内部类和静态内部类的区别

内部类：

1、内部类中的变量和方法不能声明为静态的。
2、内部类实例化：B 是 A 的内部类，实例化 B：A.B b = new A().new B()。
3、内部类可以引用外部类的静态或者非静态属性及方法。

静态内部类：

1、静态内部类属性和方法可以声明为静态的或者非静态的。
2、实例化静态内部类：B 是 A 的静态内部类，A.B b = new A.B()。
3、静态内部类只能引用外部类的静态的属性及方法。

inner classes——内部类
static nested classes——静态嵌套类
其实人家不叫静态内部类，只是叫习惯了，从字面就很容易理解了。
内部类依靠外部类的存在为前提，而静态嵌套类则可以完全独立，明白了这点就很好理解了。

非静态内部类中的变量和方法不能声明为静态的原因

静态类型的属性和方法，在类加载的时候就会存在于内存中。使用某个类的静态属性和方法，那么这个类必须要加载到虚拟机中。但是非静态内部类并不随外部类一起加载，只有在实例化外部类之后才会加载。

我们设想一个场景：在外部类并没有实例化，内部类还没有加载的时候如果调用内部类的静态成员或方法，内部类还没有加载，却试图在内存中创建该内部类的静态成员，就会产生冲突。所以非静态内部类不能有静态成员变量或静态方法。

String，StringBuilder，StringBuffer 的区别

String 字符串常量
StringBuffer 字符串变量（线程安全）
StringBuilder 字符串变量（非线程安全）

性能上通常 StringBuilder > StringBuffer > String。

String 是不可变对象，每次对 String 类型进行改变的时候都等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象，所以性能最差，对于要经常改变内容的字符串不用 String。

StringBuffer 是字符串变量，对它操作时，并不会生成新的对象，而是直接对该对象进行更改，所以性能较好。

StringBuilder 和 StringBuffer 一样，是字符串变量，但是他不带有 synchronized 关键字，不保证线程安全，所以性能最好。在单线程的情况下，建议使用 StringBuilder。

总体来说：
String：适用于少量的字符串操作的情况。
StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况。
StringBuffer：适用多线程下在字符缓冲区进行大量操作的情况。

来一些问题：
下面这段代码的输出结果是什么？

```
String a = "helloworld";
String b = "hello" + "world";
System.out.println((a == b));
```

输出结果为：True。
原因是 String 对字符串的直接相加，会在编译期进行优化。即 hello+world 在编译时期，被优化为 helloworld，所以在运行时期，他们指向了同一个对象。我们也可以推理，对于直接字符串的相加，String 不一定比其余两个慢。

下面这段代码的输出结果是什么？

```
String a = "helloworld";
String b = "hello";
String c = b + "world";
System.out.println((a == c));
```

输出结果为：False。
原因是 c 并非两个字符串直接相加，包含了一个字符串引用，这时不会做编译期的优化。所以 a、c 最终生成了两个对象，这时他的效率低。

集合和数组之间的相互转换

数组变集合：

通常我们会回答的是以下代码：

```
List<String>  list  = Arrays.asList(array);
```

但这并不是很好的答案，此时组合成的 list 是 Arrays 里面的一个静态内部类，该类并未实现 add、remove 方法，因此在使用时存在问题。

可以这样：

```
String  array[]= {"hello","world","java","zhiyin"};
List<String>  list  =  new  ArrayList<String>(Arrays.asList(array));
```

集合变数组：

```
String[]  array=list.toArray(new  String[list.size()]);
```

面向对象的特征有哪些方面？

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

- **继承**：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类；得到继承信息的类被称为子类。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段。

- **封装**：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口。

- **多态性**：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。方法重载实现的是编译时的多态性，而方法重写实现的是运行时的多态性。