父类或实现某个接口，但匿名内部类只能显式继承某个父类或实现某个接口。下面是匿名内部类特殊的语法。

```
new 父类|父接口()
{
    类体实现部分
}
```

从上面语法不难看出，匿名内部类必须继承其他类或实现其他接口。

对于使用函数式接口（只包含一个抽象方法的接口），Java 8 提供了简洁的 Lambda 表达式来创建对象。对于只包含一个抽象方法的函数式接口而言，可使用如下代码来创建实现该接口的对象：

```
(被实现的抽象方法的形参列表) -> {
    实现抽象方法的方法体;
}
```

从上面语法规范可以看出，Lambda 表达式就负责实现函数式接口中的抽象方法，系统根据 Lambda 表达式来创建实现函数式接口的对象。

48．super.getClass()方法调用。

下面程序的输出结果是多少？

```
import java.util.Date;
public class Test extends Date{
    public static void main(String[] args) {
        new Test().test();
    }
    public void test(){
        System.out.println(super.getClass().getName());
    }
}
```

程序输出的是 Test。

《疯狂 Java 讲义》（第 2 版）中有关于 super 关键字很透彻的解释：super 只是一个限定词，当用 super 引用时，它也是引用当前对象本身，super 只是限定了访问当前对象从父类那里继承得到成员变量或方法。

如果需要访问父类的类名，应该使用如下语法：

```
super.getClass().getSuperclass().getName()
```

49．JDK 中哪些类是不能继承的？

使用 final 修饰的类都不可以被继承。

实际上即使是自己开发的类，也可以通过使用 final 修饰来阻止被继承。使用 final 修饰的类被称为最终类，最终类不能派生子类，这样该类就被完全封闭起来了，不会有子类来重写它的方法，因此更加安全。

50．String s = "Hello";s = s + " world!";这两行代码执行后，原始的 String 对象中的内容到底变了没有？

没有。因为 String 被设计成不可变类（immutable），所以它的所有对象都是不可变对象。在上面两行代码中，s 原先指向一个 String 对象，内容是"Hello"，然后我们对 s 进行了+操作，那么 s 所指向的那个对象是否发生了改变呢？答案是没有。此时 s 不指向原来那个对象

了，而是指向了另一个 String 对象，内容为"Hello world!"，原来那个对象还存在于内存之中，只是 s 这个引用变量不再指向它了。

通过上面的说明，我们很容易得出一个结论：如果经常对字符串进行各种各样的修改，那么使用 String 来代表字符串则会引起很大的内存开销。因为 String 对象建立之后不能再改变，所以对于每一个不同的字符串都需要一个 String 对象来表示。这时，应该考虑使用 StringBuffer 类，它允许修改，而不是每个不同的字符串都要生成一个新的对象。并且，这两种类的对象转换十分容易。

实际上，当我们需要一个字符串对象时，应该使用如下语法来创建 String 对象：

```
Sring s = "fkjava.org";
```

也就是直接使用字符串直接量的语法。而不是：

```
String s = new String("fkjava.org");
```

对于第二种语法而言，每次都会调用构造器生成新的 String 对象，性能低下且内存开销大，并且没有意义。因为 String 对象不可改变，所以对于内容相同的字符串，只要一个 String 对象来表示就可以了。

基于这样一种想法，Java 提供了字符串缓存池来管理字符串直接量，当程序多次用到同一个字符串直接量时，系统会让它们都引用字符串缓存池中的同一个 String 对象。因此，在程序中使用字符串直接量可以充分利用这个特性来降低系统内存开销，提高程序性能。

51．是否可以继承 String 类？

String 类是 final 类，不可以被继承。

52．如何把一段用逗号分割的字符串转换成一个数组？

A. 以前，Java 提供了一个 StingTokenizer 工具类来处理字符串分割的问题。比如使用如下语法：

```
StringTokenizer st = new StringTokenizer("this,is,a,test" , ",");
while (st.hasMoreTokens())
{
    System.out.println(st.nextToken());
}
```

这样程序将会输出

```
this
is
a
test
```

B. 后来，Java 为 String 类增加了正则表达式支持，StingTokenizer 基本上没用了。因此，上面代码可以简写为：

```
String [] result = "this,is,a,test".split(",");
```

其中 result 数组中就存放了 this、is、a、test 等字符串元素。

53．下面这条语句一共创建了多少个对象？String s="a"+"b"+"c"+"d";

答：对于如下代码：

```
String s1 = "a";
String s2 = s1 + "b";
String s3 = "a" + "b";
System.out.println(s2 == "ab");
System.out.println(s3 == "ab");
```

第一条语句打印的结果为 false，第二条语句打印的结果为 true。

Java 会在编译时对字符串相加进行优化处理，如果整个表达式中所有参与运算的都是字符串直接量，Java 会在编译时就把这个表达式的值计算出来，然后直接将结果赋值给字符串引用变量。因此，上面题目中定义的 String s = "a" + "b" + "c" + "d";实际上相当于直接定义了"abcd"字符串直接量，因此上面的代码只创建了一个 String 对象。

而且这个字符串直接量会被放入字符串缓存池中。如下两行代码：

```
String s = "a" + "b" + "c" + "d";
System.out.println(s == "abcd");
```

由于 s 引用了字符串缓存池中的"abcd"字符串，因此上面代码的输出结果应该为 true。

54．Collection 框架中实现比较要实现什么接口？

Java 集合框架中需要比较大小的集合包括 TreeMap、TreeSet，其中 TreeMap 会根据 key-value 对中 key 的大小进行排序，而 TreeSet 则会对集合元素进行排序。

因此 TreeMap 的 key、TreeSet 的集合元素，都需要比较大小。集合框架中比较大小有两种方式：

A．自然排序：对于自然排序来说，要求 TreeMap 中的所有 key 都实现 Comparable 接口，实现该接口时需要实现一个 int compareTo(T o)方法，用于判断当前对象与 o 对象之间的大小关系。如果该方法返回正整数，则说明当前对象大于被比较的 o 对象；如果该方法返回 0，则说明两个对象相等；如果该方法返回负整数，则说明当前对象小于被比较的 o 对象。JDK 的很多类都已经实现了 Comparable 接口，例如 String、Date、BigDecimal 等。

B．定制排序：定制排序需要在创建 TreeMap 或 TreeSet 时传入一个 Comparator 对象，此时 TreeMap 或 TreeSet 不再要求 key，集合元素本身是可比较大小的，而是由 Comparator 来负责比较集合元素的大小。Comparator 本身只是一个接口，因此创建 Comparator 对象只能是创建它的实现类的对象，Comparator 的实现类需要实现 int compare(T o1, T o2)方法，该方法用于判断 o1、o2 两个对象的大小，如果该方法返回正整数，则说明 o1 大于 o2；如果该方法返回负整数，则说明 o1 小于 o2；如果返回 0，则说明两个对象相等。

