# JAVA程序设计基础概念

### Java语言特征

#### 面向对象的特征有哪些方面

答：  
**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个方面。抽象一般只关注这些对象有哪些属性和行为并不关心这些行为是什么。

**封装**：通常认为封装就是将数据和操作数据的方法绑定起来，对数据的访问只能通过定义好的接口。可以说封装就是隐藏一切能隐藏的东西，只向外界提供最简单的操作方式。

**继承**：继承是从已有类得到信息，创建新类的过程 。提供信息的类被称作父类（超类，基类），继承信息的类被称作子类（派生类）。继承让变化中的软件系统有了延续性，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）。

**多态**：多态是指不同的子类对象对同一消息做出不同的响应。简单来说就是就是用同样的对象，调用同样的方法但是做了不同的事情。多态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已有的(可以是抽象,已实现的，为被final修饰的)方法  
2）对象造型：用父类型引用子类型对象，这样才调用方法就会通过同样的引用同样的方法跟据子类的不同而表现出不同的行为

#### **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。）

   


#### **描述一下JVM加载class文件的原理机制？**

**答：**  
 Java中的所有类，都需要由类加载器装载到JVM中才能运行。类加载器本身也是一个类，而它的工作就是把class文件从硬盘读取到内存中。在写程序的时候，我们几乎不需要关心类的加载，因为这些都是隐式装载的，除非我们有特殊的用法，像是反射，就需要显式的加载所需要的类。

Java类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行的基础类(像是基类)完全加载到jvm中，至于其他类，则在需要的时候才加载。这当然就是为了节省内存开销。

Java的类加载器有三个，对应Java的三种类:

1）Bootstrap ClassLoader

负责加载$JAVA\_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类

2）Extension ClassLoader

负责加载java平台中扩展功能的一些jar包，包括$JAVA\_HOME中jre/lib/\*.jar或-Djava.ext.dirs指定目录下的jar包

3）App ClassLoader

负责加载classpath中指定的jar包及目录中class(程序员自定义的类)

三个加载器各自完成自己的工作，但它们是如何协调工作呢？哪一个类该由哪个类加载器完成呢？为了解决这个问题，Java采用了委托模型机制。

委托模型机制的工作原理很简单：当类加载器需要加载类的时候，先请示其Parent(即上一层加载器)在其搜索路径载入，如果找不到，才在自己的搜索路径搜索该类。这样的顺序其实就是加载器层次上自顶而下的搜索，因为加载器必须保证基础类的加载。之所以是这种机制，还有一个安全上的考虑：如果某人将一个恶意的基础类加载到jvm，委托模型机制会搜索其父类加载器，显然是不可能找到的，自然就不会将该类加载进来。

我们可以通过这样的代码来获取类加载器:

ClassLoader loader = ClassName.class.getClassLoader();

ClassLoader ParentLoader = loader.getParent();

注意一个很重要的问题，就是Java在逻辑上并不存在BootstrapLoader的实体！因为它是用C++编写的，所以打印其内容将会得到null。  
前面是对类加载器的简单介绍，它的原理机制非常简单，就是下面几个步骤:

1.装载:查找和导入class文件;

2.连接:

(1)检查:检查载入的class文件数据的正确性;

(2)准备:为类的静态变量分配存储空间;

(3)解析:将符号引用转换成直接引用(这一步是可选的)

3.初始化:初始化静态变量，静态代码块。

这样的过程在程序调用类的静态成员的时候开始执行，所以静态方法main()才会成为一般程序的入口方法。类的构造器也会引发该动作。

**注：**  
**符号引用**（Symbolic References）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能够无歧义的定位到目标即可。例如，在Class文件中它以CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info等类型的常量出现。符号引用与虚拟机的内存布局无关，引用的目标并不一定加载到内存中。在Java中，一个java类将会编译成一个class文件。在编译时，java类并不知道所引用的类的实际地址，因此只能使用符号引用来代替。比如org.simple.People类引用了org.simple.Language类，在编译时People类并不知道Language类的实际内存地址，因此只能使用符号org.simple.Language（假设是这个，当然实际中是由类似于CONSTANT\_Class\_info的常量来表示的）来表示Language类的地址。各种虚拟机实现的内存布局可能有所不同，但是它们能接受的符号引用都是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

**直接引用**：直接引用可以是

（1）直接指向目标的指针（比如，指向“类型”【Class对象】、类变量、类方法的直接引用可能是指向方法区的指针）

（2）相对偏移量（比如，指向实例变量、实例方法的直接引用都是偏移量）

（3）一个能间接定位到目标的句柄直接引用是和虚拟机的布局相关的，同一个符号引用在不同的虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经被加载入内存中了。

#### ****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。

#### **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！

答：  
float max(int a, int b);  
int max(int a, int b);  
当调用max(1, 2);时无法确定调用的是哪个，单从这一点上来说，仅返回值类型不同的重载是不应该允许的。  
函数的返回值只是作为函数运行之后的一个“状态”  
他是保持方法的调用者与被调用者进行通信的关键。  
并不能作为某个方法的“标识”

#### **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。

#### **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。

#### **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。

#### **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。

#### **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）。

#### **指出下面程序的运行结果。**

public class Outer {

static class A {

static {

System.out.print("1");

}

public A() {

System.out.print("2");

}

}

static class B extends A{

static {

System.out.print("a");

}

public B() {

System.out.print("b");

}

}

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。

#### **抽象类（abstract class）和接口（interface）有什么异同？**

答：  
1.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。  
2.一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。  
3.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。  
4.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。  
5.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。  
6.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。

#### **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;

}

}

}

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();

#### **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。

#### **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。

#### **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。

#### **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。

#### **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](http://z.cn/)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。

#### **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。

#### **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）

#### **switch 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。

#### **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）

例子：

loop1:for(int i = 0;i<100;i++)  
{  
 System.out.println("i = " + i);  
 loop2:for(int j = 0;j<100;j++)  
 {  
 System.out.println("j = " + j);  
 continue loop1;  
 //或者break loop1；  
 }  
}