55．ArrayList 和 Vector 的区别是什么？

这两个类都实现了 List 接口（List 接口继承了 Collection 接口），它们都是有序集合，即存储在这两个集合中的元素的位置都是有顺序的，相当于一种动态的数组，以后可以按位置

索引号取出某个元素，并且其中的数据是允许重复的——这是由 List 集合规范制订的。

ArrayList 与 Vector 底层都是基于数组的，因此它们的实现代码也大致相似。区别在于 Vector 是一个古老的集合，从 JDK 1.0 开始就有了，因此它包含了大量方法名很长的方法，从 JDK 1.2 开始引入集合框架，引入 List 接口，才让 Vector 实现了 List 接口，因此又增加了一些 List 接口中定义的方法。总体来说，ArrayList 可以完全代替 Vector，除了在一些很古老的 API 中强制要求使用 Vector 之外。

Vector 还有一个特征：它是线程安全的，因此性能比较差。而 ArrayList 并不是线程安全的，因此性能较好。实际上即使需要在多线程环境下使用 List 集合，也应该选择 ArrayList，而不是 Vector，因为 Java 还提供了一个 Collections 工具类，它可以把 ArrayList 包装成线程安全的集合类，例如如下代码：

```
List list = Collections.synchronizedList(new ArrayList());
```

56．HashMap 和 Hashtable 的区别是什么？

HashMap 与 Hashtable 的区别类似于 ArrayList 与 Vector 的区别。

Hashtable 与 Vector 都是从 JDK 1.0 开始就有的古老的集合，因此 Hashtable 是一个继承自 Dictionary 的古老集合。

从 JDK 1.2 引入集合框架的 Map 接口之后，Java 让 Hashtable 也实现了 Map 接口，因此 Hashtable 也新增实现了一些 Map 接口中定义的方法。实际上 Hashtable 与 HashMap 底层的实现很相似，它们都是基于 Hash 表的实现。

HashMap 与 Hashtable 的区别主要有如下两点：

A．HashMap 允许使用 null 作为 key 或 value，而 Hashtable 不允许。

B．HashMap 是线程不安全的，因此性能较好；但 Hashtable 是线程安全的，因此性能较差。

实际上，即使在多线程环境下，Java 提供了 Collections 工具类把 HashMap 包装成线程安全的类，因此依然应该使用 HashMap，如下代码所示：

```
Map map = Collections. synchronizedMap(new HashMap());
```

简单地说，编程时应该尽量避免使用 Hashtable，除非在一个古老的 API 中强制要求使用 Hashtable。

57．List 和 Map 区别是什么？

从表面来看，List 是一个只是存放单个元素的集合，List 集合所包含的元素可以重复，元素按放入的先后顺序来存放，程序可以通过元素的索引来读取元素，因此 List 相当于一个动态数组；Map 则是一个存放 key-value 对的集合，Map 里存放的 key-value 对是无序的，Map 包含的 key 是不允许重复的，程序可以 key 来取出该 key 对应的 value。

深入阐述：如果换个角度来看，完全可以把 List 当成 Map 来看，List 相当于一个 key 都是 int 类型的 Map，程序通过元素的索引（相当于通过 int 类型的 key）来读取 List 集合的元素时，完全也可以当成 Map 根据 key 来读取 value。从另一个角度来看，Map 也可以当成元素索引可以是任意类型的 List 集合。

58．List、Set、Map 是否继承自 Collection 接口？

List、Set 是，Map 不是。

59．List、Map、Set 三个接口存取元素时，各有什么特点？

Set 集合是最接近 Collection 的集合，因此 Set 集合几乎没有在 Collection 上增加什么方法。Set 集合代表了元素无序、元素不允许重复的集合（Set 只是在 Collection 规范上增加了元素不允许重复的约束）。

List 集合则在 Collection 的基础上为元素增加了索引的特性，因此 List 集合代表了集合元素有序、集合元素可以重复的集合。

Map 则代表了存放 key-value 对的集合，程序可以通过 key 来获取其中的 value。

就 Set 集合来说，对于开发者而言，它的集合元素是无序的，似乎显得有些杂乱、无规律，但对计算机而言这不可能，因此计算机需要快速存、取 Set 集合中的元素。Set 集合有两个实现类：HashSet 与 TreeSet，其中 HashSet 底层其实使用了一个数组来存放所有的集合元素，然后通过 Hash 算法来决定每个集合元素在底层数组中的存放位置，因此 HashSet 对集合元素的存、取就是 Hash 算法+数组存、取——也就是说，HashSet 只比数组存、取多了些 Hash 算法开销，因此性能非常好。TreeSet 底层则完全是一个红黑树，因此红黑树是折中平衡的排序二叉树，它底层没有数组开销，存、取元素时都是基于红黑树算法的，因此内存开销较小，但性能略差。

对于 List 集合而言，主要有两个实现类：ArrayList 与 LinkedList，其中 ArrayList 底层是基于数组的，而且 ArrayList 存、取元素本身就是通过元素索引来进行的，因此 ArrayList 对元素的存、取性能非常好，几乎等同于存、取数组元素。但是添加、删除元素时需要对数组元素进行“整体搬家”，因此添加、删除元素时性能较差。而 LinkedList 底层则是基于一个链表实现的，当从链表中存、取元素时，需要定位元素的位置，系统开销较大。但添加、删除元素时，只要修改元素的引用（相当于指针）即可，因此性能非常好。

对于 Map 集合而言，其底层存、取性能与 Set 集合完全一样。其实 Set 集合本身就是基于 Map 实现的——如果我们把 Map 集合的所有 value 都当成空对象处理，只考虑 Map 集合的 key，Map 集合就变成了 Set 集合。换个角度来看，如果我们向 Set 集合中添加的对象是 key-value 所组成的 Entry 对象，那么 Set 集合也就变成了 Map 集合。

60．说出 ArrayList、Vector、LinkedList 的存储性能和特性。

ArrayList 和 Vector 都是使用数组方式存储数据的，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 由于使用了 synchronized 方法（线程安全），通常性能上较 ArrayList 差，而 LinkedList 使用双向链表实现存储，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。

LinkedList 也是线程不安全的，LinkedList 提供了一些方法，使得 LinkedList 可以被当作栈和队列来使用。

实际上 Java 提供了 Collections 工具类，它可以把 ArrayList、LinkedList 包装成线程安全的集合，因此在实际编程中应该避免使用 Vector。

61．去掉一个 Vector 集合中重复的元素。

```
Vector newVector = new Vector();
for (int i = 0 ; i < vector.size() ; i++)
{
    Object obj = vector.get(i);
    if(!newVector.contains(obj))
    {
        newVector.add(obj);
    }
}
```

另外，还有一种更加简单的方式：将 Vector 添加到 HashSet 中，例如如下代码：

```
HashSet set = new HashSet(vector);
```