#### **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。

#### **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。

### 基本数据类型(包装类)与类型转换

#### Java的基本数据类型与转换

答：  
整形类型：byte (8bits),short(16bits) , int (32bits) , long (64bits)  
浮点类型：float(32bits) , double(64bits)  
布尔类型：boolean(8bits)  
字符类型：char(16bits)( java内部使用了unicode编码可以储存一个汉字)   
  
**转换规则：从低精度向高精度转换byte 、short、int、long、float、double、char  
高精度向低精度转换使用强制转换但会丢失精度  
注：两个char型运算时，自动转换为int型；当char与别的类型运算时，也会先自动转换为int型的，再做其它类型的自动转换。**  
对应的包装类有：Byte、Short、Integer、Long 、Float、Double、Character、Boolean  
基本类型能够和包装类型进行隐式转换也就是装箱和拆箱  
例1：  
public static void main(String[] args) throws Exception {  
 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 自动拆箱成int类型再和c比较  
   
}

例2：

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);

}

}

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。查看源码后发现，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。

#### **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;。

#### **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。

### 运算符

#### i++ 面试题1(下面的面试题会输出什么？)

public class Test {  
 static {  
 int x = 5;  
 }  
 static int x, y;  
 public static void main(String[] args) {  
 x--;  
 myMethod();  
 System.out.println(x + y++ + x);  
 }  
 public static void myMethod() {  
 y = x++ + ++x;  
 }  
}

答：  
结果为：2  
public class Test {  
 static {  
 int x = 5;//第一次被载入jvm时运行。由于是局部变量，x=5不影响后面的值  
 }  
 static int x, y;//初始化时x=0,y=0 ;  
 public static void main(String[] args) {  
 x--;// x = -1  
 myMethod();  
 //1 + 0 + 1 输出 2   
 System.out.println(x + y++ + x);  
 //此时 y = 1  
 }  
 public static void myMethod() {  
 //y = 0 + 0 注先运算++x（从右到作运算）  
 y = x++ + ++x;  
 //此时x = 1  
 }  
}

注：

**i++**可以当作运算一个方法来理解如（以c#方式理解）  
int i++（ref int i）

{  
int temp = i；

i = i + 1;

return temp;

}

**++i**也可以当作运算一个方法来理解如（以c#方式理解）

int ++i（ref int i）

{

i = i + 1;

return i;

}

#### i++ 面试题2(下面的面试题会输出什么？)

public static void main(String[] args) {  
 int j = 0;  
 for (int i = 0; i < 100; i++) {  
 j = j++;  
 }  
 System.out.println("j = " + j);  
}

答  
输出 j = 0。看上一题注解

#### **>> 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JVM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.hashCode());

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;

}

}

#### ****&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1符（|）和短路或运算符（||）的差别也是如此。

**补充1：**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。

**按位与规则：**1&1=1  
1&0=0  
0&1=0  
0&0=0

#### | 和 || 的区别

答：  
两者都表示“或”运算，但是||运算符第一个表达式成立的话，后面的表达式不运算，直接返回。而|对所有表达式都得判断。

注：  
| 叫 按位或  
1|1=1  
1|0=1  
0|1=1  
0|0=0

### 异常

#### **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）。

#### **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。

#### **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

#### **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）

#### ****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，强烈建议将此项设置为编译错误。

#### **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!");

}

}

}

答：  
Caught Annoyance  
Caught Sneeze  
Hello World!

解释：  
抛出异常的时候，异常处理系统会按照代码的书写顺序找出最近的处理程序。找到匹配的处理程序后,它将认为异常得到处理，然后就不在寻找。  
查找的时候并不要求抛出的异常同处理程序所声明的异常完全匹配。派生类的对象也可以匹配基类的处理程序。  
多重 catch 语句中，异常类型必须子类在前父类在后，如果你把父类放前面就执行不到后边的了，比如你把Exception放到第一位，那么后面的就不会得到执行了，而且通常子类的信息比父类的要更精准一点，所以通常都是子类放前面。

### 反射

#### 概念

反射的概念是由Smith在1982年首次提出的，主要是指程序可以访问、检测和修改它本身状态或行为的一种能力。这一概念的提出很快引发了[计算机科学](https://www.baidu.com/s?wd=%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)领域关于应用反射性的研究。它首先被程序语言的设计领域所采用,并在Lisp和[面向对象](https://www.baidu.com/s?wd=%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)方面取得了成绩。其中LEAD/LEAD++ 、OpenC++ 、MetaXa和OpenJava等就是基于反射机制的语言。最近，反射机制也被应用到了视窗系统、[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)和文件系统中。

反射本身并不是一个新概念，它可能会使我们联想到光学中的反射概念，尽管[计算机科学](https://www.baidu.com/s?wd=%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)赋予了反射概念新的含义，但是，从现象上来说，它们确实有某些相通之处，这些有助于我们的理解。在[计算机科学](https://www.baidu.com/s?wd=%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)领域，反射是指一类应用，它们能够自描述和自控制。也就是说，这类应用通过采用某种机制来实现对自己行为的描述（self-representation）和监测（examination），并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。可以看出，同一般的反射概念相比，计算机科学领域的反射不单单指反射本身，还包括对反射结果所采取的措施。所有采用反射机制的系统（即反射系统）都希望使系统的实现更开放。可以说，实现了反射机制的系统都具有开放性，但具有开放性的系统并不一定采用了反射机制，开放性是反射系统的必要条件。一般来说，反射系统除了满足开放性条件外还必须满足原因连接（Causally-connected）。所谓原因连接是指对反射系统自描述的改变能够立即反映到系统底层的实际状态和行为上的情况，反之亦然。开放性和原因连接是反射系统的两大基本要素。13700863760  
Java中，反射是一种强大的工具。它使您能够创建灵活的代码，这些代码可以在运行时装配，无需在组件之间进行源代表链接。反射允许我们在编写与执行时，使我们的程序代码能够接入装载到JVM中的类的内部信息，而不是源代码中选定的类协作的代码。这使反射成为构建灵活的应用的主要工具。但需注意的是：如果使用不当，反射的成本很高。