但上面代码将会导致 Vector 中元素丢失顺序。

62．Set 里的元素是不能重复的，那么用什么方法来区分重复与否呢？是用==还是 equals()？它们有何区别？

说明：其实这个题目本身有问题！因为 Set 只是一个接口，它的不同实现类判断元素是否相等的标准是不同的。笼统地说，Set 里的元素是不能重复的，判断元素重复使用 equals()，而不是==。

对于 HashSet 而言，判断两个对象是否相等是通过 equals()和 hashCode()方法，只要两个对象通过 equals()比较返回 false，或两个对象的 hashCode()不相等，那么 HashSet 就会把它们当成不相同。

对于 TreeSet 而言，判断两个对象相等的唯一标准是：两个对象通过 compareTo(Object obj)比较是否返回 0，与 equals()方法无关。只要两个对象通过 compareTo(Object obj)比较没有返回 0，Java 就会把它们当成两个对象处理——这一点是很多人容易误解的，不过我们可以通过《疯狂 Java 讲义》中的一个示例来说明：

```
class Z implements Comparable
{
    int age;
    public Z(int age)
    {
        this.age = age;
    }
    // 重写 equals()方法，总是返回 true
    public boolean equals(Object obj)
    {
        return true;
    }
    // 重写了 compareTo(Object obj)方法，总是返回正整数
    public int compareTo(Object obj)
```

```
    {
        return 1;
    }
}
public class TreeSetTest2
{
    public static void main(String[] args)
    {
        TreeSet set = new TreeSet();
        Z z1 = new Z(6);
        set.add(z1);
        // 输出 true，表明添加成功
        System.out.println(set.add(z1)); // ①
        // 下面输出 set 集合，将看到有两个元素
        System.out.println(set);
        // 修改 set 集合的第一个元素的 age 变量
         ((Z)(set.first())).age = 9;
        //输 出 set 集合的最后一个元素的 age 变量，将看到也变成了 9
        System.out.println(((Z)(set.last())).age);
    }
}
```

上面程序中两个 Z 对象通过 equals()比较总会返回 true，但通过 compareTo(Object obj)比较总是不会返回 0，因此两次向 TreeSet 中添加同一个元素，TreeSet 会把它们当成不同的对象进行处理，最后 TreeSet 集合中会显示有两个对象，但实际上是同一个对象。

63．你所知道的集合类都有哪些？主要方法有哪些？

最常用的集合接口是 Set、List、Queue，它们都是 Collection 的子接口，除此之外，还有 Map 接口。

对于 Set 集合而言，它的常用实现类包括 HashSet 与 TreeSet。HashSet 还有一个子类：LinkedHashSet。

对于 List 集合而言，它的常用实现类包括 ArrayList、Vector 与 LinkedList。

对于 Queue 集合而言，它有一个子接口 Deque（代表双端队列），它的常用实现类包括 ArrayDeque 与 LinkedList。

对于 Map 集合而言，它的常用实现类是 HashMap 与 TreeMap。HashMap 还有一个子类：LinkedHashMap。

至于这些集合的方法，由于集合类也就是所谓的“容器类”，因此它的方法无非就是向容器中添加、删除、取出、遍历元素的方法。

对于 List 集合而言，由于它的集合元素都是有序的、有索引的，因此它包括了大量根据索引来添加、删除、取出集合元素的方法。

对于 Deque 集合而言，由于它是双端队列，既可当成队列使用，也可当成栈使用，因此它增加栈、队列的方法，如 offer、peek、push、pop 等。

对 Map 而言，它所包含的无非就是根据 key 来添加、删除、取出 value 的方法。

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

对。因为 equals()方法可以由开发者重写，hashCode()方法也可以由开发者来重写，因此它们是否相等并没有必然的关系。

一般来说，如果对象要保存在 HashSet 中或作为 HashMap 的 key，它们通过 equals()比较相等，那么它们的 hashCode()返回值也应该相等。

65．TreeSet 里面放对象，如果同时放入了父类和子类的实例对象，那么比较时使用的是父类的 compareTo()方法，还是子类的 compareTo()方法，抑或是抛出异常？

根据 TreeSet 底层的实现：TreeSet 底层的实现就是红黑树，因此当程序向 TreeSet 中添加集合元素时，程序会多次调用该对象的 compareTo()方法与 TreeSet 中的集合元素进行比较，直到找到该元素在红黑树中所在节点的位置。因此，该问题的答案是：当前正在添加父类对象就多次调用父类对象的 compareTo()方法；当前正在添加子类对象就多次调用子类对象的 compareTo()方法。

至于程序是否抛出异常，则取决于 compareTo()方法的实现，如果子类在实现 compareTo()方法时，试图把被比较对象转换为子类对象之后再进行比较——如果 TreeSet 集合中已经包括了父类对象，这就会引起 ClassCastException 异常。

示例代码如下：

```
class A implements Comparable
{
    int age;
    public A(int age)
    {
        this.age = age;
    }

    public int compareTo(Object obj)
    {
        System.out.println("AAAAAAAAAA");
        A target = (A)obj;
        return age > target.age ? 1 : age < target.age ? -1 : 0;
    }
    public String toString()
    {
        return getClass() + ",age:" + age;
    }
```

```
}
class B extends A implements Comparable
{
    public B(int age)
    {
        super(age);
    }
    public int compareTo(Object obj)
    {
        System.out.println("BBBBBBBBB");
        A target = (A)obj;
        return age > target.age ? 1 : age < target.age ? -1 : 0;
    }
}
public class TreeSetTest2
{
    public static void main(String[] args)
    {
        TreeSet set = new TreeSet();
        set.add(new A(3));
        set.add(new B(1));
        set.add(new A(2));
        for(Iterator it = set.iterator(); it.hasNext() ;)
        {
            System.out.println(it.next());
        }
    }
}
```

上面程序的输出如下：

```
AAAAAAAAAA
BBBBBBBBB
AAAAAAAAAA
AAAAAAAAAA
```

第一次添加 A 对象，所以调用 A 对象的 compareTo()方法；第二次添加 B 对象，所以程序调用了 B 对象的 compareTo()方法；第三次再次添加 A 对象，由于集合中已经有两个对象，因此程序两次调用了 A 对象的 compareTo()方法与集合中的元素进行比较。

66．说出一些常用的类、包、接口，请各举 5 个。

常用的包有：

java.lang 包下包括 Math、System、StringBuilder、StringBuffer、Runtime、Thread、Runnable 等。

java.util 包下包括 List、Set、Map，以及这些接口的常用实现类：ArrayList、LinkedList、HashSet、TreeSet、HashMap、TreeMap 等。

java.io 包下包括 InputStream、OutputStream、Reader、Writer、FileInputStream、FileOutputStream、FileReader、FileWriter、BufferedInputStream、BufferedOutputStream、BufferedReader、BufferedWriter 等

java.sql 包下包括 Connection、Statement、PreparedStatement、ResultSet 等。

java.net 包下包括 Socket、ServerSocket、URL、URLConnection、DatagramPacket、DatagramSocket 等。

如果为了让别人感觉你对 Android 很熟悉，还应该增加一些 Android 常用的包、类。例如：

android.app 包下有 Activity、ListActivty、TabActivity、AlertDialog、AlertDialog.Builder、Notification、Service 等。

android.content 包下有 ContentProvider、ContentResolver、ContentUris、ContentValues、Context 等

android.database 包下有 Cursor 等

android.database.sqlite 包下有 SQLiteDatabase、SQLiteOpenHelper 等

android.graphics 包下有 Bitmap、BitmapFactory、Canvas、Color、Matrix、Paint、Path 等。

android.widget 包下有 TextView、Button、CheckBox、RadioButton、ListView、GridView、Spinner、Gallery 等。

67．Java 中有几种类型的流？JDK 为每种类型的流提供了一些抽象类以供继承，请说出它们分别是哪些类？

字节流和字符流。字节流由 InputStream、OutputStream 派生出来，字符流由 Reader、Writer 派生出来。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。

Java 的 IO 流也分为节点流和过滤流，其中节点流是直接关联到物理 IO 节点的流，而过滤流则需要建立在其他流的基础之上。Java 为过滤流提供了 FilterInputStream、FilterOutputStream、FilterReader、FilterWriter 这 4 个基类。

一般来说，建议读者按《疯狂 Java 讲义精粹》的 11.4 节（或《疯狂 Java 讲义》的 15.4 节）中关于 IO 流体系的表格进行回答。

68．字节流与字符流的区别是什么？

字节流和字符流的区别非常简单，它们的用法几乎完全一样，区别在于字节流和字符流所操作的数据单元不同：字节流操作的数据单元是 8 位的字节，而字符流操作的数据单元是 16 位的字符。

字节流主要由 InputStream 和 OutputStream 作为基类，而字符流则主要由 Reader 和 Writer 作为基类。

字节流是基于字节直接进行输入、输出的，因此它的适用性更广。字符流则在处理文本内容的输入、输出时更加方便——不会出现读取半个字符的情形。

Java 提供了将字节流转换为字符串的 InputStreamReader 和 OutputStreamWriter，但没有提供将字符流转换为字节流的方法。因为：字节流比字符流的使用范围更广，但字符流比字节流操作方便。如果有一个流已经是字符流了，也就是说，是一个用起来更方便的流，为什

么要转换成字节流呢？反之，如果现在有一个字节流，但我们知道这个字节流的内容都是文本内容，那么把它转换成字符流来处理就会更方便一些，所以 Java 只提供了将字节流转换为字符流的转换流，没有提供将字符流转换成字节流的转换流。

69．什么是 Java 序列化？如何实现 Java 序列化？请解释 Serializable 接口的作用。

Java 序列化的目标是将对象保存到磁盘中，或者允许在网络中直接传输对象，对象序列化机制允许把内存中的 Java 对象转换成平台无关的二进制流，从而允许把这种二进制流持久保存在磁盘上，通过网络将这种二进制流传输到另一个网络节点。其他程序一旦获得了这种二进制流（无论是从磁盘中获取，还是通过网络获取），都可以将这种二进制流恢复成原来的 Java 对象。

实现 Java 序列化有两种方式：

A．让 Java 类实现 Serializable 接口。

B．让 Java 类实现 Externalizable 接口，实现该接口时还必须实现 readExternal()、writeExternal()这两个方法。

一旦 Java 类实现了上面两种接口，接下来程序中就可通过 ObjectInputStream、ObjectOutputStream 来读取、保存 Java 对象了。

Serializable 接口只是一个标记接口，实现该接口无须实现任何方法，实现了该接口的类就是可序列化的类。

序列化对于 Java 开发非常重要，例如在 Web 开发中，如果对象需要保存在 Session 中，Tomcat 在某些时候需要把 Session 中的对象序列化到硬盘，因此放入 Session 中的对象必须是可序列化的，要么实现 Serializable 接口，要么实现 Externalizable 接口。还有，如果一个对象要经过网络传输（比如 RMI 远程方法调用的形参或返回值），这个对象也应该是可序列化的。

70．描述一下 JVM 加载 class 文件的原理机制。

当程序主动使用某个类时，如果该类还未被加载到内存中，系统会通过加载、连接、初始化三个步骤来对该类进行初始化，如果没有意外，JVM 将会连续完成这三个步骤，所以有时也把这三个步骤统称为类加载或类初始化。

类加载指的是将类的 class 文件读入内存，并为之创建一个 java.lang.Class 对象。也就是说，当程序中使用任何类时，系统都会为之建立一个 java.lang.Class 对象。

类的加载由类加载器完成，类加载器通常由 JVM 提供，JVM 提供的这些类加载器通常被称为系统类加载器。除此之外，开发者可以通过继承 ClassLoader 基类来创建自己的类加载器。

通过使用不同的类加载器，可以从不同来源加载类的二进制数据，通常有如下几种来源：

A．从本地文件系统来加载 class 文件，绝大部分自己写的 Java 类的 class 文件都采用这种加载方式。

B．从 JAR 包中加载 class 文件，这种方式也是很常见的，比如 JDBC 编程时用到的数据库驱动类就是放在 JAR 文件中的，JVM 可以从 JAR 文件中直接加载该 class 文件。

C．通过网络加载 class 文件。

D．把一个 Java 源文件动态编译，并执行加载。

当 JVM 启动时，会形成由三个类加载器组成的初始类加载器层次结构：

（1）Bootstrap ClassLoader：根类加载器。

（2）Extension ClassLoader：扩展类加载器。

（3）System ClassLoader：系统类加载器。

Bootstrap ClassLoader，被称为引导（也称为原始或根）类加载器，它负责加载 Java 的核心类。在 Sun 的 JVM 中，当执行 java.exe 命令时使用-Xbootclasspath 选项或使用-D 选项指定 sun.boot.class.path 系统属性值可以指定加载附加的类。

Extension Classloader，被称为扩展类加载器，它负责加载 JRE 的扩展目录（%JAVA_HOME%/jre/lib/ext 或者由 java.ext.dirs 系统属性指定的目录）中 JAR 包的类。

通过这种方式，我们就可以为 Java 扩展核心类以外的新功能，只要我们把自己开发的类打包成 JAR 文件，然后放入%JAVA_HOME%/jre/lib/ext 路径即可（其中 JAVA_HOME 代表 JDK 安装路径）。