#### Java中的类反射：

Reflection 是 Java 程序开发语言的特征之一，它允许运行中的 Java 程序对自身进行检查，或者说“自审”，并能直接操作程序的内部属性。Java 的这一能力在实际应用中也许用得不是很多，但是在其它的程序设计语言中根本就不存在这一特性。例如，Pascal、C 或者 C++ 中就没有办法在程序中获得函数定义相关的信息。  
  
1．检测类：  
  
1.1 reflection的工作机制  
  
考虑下面这个简单的例子，让我们看看 reflection 是如何工作的。  
  
import java.lang.reflect.\*;  
public class DumpMethods {  
public static void main(String args[]) {  
try {  
Class c = Class.forName(args[0]);  
Method m[] = c.getDeclaredMethods();  
for (int i = 0; i < m.length; i++)  
System.out.println(m[i].toString());  
} catch (Throwable e) {  
System.err.println(e);  
}  
}  
}  
  
按如下语句执行：  
  
java DumpMethods java.util.Stack  
  
它的结果输出为：  
  
public [java.lang.Object](https://www.baidu.com/s?wd=java.lang.Object&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm) java.util.Stack.push([java.lang.Object](https://www.baidu.com/s?wd=java.lang.Object&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm))  
  
public synchronized [java.lang.Object](https://www.baidu.com/s?wd=java.lang.Object&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm) java.util.Stack.pop()  
  
public synchronized java.lang.Object java.util.Stack.peek()  
  
public boolean java.util.Stack.empty()  
  
public synchronized int java.util.Stack.search(java.lang.Object)  
  
这样就列出了java.util.Stack 类的各方法名以及它们的限制符和[返回类型](https://www.baidu.com/s?wd=%E8%BF%94%E5%9B%9E%E7%B1%BB%E5%9E%8B&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)。  
  
这个程序使用 Class.forName 载入指定的类，然后调用 getDeclaredMethods 来获取这个类中定义了的方法列表。java.lang.reflect.Methods 是用来描述某个类中单个方法的一个类。  
  
1.2 Java类反射中的主要方法  
  
对于以下三类组件中的任何一类来说 -- [构造函数](https://www.baidu.com/s?wd=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)、字段和方法 -- java.lang.Class 提供四种独立的反射调用，以不同的方式来获得信息。调用都遵循一种标准格式。以下是用于查找[构造函数](https://www.baidu.com/s?wd=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)的一组反射调用：  
  
l Constructor getConstructor(Class[] params) -- 获得使用特殊的参数类型的公共[构造函数](https://www.baidu.com/s?wd=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)，  
  
l Constructor[] getConstructors() -- 获得类的所有公共构造函数  
  
l Constructor getDeclaredConstructor(Class[] params) -- 获得使用特定参数类型的构造函数(与接入级别无关)  
  
l Constructor[] getDeclaredConstructors() -- 获得类的所有构造函数(与接入级别无关)  
  
获得字段信息的Class 反射调用不同于那些用于接入构造函数的调用，在参数类型数组中使用了字段名：  
  
l Field getField(String name) -- 获得命名的公共字段  
  
l Field[] getFields() -- 获得类的所有公共字段  
  
l Field getDeclaredField(String name) -- 获得类声明的命名的字段  
  
l Field[] getDeclaredFields() -- 获得类声明的所有字段  
  
用于获得方法信息函数：  
  
l Method getMethod(String name, Class[] params) -- 使用特定的参数类型，获得命名的公共方法  
  
l Method[] getMethods() -- 获得类的所有公共方法  
  
l Method getDeclaredMethod(String name, Class[] params) -- 使用特写的参数类型，获得类声明的命名的方法  
  
l Method[] getDeclaredMethods() -- 获得类声明的所有方法  
  
1.3开始使用 Reflection：  
  
用于 reflection 的类，如 Method，可以在 java.lang.relfect 包中找到。使用这些类的时候必须要遵循三个步骤：第一步是获得你想操作的类的 java.lang.Class 对象。在运行中的 Java 程序中，用 java.lang.Class 类来描述类和接口等。  
  
下面就是获得一个 Class 对象的方法之一：  
  
Class c = Class.forName("[java.lang.String](https://www.baidu.com/s?wd=java.lang.String&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)");  
  
这条语句得到一个 String 类的类对象。还有另一种方法，如下面的语句：  
  
Class c = int.class;  
  
或者  
  
Class c = Integer.TYPE;  
  
它们可获得基本类型的类信息。其中后一种方法中访问的是基本类型的封装类 (如 Integer) 中预先定义好的 TYPE 字段。  
  
第二步是调用诸如 getDeclaredMethods 的方法，以取得该类中定义的所有方法的列表。  
  
一旦取得这个信息，就可以进行第三步了——使用 reflection API 来操作这些信息，如下面这段代码：  
  
Class c = Class.forName("[java.lang.String](https://www.baidu.com/s?wd=java.lang.String&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)");  
  
Method m[] = c.getDeclaredMethods();  
  
System.out.println(m[0].toString());  
  
它将以文本方式打印出 String 中定义的第一个方法的原型。  
  
2．处理对象：  
  
如果要作一个开发工具像debugger之类的，你必须能发现filed values,以下是三个步骤:  
  
a.创建一个Class对象  
b.通过getField 创建一个Field对象  
c.调用Field.getXXX(Object)方法(XXX是Int,Float等，如果是对象就省略；Object是指实例).  
  
例如：  
import java.lang.reflect.\*;  
import [java.awt](https://www.baidu.com/s?wd=java.awt&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm).\*;  
  
class SampleGet {  
  
public static void main(String[] args) {  
Rectangle r = new Rectangle(100, 325);  
printHeight(r);  
  
}  
  
static void printHeight(Rectangle r) {  
Field heightField;  
Integer heightValue;  
Class c = r.getClass();  
try {  
heightField = c.getField("height");  
heightValue = (Integer) heightField.get(r);  
System.out.println("Height: " + heightValue.toString());  
} catch (NoSuchFieldException e) {  
System.out.println(e);  
} catch (SecurityException e) {  
System.out.println(e);  
} catch (IllegalAccessException e) {  
System.out.println(e);  
}  
}  
}  
  
三、安全性和反射：  
在处理反射时安全性是一个较复杂的问题。反射经常由框架型代码使用，由于这一点，我们可能希望框架能够全面接入代码，无需考虑常规的接入限制。但是，在其它情况下，不受控制的接入会带来严重的安全性风险，例如当代码在不值得信任的[代码共享](https://www.baidu.com/s?wd=%E4%BB%A3%E7%A0%81%E5%85%B1%E4%BA%AB&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)的环境中运行时。  
  
由于这些互相矛盾的需求，[Java编程语言](https://www.baidu.com/s?wd=Java%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)定义一种多级别方法来处理反射的安全性。基本模式是对反射实施与应用于源代码接入相同的限制：  
  
n 从任意位置到类公共组件的接入  
  
n 类自身外部无任何到私有组件的接入  
  
n 受保护和打包（缺省接入）组件的有限接入  
  
不过至少有些时候，围绕这些限制还有一种简单的方法。我们可以在我们所写的类中，扩展一个普通的基本类java.lang.reflect.AccessibleObject 类。这个类定义了一种setAccessible方法，使我们能够启动或关闭对这些类中其中一个类的实例的接入检测。唯一的问题在于如果使用了安全性管理器，它将检测正在关闭接入检测的代码是否许可了这样做。如果未许可，安全性管理器抛出一个例外。  
  
下面是一段程序，在TwoString 类的一个实例上使用反射来显示安全性正在运行：  
  
public class ReflectSecurity {  
  
public static void main(String[] args) {  
  
try {  
  
TwoString ts = new TwoString("a", "b");  
  
Field field = clas.getDeclaredField("m\_s1");  
  
// field.setAccessible(true);  
  
System.out.println("Retrieved value is " +  
  
field.get(inst));  
  
} catch (Exception ex) {  
  
ex.printStackTrace(System.out);  
  
}  
  
}  
  
}  
  
如果我们编译这一程序时，不使用任何特定参数直接从命令行运行，它将在field .get(inst)调用中抛出一个IllegalAccessException异常。如果我们不注释field.setAccessible(true)代码行，那么重新编译并重新运行该代码，它将编译成功。最后，如果我们在命令行添加了JVM参数-Djava.security.manager以实现安全性管理器，它仍然将不能通过编译，除非我们定义了ReflectSecurity类的许可权限。  
  
四、反射性能：  
反射是一种强大的工具，但也存在一些不足。一个主要的缺点是对性能有影响。使用反射基本上是一种解释操作，我们可以告诉JVM，我们希望做什么并且它满足我们的要求。这类操作总是慢于只直接执行相同的操作。  
  
下面的程序是字段接入性能测试的一个例子，包括基本的测试方法。每种方法测试字段接入的一种形式 -- accessSame 与同一对象的成员字段协作，accessOther 使用可直接接入的另一对象的字段，accessReflection 使用可通过反射接入的另一对象的字段。在每种情况下，方法执行相同的计算 -- 循环中简单的加/乘顺序。  
  
程序如下：  
  
public int accessSame(int loops) {  
  
m\_value = 0;  
  
for (int index = 0; index < loops; index++) {  
  
m\_value = (m\_value + ADDITIVE\_VALUE) \*  
  
MULTIPLIER\_VALUE;  
  
}  
  
return m\_value;  
  
}  
  
public int accessReference(int loops) {  
  
TimingClass timing = new TimingClass();  
  
for (int index = 0; index < loops; index++) {  
  
timing.m\_value = (timing.m\_value + ADDITIVE\_VALUE) \*  
  
MULTIPLIER\_VALUE;  
  
}  
  
return timing.m\_value;  
  
}  
  
public int accessReflection(int loops) throws Exception {  
  
TimingClass timing = new TimingClass();  
  
try {  
  
Field field = TimingClass.class.  
  
getDeclaredField("m\_value");  
  
for (int index = 0; index < loops; index++) {  
  
int value = (field.getInt(timing) +  
  
ADDITIVE\_VALUE) \* MULTIPLIER\_VALUE;  
  
field.setInt(timing, value);  
  
}  
  
return timing.m\_value;  
  
} catch (Exception ex) {  
  
System.out.println("Error using reflection");  
  
throw ex;  
  
}  
  
}  
  
在上面的例子中，测试程序重复调用每种方法，使用一个大循环数，从而平均多次调用的时间衡量结果。平均值中不包括每种方法第一次调用的时间，因此初始化时间不是结果中的一个因素。下面的图清楚的向我们展示了每种方法字段接入的时间：  
  
图 1：字段接入时间 ：  
  
我们可以看出：在前两副图中(Sun JVM)，使用反射的执行时间超过使用直接接入的1000倍以上。通过比较，IBM JVM可能稍好一些，但反射方法仍旧需要比其它方法长700倍以上的时间。任何JVM上其它两种方法之间时间方面无任何显著差异，但IBM JVM几乎比Sun JVM快一倍。最有可能的是这种差异反映了Sun Hot Spot JVM的专业优化，它在简单基准方面表现得很糟糕。反射性能是Sun开发1.4 JVM时关注的一个方面，它在反射方法调用结果中显示。在这类操作的性能方面，Sun 1.4.1 JVM显示了比1.3.1版本很大的改进。  
  