System Classloader，被称为系统（也称为应用）类加载器，它负责在 JVM 启动时加载来自 java 命令的-classpath 选项、java.class.path 系统属性，或 CLASSPATH 环境变量所指定的 JAR 包和类路径。程序可以通过 ClassLoader 的静态方法 getSystemClassLoader()获取系统类加载器。如果没有特别指定，则用户自定义的类加载器都以类加载器作为父加载器。

71．heap 和 stack 有什么区别？

stack 内存指的是程序进入一个方法时，系统会专门为这个方法分配一块内存空间，这块内存空间也被称为该方法栈区，该方法的栈区专门用于存储该方法中定义的局部变量，包括基本类型的变量和引用变量。当这个方法结束时，该方法栈区将会自动被销毁，栈区中的所有局部变量都会随之销毁。

heap 内存是 Java 虚拟机拥有的内存区，所有的 Java 对象都将被放在 heap 内存内，位于 heap 内存中的 Java 对象由系统的垃圾回收器负责跟踪管理——也就是进行垃圾回收，当堆内存中的 Java 对象没有引用变量引用它时，这个 Java 对象就变成了垃圾，垃圾回收器就会在合适的时候回收它。

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

肯定会执行。finally{}块的代码只有在 try{}块中包含遇到 System.exit(0);之类的导致 Java 虚拟机直接退出的语句才会不执行。

当程序执行 try{}遇到 return 时，程序会先执行 return 语句，但并不会立即返回——也就是把 return 语句要做的一切事情都准备好，即在将要返回但并未返回的时候，程序把执行流程转去执行 finally 块，当 finally 块执行完成后就直接返回刚才 return 语句已经准备好的结果。

例如如下程序：

```
public class Test
```

```
{
    public static void main(String[] args)
    {
        System.out.println(new Test().test());;
    }
    static int test()
    {
        int x = 1;
        try
        {
            return x;
        }
        finally
        {
            System.out.println("finally 块执行:" + ++x);
        }
    }
}
```

此时的输出结果为：

```
finally 块执行:2
1
```

看到上面程序中的 finally 块已经执行了，而且程序执行 finally 块时已经把 x 变量增加到 2 了。但 test()方法返回的依然是 1，这就是由 return 语句执行流程决定的，Java 会把 return 语句先执行完，把所有需要处理的东西都先处理完成，需要返回的值也都准备好之后，但是还未返回之前，程序流程会转去执行 finally 块，但此时 finally 块中对 x 变量的修改已经不会影响 return 要返回的值了。

但如果 finally 块里也包含 return 语句，那就另当别论了，因为 finally 块里的 return 语句也会导致方法返回，例如把程序该为如下形式：

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(new Test().test());;
    }
    static int test()
    {
        int x = 1;
        try
        {
            return x++;
        }
        finally
        {
            System.out.println("finally 块执行:" + ++x);
```

```
            return x;
        }
    }
}
```

此时的输出结果为：

```
finally 块执行:3
3
```

正如所介绍的，程序在执行 return x++;时，程序会把 return 语句执行完成，只是等待返回，此时 x 的值已经是 2 了，但程序此时准备的返回值依然是 1。接下来程序流程转去执行 finally 块，此时程序会再次对 x 自加，于是 x 变成了 3，而且由于 finally 块中也有 return x; 语句，因此程序将会直接由这条语句返回，因此上面 test()方法将会返回 3。

73．下面的程序代码输出的结果是多少？

```
public class smallT
{
    public static void main(String args[])
    {
        smallT t = new smallT();
        int b = t.get();
        System.out.println(b);
    }
    public int get()
    {
        try
        {
            return 1 ;
        }
        finally
        {
            return 2 ;
        }
    }
}
```

输出结果是：2。

这个程序还是刚才介绍的 return 语句和 finally 块的顺序问题。

Java 会把 return 语句先执行完，把所有需要处理的东西都先处理完成，需要返回的值也都准备好之后，但是还未返回之前，程序流程会转去执行 finally 块。但如果在执行 finally 块时遇到了 return 语句，程序将会直接使用 finally 块中的 return 语句返回——因此上面程序将会输出 2。

74．final、finally 和 finalize 的区别是什么？

final 是一个修饰符，它可以修饰类、方法、变量。

final 修饰类时表明这个类不可以被继承。

final 修饰方法时表明这个方法不可以被其子类重写。

final 修饰变量时可分为局部变量、实例变量和静态变量，当 final 修饰局部变量时，该局部变量可以被一次赋值，以后该变量的值不能发生改变；当 final 修饰实例变量时，实例变量必须由程序员在构造器、初始化块、定义时这三个位置的其中之一指定初始值；当 final 修饰静态变量时，静态变量必须由程序在静态初始化块、定义时这两个位置的其中之一指定初始值。

finally 是异常处理语句结构的一部分，表示总会执行的代码块。

finalize 是 Object 类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收。但实际上重写该方法进行资源回收并不安全，因为 JVM 并不保证该方法总被调用。

75．运行时异常与一般异常有何异同？

Checked 异常体现了 Java 的设计哲学：没有完善错误处理的代码根本就不会被执行！

对于 Checked 异常的处理方式有两种：

A．当前方法明确知道如何处理该异常，程序应该使用 try...catch 块来捕获该异常，然后在对应的 catch 块中修补该异常。

B．当前方法不知道如何处理这种异常，应该在定义该方法时声明抛出该异常。

运行时异常指的就是 RuntimeException 或其子类的实例，运行时异常则更加灵活，编译器不会强制要求程序员必须处理该异常，运行时异常可以既不显式声明抛出，也不使用 try...catch 进行捕获。但如果程序需要捕获运行时异常，也可以使用 try...catch 块来捕获运行时异常。

运行时异常的优点在于灵活：程序员想处理就处理，不想处理直接忽略该异常也行。但由于编译器不会强制检查运行时异常，因此程序完全有可能在运行时因为这些异常而结束。常见的运行时异常有 NullPointerException 、 ArrayIndexOutOfBoundsException 、ClassCastException、ArithmeticException、IllegalArgumentException 等。

76．Error 和 Exception 有什么区别？

Error 错误，一般是指虚拟机相关的问题，如系统崩溃、虚拟机出错误、动态链接失败等，这种错误无法恢复或不可能捕获，将导致应用程序中断。通常应用程序无法处理这些错误，因此应用程序不应该试图使用 catch 块来捕获 Error 对象。

由于编译器会对 Error 进行检查，不会强制要求程序员必须处理 Error，因此 Error 也被归入 unchecked 异常分类中（另外，运行时异常也属于 unchecked 异常）。

Exception 表示一种设计或实现问题。也就是说，程序员应该对这些情况进行考虑，并提供相应的处理。

77．Java 中的异常处理机制的简单原理和应用。

在程序运行过程中可能会出现各种“非预期”情况，这些非预期情况可能导致程序非正常结束。为了提高程序的健壮性，Java 提供了异常处理机制：