如果为为创建使用反射的对象编写了类似的计时测试程序，我们会发现这种情况下的差异不象字段和方法调用情况下那么显著。使用newInstance()调用创建一个简单的java.lang.Object实例耗用的时间大约是在Sun 1.3.1 JVM上使用new Object()的12倍，是在IBM 1.4.0 JVM的四倍，只是Sun 1.4.1 JVM上的两部。使用Array.newInstance(type, size)创建一个数组耗用的时间是任何测试的JVM上使用new type[size]的两倍，随着数组大小的增加，差异逐步缩小。  
  
结束语：  
Java语言反射提供一种动态链接程序组件的多功能方法。它允许程序创建和控制任何类的对象(根据安全性限制)，无需提前硬编码目标类。这些特性使得反射特别适用于创建以非常普通的方式与对象协作的库。例如，反射经常在持续存储对象为数据库、XML或其它外部格式的框架中使用。Java reflection 非常有用，它使类和数据结构能按名称动态检索相关信息，并允许在运行着的程序中操作这些信息。Java 的这一特性非常强大，并且是其它一些常用语言，如 C、C++、Fortran 或者 Pascal 等都不具备的。  
  
但反射有两个缺点。第一个是性能问题。用于字段和方法接入时反射要远慢于直接代码。性能问题的程度取决于程序中是如何使用反射的。如果它作为程序运行中相对很少涉及的部分，缓慢的性能将不会是一个问题。即使测试中最坏情况下的计时图显示的反射操作只耗用几微秒。仅反射在性能关键的应用的核心逻辑中使用时性能问题才变得至关重要。  
  
许多应用中更严重的一个缺点是使用反射会模糊程序内部实际要发生的事情。程序人员希望在源代码中看到程序的逻辑，反射等绕过了源代码的技术会带来维护问题。反射代码比相应的直接代码更复杂，正如性能比较的代码实例中看到的一样。解决这些问题的最佳方案是保守地使用反射——仅在它可以真正增加灵活性的地方——记录其在目标类中的使用。  
  
利用反射实现类的动态加载  
  
Bromon原创 请尊重版权  
  
最近在成都写一个移动增值项目，俺负责后台server端。功能很简单，手机用户通过GPRS打开Socket与服务器连接，我则根据用户传过来的数据做出响应。做过类似项目的兄弟一定都知道，首先需要定义一个类似于MSNP的通讯协议，不过今天的话题是如何把这个系统设计得具有高度的扩展性。由于这个项目本身没有进行过较为完善的客户沟通和需求分析，所以以后肯定会有很多功能上的扩展，通讯协议肯定会越来越庞大，而我作为一个不那么勤快的人，当然不想以后再去修改写好的程序，所以这个项目是实践[面向对象](https://www.baidu.com/s?wd=%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPHfkuhD1PADkrAu9uyuh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjTvnHcYPWm)设计的好机会。  
  
首先定义一个接口来隔离类：  
  
package org.bromon.reflect;  
  
public interface Operator  
  
{  
  
public java.util.List act(java.util.List params)  
  
}  
  
根据设计模式的原理，我们可以为不同的功能编写不同的类，每个类都继承Operator接口，客户端只需要针对Operator接口编程就可以避免很多麻烦。比如这个类：  
  
package org.bromon.reflect.\*;  
  
public class Success implements Operator  
  
{  
  
public java.util.List act(java.util.List params)  
  
{  
  
List result=new ArrayList();

#### 如何通过反射创建对象

答：  
1) 通过类对象调用newInstance()方法(注意构造器是否重载)  
例：  
 Class classType = Class.forName("包名.类名");//类的路径  
 Object obj = classType.newInstance();//创建对象的实例  
2）通过调用类的自定义构造器创建对象  
例：  
 public class Test {  
 public Test(int a) {  
 System.out.println("a = " + a);  
 }  
 int a = 1;  
 public static void main(String[] args) throws ClassNotFoundException, IllegalAccessException, InstantiationException, NoSuchMethodException, InvocationTargetException {  
 @SuppressWarnings("rawtypes")  
 Class classType = Test.class;//这里使用第一种方式也是可以的  
 @SuppressWarnings("unchecked")  
 Constructor<Test> con = classType.getConstructor(int.class);//获取构造函数需要传入指定类型的class  
 Object obj = con.newInstance(23);//调用构造函数返回创建的实例对象的引用  
 }  
}

#### 通过反射设置和获取字段的私有值？

答：  
public class Test {  
 private int a = 1;  
 public static void main(String[] args) throws ClassNotFoundException, IllegalAccessException, InstantiationException, NoSuchMethodException, InvocationTargetException, NoSuchFieldException {  
 //获取字节码对象  
 Class clazz = Class.forName("test.Test");  
 //实例化  
 Object obj = clazz.newInstance();  
 //转化为实际对象(这里可以用继承实现)  
 Test t = (Test)obj;  
 //获取字段-字段名称  
 Field field = clazz.getDeclaredField("a");  
 //设置私有变量访问(私有 公有都可以设置不会报错)  
 field.setAccessible(true);  
 //设置变量值  
 field.set(t, 23);  
 //获取变量值  
 System.out.println(field.get(t).toString());  
 }  
}

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;

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, Object 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().getDeclaredConstructor();

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);

}

}

}

#### **如何通过反射调用对象的方法？**

答：  
String str = "hello";  
Method m = str.getClass().getMethod("toUpperCase");  
System.out.println(m.invoke(str)); // HELLO

#### 反射效率问题

答：  
今天，用到了java的反射机制，忽然想到有同事说java的反射机制的效率非常的低，我google了一下，已经很多同学做了实验，效率比直接调用低了20多倍(java7)，不过，后来看到有人说，可以优化一下，把反射后的Method对象缓存下来，保存在一个map中，每次直接调用method.invoke(Object obj, Object... args)即可，但我担心线程安全问题。  
不过，经过多方资料显示，method.invoke(Object obj, Object... args)方法是线程安全的，因为，spring的pxoryFactoryBean也是用的反射机制，并且它也是应用于多线程的环境。所以这个方法应该是线程安全的。

# 字符串-数组-泛型

### String

#### **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），剩下的都是引用类型（reference type），Java 5以后引入的枚举类型也算是一种比较特殊的引用类型。

#### **String有没有length()方法？数组有没有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。

#### **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）。

#### **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";//此时存在于栈中->常量池  
String s2 = new String("Programming");//在堆上分配内存  
String s3 = "Program";  
String s4 = "ming";  
String s5 = "Program" + "ming";//在栈上分配内存  
String s6 = s3 + s4;//在堆上分配内存  
System.out.println(s1 == s2);//s1内存指向栈->常量池中某块内存，s2则指向堆上的某块内存  
System.out.println(s1 == s5);//由于s1、s5字符串相同，并且内存都分配在栈上  
System.out.println(s1 == s6);//s1内存指向栈->常量池中某块内存，s6则指向堆上的某块内存  
System.out.println(s1 == s6.intern());//s1内存指向栈->常量池中某块内存，s6.intern()则指向常量池中某块内存  
System.out.println(s2 == s2.intern());//s2指向堆上的某块内存,s2.intern()则指向常量池中某块内存

}

}

补充：解答上面的面试题需要清除两点：1. 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；2. 字符串的+操作其本质是创建了StringBuilder对象进行append操作，然后将拼接后的StringBuilder对象用toString方法处理成String对象，这一点可以用javap -c StringEqualTest.class命令获得class文件对应的JVM字节码指令就可以看出来。

#### **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。

#### **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");

#### **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。

例1：  
public static void main(String[] args) {  
 String a="abcde";  
 StringBuilder b = new StringBuilder(a);  
 System.out.print(b.reverse().toString());  
}  
例2：

public static void main(String[] args) {  
 // 创建字符串缓冲区对象  
 StringBuffer sb = new StringBuffer("霞青林爱我");  
 System.out.println("sb:" + sb);  
 sb.reverse();  
 System.out.println("sb:" + sb);  
}

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);  
}

#### **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

#### 扩展知识1：Constant Pool常量池概念

在java编译好的class文件中有个区域被称为Constant Pool，它是一个由数组组成的表，类型为cp\_info\_pool[]，用来储存程序中使用的各种常量，包括class String Integer等java数据类型

String pool是对应于Constant Pool 中储存String 常量的区域，被大家习惯性的称为String pool

对于Contstant Pool，表的基本通用结构如下

cp\_info{

u1 tag;

u2 string\_index;

}

Tag是一个数字，用来表示存储的常量类型。如8表示的String类型，5表示的是Long类型，info[]根据类型吗tag的不同会发生相应变化。

对于String 类型，表的结构为

CONSTANT\_String\_info{

u1 tag;

u2 string\_index；

}

Tag固定为8时，sting\_index是字符串内容信息，其类型为：

CONSTANT\_Utf8\_info{

u1 tag;

u2 length;

u3 bytes[length]

}

Tag固定为1时，length为字符串的长度，bytes[length]为字符串内容。为了详细理解常量池的结构请看如下代码：  
public class Test{

Public static void main(String[] args){

String s1 = “aaa111”;

String s2 = “aaa111”;

}

}

在上面的程序中存在两个相同的常量“aaa111”，对于相同的String常量，在Contstant Pool中只会创建一个，所以在编译好的class 文件中，我们只能找到一个对”aaa111”的表示。

在程序执行的时候，Constant Pool 会储存在Method Area中，而不是堆中。另外，对于“”内容为空的字符串常量，会创建一个长度为0、内容为空的字符串放到Contstant Pool中，而且Contstant Pool在运行期是可以动态扩展的。

#### 扩展知识2：关于String类的说明

String 使用private final char value[] 来实现字符串的储存，也就是说，String对象创建之后，就不能再修改此对象中储存的字符串内容，就是因为如此才说String 类型是不可变的。

String有一个特殊的创建方法，就是通过使用“”双引号来创建，例如new String（“str”）实际上是创建了两个String对象，一个是“str”通过“”双引号创建的，另一个是通过new 创建的。只不过他们创建的时期不同，一个是编译期，另一个是运行期。

Java对String类型重载了+号操作符，可以直接使用 + 对两个字符串进行连接

运行期间调用String类的intern()方法可以向String Pool中动态添加对象(并且intern方法会返回String pool对于该常量的引用)

String的创建方法一般有如下几种：  
1、直接使用“” 号创建  
2、使用new String（“”）创建  
3、使用new String（“啦啦啦”）创建以及其他的一些重载构造函数创建。  
4、使用重载的字符串连接符“+”创建

#### 关于String扩展知识的示例

例1：  
String s1 = “aaa777”;  
String s2 = “aaa777”;  
System.out.println(s1 == s2);//结果为true

解释:  
“aaa777”是编译期常量，编译时已经能确定它的值，在编译好的class文件中，它已经在String Pool中了，此语句会在String Pool中查找等于“aaa77”的字符串(用equals(Object)方法确定)，如果存在，就把引用返回赋值给s1；若不存在就会创建一个“aaa777”放在String Pool中，然后把引用返回，赋值给s1。  
由于String Pool只会维护一个值相同的对象，上面两句的得到的引用是String Pool中的同一个对象，所以它们的值相等

例2：  
String s1 = new String（“aaa777”）；  
String s2 = “aaa777”；  
System.out.println(s1 == s2);//结果为false

解释：  
在java中，使用new关键字会创建一个新的对象，在本例中，不管在String Pool中是否已经有值相同的对象，都会创建一个新的String对象储存在heap中，然后把引用返回，赋值给s1.本例中使用了 String 的public String（String original）构造函数。  
由于s1是用new创建出的新的对象，储存在heap中，s2指向的对象储存在String Pool中，它们肯定不是同一个对象，只是储存的字符串的值相同。所以返回flase

例3：  
String s1 = new String（“aaa777”）；  
s1 = s1.intern();  
String s2 = “aaa777”；  
System.out.println(s1 == s2);//结果为true