```
try
{
    s1...
    s2...
    s3...
}
catch(Exception ex)
{
    // 对异常情况的修复处理
}
```

对于上面的处理流程，当程序执行 try 块里的 s1、s2、s3 遇到异常时，Java 虚拟机将会把这个异常情况封装成异常对象，这个异常对象可以被后面对应的 catch 块捕获到，这样保证这些异常会得到合适的处理。

Java 对异常进行了分类，不同类型的异常分别用不同的 Java 类表示，所有异常的根类为 java.lang.Throwable，Throwable 下面又派生了两个子类：Error 和 Exception，Error 错误，一般是指虚拟机相关的问题，如系统崩溃、虚拟机出错误、动态链接失败等，这种错误无法恢复或不可能捕获，将导致应用程序中断。通常应用程序无法处理这些错误，因此应用程序不应该试图使用 catch 块来捕获 Error 对象。

Exception 表示一种设计或实现问题。也就是说，程序员应该对这些情况进行考虑，并提供相应的处理。

异常又可分为 Runtime 异常和 Checked 异常，Checked 异常体现了 Java 的设计哲学：没有完善错误处理的代码根本就不会被执行！对于 Checked 异常的处理方式有两种：

A．当前方法明确知道如何处理该异常，程序应该使用 try...catch 块来捕获该异常，然后在对应的 catch 块中修补该异常。

B．当前方法不知道如何处理这种异常，应该在定义该方法时声明抛出该异常。

实际上 Java 的 Checked 异常后来“争议不断”，因为 Checked 异常要求程序员要么显式声明抛出，要么进行捕获，不能对 Checked 异常不闻不问，这样就给编程带来了一定的复杂度，比如 Spring、Hibernate 框架的一大特点就是把 Checked 异常包装成了 Runtime 异常。

Runtime 异常则比较灵活，开发者既可以选择捕获 Runtime 异常，也可以不捕获。

78．请写出你最常见到的 5 个 runtime exception。

对于一个有 1～2 年编程经验的人来说，总会经常遇到一些常见的异常，其中有些就是 Runtime Exception。比如：

NullPointerException——当调用一个未初始化的引用变量（实际值为 null）的实例 Field、实例方法时都会引发该异常。

ArithmeticException——算术异常。比如 5/0 将引发该异常。

ArrayIndexOutOfBoundsException——数组索引越界异常。

ClassCastException——类型转换异常。

IllegalArgumentException——参数非法的异常。

79．Java 语言如何进行异常处理？关键字 throws、throw、try、catch、finally 分别代表什么意义？在 try 块中可以抛出异常吗？

try 块表示程序正常的业务执行代码。如果程序在执行 try 块的代码时出现了“非预期”情况，JVM 将会生成一个异常对象，这个异常对象将会被后面相应的 catch 块捕获。

catch 块表示一个异常捕获块。当程序执行 try 块引发异常时，这个异常对象将会被后面相应的 catch 块捕获。

throw 用于手动抛出异常对象。throw 后面需要一个异常对象。

throws 用于在方法签名中声明抛出一个或多个异常类，throws 关键字后可以紧跟一个或多个异常类。

finally 块代表异常处理流程中总会执行的代码块。

对于一个完整的异常处理流程而言，try 块是必需的，try 块后可以紧跟一个或多个 catch 块，最后还可以带一个 finally 块。Java 7 引入了自动关闭资源的 try 语句，这种自动关闭资源的 try 语句可以单独存在。例如如下代码是正确的：

```
try(
    // 声明并创建可以被自动关闭的资源
)
{
    // 执行语句
}
```

在上面 try 关键字的圆括号内，可用于声明并创建能被自动关闭的资源，这些能被自动关闭的资源必须实现 Closeable 或 AutoCloseable 接口。

80．Java 中有几种方法可以实现一个线程？用什么关键字修饰同步方法？stop()和 suspend()方法为何不推荐使用？

在 Java 5 以前，有如下两种方法：

第一种：继承 Thread 类，重写它的 run()方法。

代码如下：

```
new Thread()
{
    public void run()
    {
        // 线程执行体
    }
}.start();
```

第二种：实现 Runnable 接口，并重写它的 run()方法。

代码如下：

```
new Thread(new Runnable()
{
    public void run()
    {
        // 线程执行体
    }
}).start();
```

从上面代码不难看出，线程的执行体是一个 run()方法，然后程序通过 start()方法启动一个线程。

从 Java 5 开始，Java 提供了第三种方式来创建多线程：实现 Callable 接口，并实现 call()方法。Callable 接口相当于 Runnable 接口的增强版，因为 Callable 接口中定义的 call()方法既拥有返回值，也可以声明抛出异常。

代码如下：

```
new Thread(new FutureTask<Object >(new Callable<Object>()
{
    public Object call() throws Exception
    {
        // 线程执行体
    }
})).start();
```

不仅如此，Java 5 还提供了线程支持，ExecutorService 对象就代表了线程池，如果开发者利用 ExecutorService 来启动线程，ExecutorService 底层会负责管理线程池。此时，开发者只要把 Runnable 对象传给 ExecutorService 即可。代码如下：

```
ExecutorService pool = Executors.newFixedThreadPool(3)
pool.execute(new Runnable()
{
    public void run()
    {
        // 线程执行体
    }
});
```

如果执行通过 Callable 方式实现的线程，则可按如下代码：