解释：  
当调用intern()方法时，如果String Pool中已经包含一个等于此String对象的值的字符串(用equals(Object)方法确定),就把引用返回赋值给s1；若不存在就会创建一个“aaa777”放在String Pool中，然后把引用返回，赋值给s1,这样就会使s1指向String Pool中值为“aaa777”的字符串对象，s2也指向了该对象所以结果为true。

例4：  
String s1 = new String("777");  
String s2 = "aaa777";  
String s3 = "aaa" + "777";  
String s4 = "aaa" + s1;  
System.out.println(s2 == s3);//结果为true  
System.out.println(s2 == s4);//结果为false  
System.out.println(s2 == s4.intern());//结果为true

解释：  
由于进行连接的两个字符串都是常量编译期就能确定链接之后的值了，编译器会进行优化，直接将其表示为“aaa777”储存到String pool中。由于上边的s2 = “aaa777”已经在String Pool中加入了“aaa777”，此句会把s3指向和s2相同的对象，所以它们引用相同。此时仍然会创建出“aaa” 和 “777”两个常量，存储到String Pool中。

由于s1是变量，在编译期间不能确定它的值是多少，所以会在执行的时候创建一个新的String对象存储到heap中，然后赋值给s4。

#### substring(index，end\_index)工作方式

例1：  
String s1 = "abcdefg";  
String s1\_str = s1.substring(3, 5);  
System.out.println("s1\_str = " + s1\_str);//输出de

解释：  
 substring是前包括后不包括的。

#### substring内存泄露问题(在1.6版本上)

**示例**：  
public class TestGC {  
 private String largeString = new String(new byte[100000]);  
 String getString() {  
 return this.largeString.substring(0, 2);  
 }  
 public static void main(String[] args) {  
 java.util.ArrayList list = new java.util.ArrayList();  
 for (int i = 0; i < 1000000; i++) {  
 TestGC gc = new TestGC();  
 list.add(gc.getString());  
 }  
 }  
}

然而上面的代码，只要使用Java 6 （Java 7和8 都不会抛出异常）运行一下就会报java.lang.OutOfMemoryError: Java heap space的异常，这说明没有足够的堆内存供我们创建对象，JVM选择了抛出异常操作。

**修改后代码:**  
public class TestGC {  
 private String largeString = new String(new byte[100000]);  
 String getString() {  
 //return this.largeString.substring(0,2);  
 return new String("ab");  
 }  
 public static void main(String[] args) {  
 java.util.ArrayList list = new java.util.ArrayList();  
 for (int i = 0; i < 1000000; i++) {  
 TestGC gc = new TestGC();  
 list.add(gc.getString());  
 }  
 }  
}

执行上面的方法，**并不会导致OOM异常**，因为我们持有的时1000000个ab字符串对象，而TestGC对象（包括其中的largeString）会在java的垃圾回收中释放掉。所以这里不会存在内存溢出。

**深入Java 6实现:**在String类中存在这样三个属性  
value 字符数组，存储字符串实际的内容  
offset 该字符串在字符数组value中的起始位置  
count 字符串包含的字符的长度  
Java 6中substring的实现  
public String substring(int beginIndex, int endIndex) {  
 if (beginIndex < 0) {  
 throw new StringIndexOutOfBoundsException(beginIndex);  
 }  
 if (endIndex > count) {  
 throw new StringIndexOutOfBoundsException(endIndex);  
 }  
 if (beginIndex > endIndex) {  
 throw new StringIndexOutOfBoundsException(endIndex - beginIndex);  
 }  
 return ((beginIndex == 0) && (endIndex == count)) ? this :  
 new String(offset + beginIndex, endIndex - beginIndex, value);  
 }

上述方法调用的构造方法

//Package private constructor which shares value array for speed.

String(int offset, int count, char value[]) {

  this.value = value;

  this.offset = offset;

  this.count = count;

}

当我们调用字符串a的substring得到字符串b，其实这个操作，无非就是调整了一下b的offset和count，用到的内容还是a之前的value字符数组，并没有重新创建新的专属于b的内容字符数组。

举个和上面重现代码相关的例子，比如我们有一个1G的字符串a，我们使用substring(0,2)得到了一个只有两个字符的字符串b，如果b的生命周期要长于a或者手动设置a为null，当垃圾回收进行后，a被回收掉，b没有回收掉，那么这1G的内存占用依旧存在，因为b持有这1G大小的字符数组的引用。

共享内容字符数组

其实substring中生成的字符串与原字符串共享内容数组是一个很棒的设计，这样避免了每次进行substring重新进行字符数组复制。正如其文档说明的,共享内容字符数组为了就是速度。但是对于本例中的问题，共享内容字符数组显得有点蹩脚。

**如何解决**

对于之前比较不常见的1G字符串只截取2个字符的情况可以使用下面的代码，这样的话，就不会持有1G字符串的内容数组引用了。

|  |  |
| --- | --- |
|  | String littleString = new String(largeString.substring(0,2)); |

**Java 7 实现**

在Java 7 中substring的实现抛弃了之前的内容字符数组共享的机制，对于子字符串（自身除外）采用了数组复制实现单个字符串持有自己的应该拥有的内容。

public String substring(int beginIndex, int endIndex) {

if (beginIndex < 0) {

      throw new StringIndexOutOfBoundsException(beginIndex);

    }

    if (endIndex > value.length) {

      throw new StringIndexOutOfBoundsException(endIndex);

    }

    int subLen = endIndex - beginIndex;

    if (subLen < 0) {

      throw new StringIndexOutOfBoundsException(subLen);

    }

    return ((beginIndex == 0) && (endIndex == value.length)) ? this

                : new String(value, beginIndex, subLen);

}

substring方法中调用的构造方法，进行内容字符数组复制。

public String(char value[], int offset, int count) {

    if (offset < 0) {

          throw new StringIndexOutOfBoundsException(offset);

    }

    if (count < 0) {

      throw new StringIndexOutOfBoundsException(count);

    }

    // Note: offset or count might be near -1>>>1.