```
ExecutorService pool = Executors.newFixedThreadPool(3)
pool.execute(new FutureTask<Object >(new Callable<Object>()
{
    public Object call() throws Exception
    {
        //线程执行体
    }
}));
```

用 synchronized 关键字修饰同步方法。需要指出的是，非静态的同步方法的同步监视器是 this，也就是调用该方法的对象，而静态的同步方法的同步监视器则是该类本身。因此，使用 synchronized 修饰的静态方法、非静态方法的同步监视器并不相同，只有基于同一个同

步监视器的同步方法、同步代码块才能实现同步。

反对使用 stop()，是因为它不安全。它会解除由线程获取的所有锁定，而且如果对象处于一种不连贯状态，那么其他线程能在那种状态下检查和修改它们，结果很难检查出真正的问题所在。suspend()方法容易发生死锁。调用 suspend()的时候，目标线程会停下来，但却仍然持有在这之前获得的锁定。此时，其他任何线程都不能访问锁定的资源，除非被“挂起”的线程恢复运行。对任何线程来说，如果它们想恢复目标线程，同时又试图使用任何一个锁定的资源，就会造成死锁。所以不应该使用 suspend()，而应在自己的 Thread 类中置入一个标志，指出线程应该活动还是挂起。若标志指出线程应该挂起，便用 wait()命其进入等待状态；若标志指出线程应当恢复，则用一个 notify()重新启动线程。

81．sleep()和 wait()有什么区别？

sleep()是 Thread 类的静态方法，它的作用是让当前线程从运行状态转入阻塞状态，线程执行暂停下来，当一个线程通过 sleep()方法暂停之后，该线程并不会释放它对同步监视器的加锁。

wait()是 Object 对象的方法，但实际上只有同步监视器才能调用该方法。当程序在同步代码块或同步方法内通过同步监视器调用该方法时，将会导致当前线程释放对该同步监视器的加锁，而该线程则会进入该同步监视器的等待池中，直到该同步监视器调用 notify()或 notifyAll()来通知该线程。

82．同步和异步有何异同？在什么情况下分别使用它们？举例说明。

如果有一个资源需要被一个或多个线程共享，这个资源就变成了“竞争”资源，此时多个线程必须按某种既定的规则依次访问、修改这个“竞争”资源，当一个线程正在访问、修改该“竞争”资源时，其他线程不能同时修改这个“竞争”资源，这就是同步处理。

对于一个银行账户，当有多个线程试图去访问这个账户时，如果不对多个线程进行同步控制，有可能账户余额只有 1000 块，但多个线程都试图取款 800 块时，这些线程同时判断余额之后，都会显示余额足够，从而导致每个线程都取款成功。这显然不是我们希望看到的结果。

当程序试图执行一个耗时操作时，程序不希望阻塞当前执行流，因此程序也不应该试图立即获取该耗时操作返回的结果，此时就使用异步编程了，典型的应用场景就是 Ajax。当浏览器通过 JavaScript 发出一个异步请求之后，JavaScript 执行流并不会停下来，而是继续向下执行，这就是异步。程序会通过监听器来监听远程服务器响应的到来。

83．多线程有几种实现方法？同步有几种实现方法？

在 Java 5 以前，有如下两种方法。

第一种：继承 Thread 类，重写它的 run()方法。

代码如下：

```
new Thread()
```

```
{
    public void run()
    {
        // 线程执行体
    }
}.start();
```

第二种：实现 Runnable 接口，并重写它的 run()方法。

代码如下：

```
new Thread(new Runnable()
{
    public void run()
    {
        // 线程执行体
    }
}).start();
```

从上面代码不难看出，线程的执行体是一个 run()方法，然后程序通过 start()方法启动一个线程。

从 Java 5 开始，Java 提供了第三种方式来创建多线程：实现 Callable 接口，并实现 call()方法。Callable 接口相当于 Runnable 接口的增强版，因为 Callable 接口中定义的 call()方法既拥有返回值，也可以声明抛出异常。

代码如下：

```
new Thread(new FutureTask<Object >(new Callable<Object>()
{
    public Object call() throws Exception
    {
        // 线程执行体
    }
})).start();
```

不仅如此，Java 5 还提供了线程支持，ExecutorService 对象就代表了线程池，如果开发者利用 ExecutorService 来启动线程，ExecutorService 底层会负责管理线程池。此时，开发者只要把 Runnable 对象传给 ExecutorService 即可。代码如下：

```
ExecutorService pool = Executors.newFixedThreadPool(3)
pool.execute(new Runnable()
{
    public void run()
    {
        // 线程执行体
    }
});
```

如果执行通过 Callable 方式实现的线程，则可按如下代码：

```
ExecutorService pool = Executors.newFixedThreadPool(3)
pool.execute(new FutureTask<Object >(new Callable<Object>()
{
    public Object call() throws Exception
```

```
    {
        // 线程执行体
    }
}));
```

线程同步大致有如下 3 种实现方式：

A. 通过同步代码块来实现线程同步，使用同步代码块的语法格式为：

```
synchronized(同步监视器)
{
    同步代码块
}
```

对于同步代码块而言，需要程序员显式指定同步监视器。任何线程进入同步代码块之前，必须先对同步监听器加锁。当线程离开同步代码块的时候，该线程会释放对同步监视器的加锁。

B. 同步方法。使用 synchronized 修饰的方法就是同步方法，如果 synchronized 修饰的同步方法是实例方法，那么该同步方法的同步监视器就是 self——也就是该方法的调用者对象；如果 synchronized 修饰的同步方法是类方法，那么该同步方法的同步监视器就是该类本身。

C. 使用 Java 5 提供的 Lock 来实现同步。Lock 比传统的 synchronized 具有更精确的线程语义和更好的性能，Lock 需要程序员显式使用 tryLock()方法执行加锁，也需要程序员显式释放锁。

84．启动一个线程是用 run()还是 start()？

启动一个线程是调用 start()方法，使线程进入就绪状态，以后可以被调度为运行状态。run()方法是线程的线程执行体——也就是线程将要完成的事情。

85．当一个线程进入一个对象的 synchronized 方法后，其他线程是否可进入此对象的其他方法?

当一个线程进入一个对象的 synchronized 方法之后，其他线程完全有可能再次进入该对象的其他方法。

不过要分以下几种情况来看。

（1）如果其他方法没有使用 synchronized 关键字修饰，则可以进入。

（2）如果当前线程进入的 synchronized 方法是 static 方法，那么其他线程可以进入其他 synchronized 修饰的非静态方法；如果当前线程进入的 synchronized 方法是非 static 方法，那么其他线程可以进入其他 synchronized 修饰的静态方法。

（3）如果两个方法都是静态方法，或者都是非静态方法，并且都使用了 synchronized 修饰，但只要在该方法内部调用了同步监视器的 wait()，则其他线程依然可以进入其他使用 synchronized 修饰的方法。

（4）如果两个方法都是静态方法，或者都是非静态方法，并且都使用了 synchronized 修饰，而且没有在该方法内部调用同步监视器的 wait()，则其他线程不能进入其他使用 synchronized 修饰的方法。

86. 线程的基本概念、线程的基本状态以及状态之间的关系。

多线程扩展了多进程的概念，使得同一个进程可以同时并发处理多个任务。线程（Thread）也被称作轻量级进程（Lightweight Process），线程是进程的执行单元。就像进程在操作系统中的地位一样，线程在程序中是独立的、并发的执行流。当进程被初始化后，主线程就被创建了。对于绝大多数的应用程序来说，通常仅要求有一个主线程，但我们也可以在该进程内创建多条顺序执行流，这些顺序执行流就是线程，每个线程也是互相独立的。

线程是进程的组成部分，一个进程可以拥有多个线程，一个线程必须有一个父进程。线程可以拥有自己的堆栈、自己的程序计数器和自己的局部变量，但不再拥有系统资源，它与父进程的其他线程共享该进程所拥有的全部资源。因为多个线程共享父进程里的全部资源，因此编程更加方便；但必须更加小心，我们必须确保线程不会妨碍同一进程里的其他线程。

线程的执行需要经过如下状态：

新建

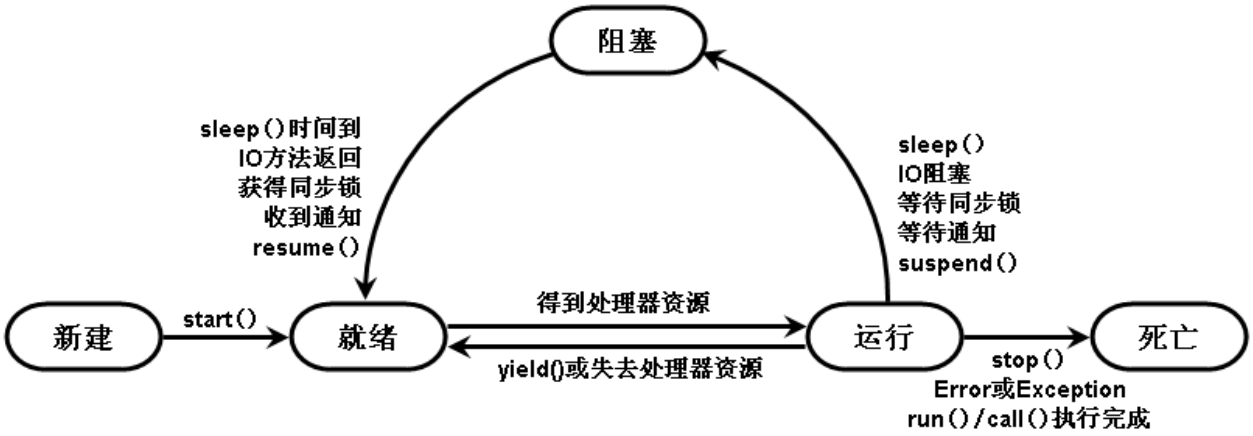
就绪

运行

阻塞

死亡

各状态的转换关系如下图所示：



87．简述 synchronized 和 java.util.concurrent.locks.Lock 的异同。

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。synchronized 会自动释放锁，而 Lock 一定要求程序员手工释放，并且必须在 finally 从句中释放。Lock 还有更强大的功能，例如，它的 tryLock 方法可以非阻塞方式去拿锁。

Java 代码查错部分

1.

```
abstract class Name {
    private String name;
    public abstract boolean isStupidName(String name) {}
}
```

答案：错。abstract 方法必须以分号结尾，且不带花括号。

2.

```
public class Something {
    void doSomething () {
        private String s = "";
        int l = s.length();
    }
}
```

有错吗？

答案：错。局部变量前不能放置任何访问修饰符（private、public 和 protected）。final 可以用来修饰局部变量

（final 如同 abstract 和 strictfp，都是非访问修饰符，strictfp 只能修饰 class 和 method 而非 variable）。

3.

```
abstract class Something {
    private abstract String doSomething ();
}
```

这好像没什么错吧？

答案：错。abstract 的 methods 不能以 private 修饰。abstract 的 methods 就是让子类 implement（实现）具体细节的，怎么可以用 private 把 abstract method 隐藏起来呢？（同理，abstract method 前不能加 final）。

4.

```
public class Something {
    public int addOne(final int x) {
        return ++x;
    }
}
```

答案：错。int x 被修饰成 final，意味着 x 不能在 addOne method 中被修改。

5.

```
public class Something {
    public static void main(String[] args) {
        Other o = new Other();
```

```
        new Something().addOne(o);
    }
    public void addOne(final Other o) {
        o.i++;
    }
}
class Other {
    public int i;
}
```

和上面的很相似，都是关于 final 的问题，这有错吗？

答案：正确。在 addOne method 中，参数 o 被修饰成 final。如果在 addOne method 里我们修改了 o 的引用（比如：o = new Other();），那么如同上例这题也是错的。但这里修改的是 o 的成员变量，而 o 的 reference 并没有改变，因此程序没有错误。

6.

```
class Something {
    int i;
    public void doSomething() {
        System.out.println("i = " + i);
    }
}
```

这有什么错吗?

答案：正确。输出的是"i = 0"。int i 是定义实例变量。系统会对实例变量执行默认初始化，因此 i 默认被赋值为 0。

7.

```
class Something {
    final int i;
    public void doSomething() {
        System.out.println("i = " + i);
    }
}
```

和上面一题只有一个地方不同，就是多了一个 final。

答案：错。使用 final 修饰的实例变量必须由程序员显式指定初始值，为 final 变量指定初始值有 3 个地方：

A．定义时指定初始值。

B．在初始化块中指定初始值。

C．在构造器中指定初始值。

8.

```
public class Something {
     public static void main(String[] args) {
        Something s = new Something();
        System.out.println("s.doSomething() returns "
```

```
            + doSomething());
    }
    public String doSomething() {
        return "Do something ...";
    }
}
```

答案：错。静态成员不允许访问非静态成员。上面的 main 方法是静态方法，而 doSomething() 是非静态方法，因此程序导致编译错误。

9.

此处，Something 类的文件名叫 OtherThing.java。

```
class Something {
    private static void main(String[] something_to_do) {
        System.out.println("Do something ...");
    }
}
```

答案：正确。由于 Something 类不是 public 类，因此 Java 源文件的主文件名可以是任意的。但 public class 的名字必须和文件名相同。

10.

```
interface A{
    int x = 0;
}
class B{
    int x =1;
}
class C extends B implements A {
    public void printX(){
        System.out.println(x);
    }
    public static void main(String[] args) {
        new C().printX ();
    }
}
```

答案：错误。在编译时会发生错误。因为 C 类既实现了 A 接口，也继承 B，因此它将会从 A 接口、B 类中分别继承到成员变量 x。所以，上面程序中 System.out.println(x);代码所引用的 x 是不明确的。

对于父类的变量，可以用 super.x 来明确指定，而接口的属性默认隐含为 public static final。所以可以通过 A.x 来明确指定。

11.

```
interface Playable {
    void play();
}
```

```
interface Bounceable {
    void play();
}
interface Rollable extends Playable, Bounceable {
    Ball ball = new Ball("PingPang");
}
class Ball implements Rollable {
    private String name;
    public String getName() {
        return name;
    }
    public Ball(String name) {
        this.name = name;
    }
    public void play() {
        ball = new Ball("Footboall");
        System.out.println(ball.getName());
    }
}
```

答案：错。"interface Rollable extends Playable, Bounceable"没有问题。interface 可继承多个 interfaces，所以这里没错。问题出在 interface Rollable 里的"Ball ball = new Ball("PingPang");"。在接口里声明的成员变量总是常量，也就是默认使用 public static final 修饰。也就是说，"Ball ball = new Ball("PingPang");"实际上是"public static final Ball ball = new Ball("PingPang");"。在 Ball 类的 Play()方法中，"ball = new Ball("Footboall");"尝试改变了 ball 的引用，而这里的 ball 引用变量是在 Rollable 中定义的，因此它有 final 修饰，final 修饰的引用变量是不能被重新赋值的，所以上面程序会导致编译错误。