    if (offset > value.length - count) {

      throw new StringIndexOutOfBoundsException(offset + count);

    }

**this.value = Arrays.copyOfRange(value, offset, offset+count);**

}

关于Java 7 对substring做的修改，收到了褒贬不一的反馈。

个人更加倾向于Java 6的实现，当进行substring时，使用共享内容字符数组，速度会更快，不用重新申请内存。虽然有可能出现本文中的内存性能问题，但也是有方法可以解决的。

Java 7的实现不需要程序员特殊操作避免了本文中问题，但是进行每次substring的操作性能总会比java 6 的实现要差一些。这种实现显得有点“糟糕”。

#### 常量池会被gc回收么

答：  
被final或者static声明的常量不会被gc函数回收，程序运行就一直存在于堆栈中。其余的一些类的变量，在运行时被创建，在必要时会被回收

#### String 和 StringBuffer 效率比较

例1：  
(1)String result1 = "hello"+"word";  
(2)StringBuffer result2 = new StringBuffer().append("hello").append("word");

解释：  
(1）的效率要比(2)的效率要好因为jvm会进行如下的处理。  
。将result1字符串进行“hello”+”word”；处理然后才赋值给result1.只开辟了一段内存  
。编译StringBuffer后还要进行append处理，花的时间要长一些

例2：  
public String getString(String s1, String s2) {  
 return s1 + s2;  
}  
public String getString(String s1, String s2) {  
 return new StringBuffer().append(s1).append(s2);  
}

解释:  
(1)与(2)的效率一样，这是因为JVM会进行如下处理：  
开辟一个新的内存段，在合并(扩展内存)，所以两者执行过程是一致的效率相当。

例3：  
(1)String s = ”s1”;  
 s+=”s2”;  
 s+=”s3”;  
(2)StringBuffer s = new StringBuffer().append(“s1”).append(“s2”).append(“s3”):

解释：  
 (2)的效率要好于(1)，因为String 是不可变对象，每次+=操作都会创建新的String 对象

例4：  
(1)StringBuffer s = new StringBuffer();  
for(int i=0;i<5000;i++){  
 s.append(“hello”);  
}

(2)StringBuffer s = new StringBuffer(250000);  
for(int i=0;i<5000;i++){  
 s.append(“hello”);  
}

解释：  
(2)的效率要稍好于(1)但是初始占用的内存更大。因为StringBuffer内部实现的char数组，默认初始化长度为16，每当字符串长度大于char数组长度的时候，jvm会构造更大的新数组，并将原先的数组内容复制到新数组。

#### StringBuffer面试关键点

1. 简单的认为append()效率要好于+是错误的
2. 尽量不要使用new 来构建String 除非是解决更严重的问题
3. 注意intern()方法的使用
4. 在编译期能确定字符串的值的情况下使用+的效率最高
5. 避免使用+=来构造字符串(会造成更大的内存消耗和gc的负担)
6. 在声明StringBuffer的时候最好指定合适的capacity(长度)，不要使用默认值16
7. 注意一下二者的区别，后者开辟了两个内存段  
   1.String s = ”a” + “b”;  
   2.String s = “a”;  
    s+=”b”;

#### 下面创建了几个对象

面试题：  
String A,B,C;  
A =”a”;  
B = “b”;  
A = A+B;  
StringBuffer D = new StringBuffer(“abc”);  
D = D.append(“567”);

A.4 B.3 C.5 D.6

解析：

要知道String s = new String(“abc”)创建了几个String Objec，首先必须引用变量与对象的区别。

(1)引用变量与对象。除了一些早期的java书籍，我们都可以从书中比较清楚的学习到两者的区别“A aa”语句声明一个类A的引用变量aa (常称为句柄)，而对象一般通过new 创建。所以题目中D仅仅是一个引用变量，它不是对象。而字符串文字”abc”才是一个String对象。  
(2)Java中所有的字符串文字(字符串常量)都是一个String的对象。有人喜欢把字符串当作字符数组，但它与字符数组是两种完全不同的对象，虽然它们有一些内在的联系。

(3)字符串对象的创建。由于字符串对象的大量使用(它是一个对象，一般而言对象总在heap上分配内存)，java中为了节省内存空间和运行时间，在编译阶段就把所有的字符串文字放到一个字符串常量池里，而字符串常量池是常量池的一部分。字符串常量池的好处就是该池中所有相同的字符串常量被合并，只占用一个空间。对两个引用变量使用 == 判断它们的值是否相等，即是指向同一个对象。现在看String s = new String(“abc”)语句这里”abc”本身就是pool中的一个对象，而在运行时执行new String()时，将pool中的对象复制一份放到heap中，并且把heap中的这个对象的引用交给s持有。这条语句就创建了两个对象。  
String s1 = new String（“abc”）；

String s2 = new String（“abc”）；  
if（s1 == s2）{

//不会执行语句  
}

这时用 == 判断就可知，虽然两个对象的内容相同(用equals()判断)，但两个引用变量所持有的引用不同。上面的代码创建了String Object pool中一个heap中两个。

下面对提目中的代码一一解析：

String A,B,C;

A =”a”;//此处创建了一个对象(“a”)和一个引用A

B = “b”; //此处创建了一个对象(“b”)和一个引用B

A = A+B;

StringBuffer D = new StringBuffer(“abc”); //此处创建了一个对象(“abc”)和一个引用D

D = D.append(“567”); //此处创建了一个对象(“567”),append()都是对同一个对象进行处理

所以该段代码共产生了五个对象

# 传递与引用

### 传值与传引用

#### 值传递和引用传递的概念？

答：  
java中的形参是复制实参的一份拷贝（对于引用型则是复制引用的拷贝）（在栈中的拷贝），所以在函数中改变形参是无法改变实参的值的，改变引用只是将形参所代表的引用指向另外的新的对象，而实参的引用还指向原来的对象，改变形参引用的成员当然会影响实参引用成员的值，因为他们的引用都指向同一个对象。

#### 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。

# 面向对象