* java中==和equals和hashCode的区别

1 “==”

在java中“==”是用来比较变量值是否相等。如果是基本类型，直接比较值。如果是对象类型，比较的是两个对象的引用，也就是地址。对象是放在堆中的，栈中存放的是对象的引用。“==”是对栈中的值进行比较的。

2 “equals”

我们都知道java所有的类都是集成自Object类，Object里有一个方法“equals”，这个方法是用来比较两个对象是否相等的。在Object类中有这样的代码：

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

说明在Object里"equals"和"=="是一回事。

3 “hashcode”

在Object里提供了hashcode这个方法。要说hashcode就得说java集合。java有的集合是不能重复的，所以需要用equeals判断集合中元素是否是同一个。但是如果集合中现在已经有1000个元素，那么第1001个元素加入集合时，它就要调用1000次equals方法。这显然会大大降低效率。于是，Java采用了哈希表的原理。哈希（Hash）实际上是个人名，由于他提出一哈希算法的概念，所以就以他的名字命名了。哈希算法也称为散列算法，是将数据依特定算法直接指定到一个地址上。可以说hashCode方法实际上返回的就是对象存储的物理地址（实际可能并不是）。这样一来，当集合要添加新的元素时，先调用这个元素的hashCode方法，就一下子能定位到它应该放置的物理位置上。如果这个位置上没有元素，它就可以直接存储在这个位置上，不用再进行任何比较了；如果这个位置上已经有元素了，就调用它的equals方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址。

关系

首先说“==”和“equals”。  
先明白一个事情：如果类没有重写equals，那么对于该类的对象来说“==”和“equals”没有区别。都是比较对象的内存地址。 但在一些类库当中这个方法被覆盖掉了，如String,Integer,Date在这些类当中equals有其自身的实现，而不再是比较类在堆内存中的存放地址了。所以会有很多经典的面试题：

String a = new String ("abc");  
String b = new String ("abc");  
System.out.println(a.equals(b));  
System.out.println(a==b);

结果大家都知道 是true和false，其实这才是特殊情况。并不能得出一般的结论而笼统的说：”equals比较对象值，'=='比较地址“。要看是否重写了equals方法。再说equals和hashCode之间的关系。首先hashCode存在就是为了提高效率并且辅助equals的，一般重写equals要同时重写hashCode。java中这样规定他们的关系：1、如果两个对象相同，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 ，上面说的对象相同指的是用eqauls方法比较。反过来：hashcode()不等，一定能推出equals()也不等；hashcode()相等，equals()可能相等，也可能不等。 如果重写这两个方法最好遵循以上原则。所以比较两者还要看具体是如何重写的。

特殊情况

说一种特殊情况：

String s1 = "Lpnpcs";  
String s2 = "Lpnpcs";  
if (s1 == s2) {  
System.out.println("s1 == s2");  
} else{  
System.out.println("s1 != s2");  
}输出s1==s2；  
String s1 = "Lpnpcs";  
String s2 = new String("Lpnpcs");  
if (s1 == s2)  
{System.out.println("s1 == s2");}  
else  
{System.out.println("s1 != s2");}  
if (s1.equals(s2)) {System.out.println("s1 equals s2");}  
else{  
System.out.println("s1 not equals s2");  
}

输出s1 != s2 s1 equals s2说明：s1 s2分别引用了两个"Lpnpcs"String对象。这是什么情况？ 这是由于java中对于字符串定义了一个字符串缓冲池，程序在运行的时候会创建一个字符串缓冲池当使用 s2 = "Lpnpcs" 这样的表达是创建字符串的时候，程序首先会在这个String缓冲池中寻找相同值的对象，在第一个程序中，s1先被放到了池中，所以在s2被创建的时候，程序找到了具有相同值的 s1将s2引用s1所引用的对象"Lpnpcs"。第二段程序中，使用了 new 操作符，他明白的告诉程序："我需要新建一个新的"于是一个新的"Lpnpcs"String对象被创建在内存中。他们的值相同，但是位置不同。所以可见java定义这个缓冲池就是为了节约资源。我们在用字符串的时候 尽量采用 ：String a =“”；这种形式。以上就是他们的实现和原理，相信现在应该很清楚了。

* int、char、long各占多少字节数

4,2,8

* int与integer的区别

1、Integer是int提供的封装类，而int是Java的基本数据类型；

2、Integer默认值是null，而int默认值是0；

3、声明为Integer的变量需要实例化，而声明为int的变量不需要实例化；

4、Integer是对象，用一个引用指向这个对象，而int是基本类型，直接存储数值。

* 探探对java多态的理解

1.面向对象的三大特性：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。这是我们最后一个概念，也是最重要的知识点。

2.多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）

3.实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

4.多态的作用：消除类型之间的耦合关系。

5.现实中，关于多态的例子不胜枚举。

下面是多态存在的三个必要条件，要求大家做梦时都能背出来！

多态存在的三个必要条件

一、要有继承；

二、要有重写；

三、父类引用指向子类对象。

* String、StringBuffer、StringBuilder区别

三者在执行速度方面的比较：StringBuilder >  StringBuffer  >  String

String <（StringBuffer，StringBuilder）的原因

String：字符串常量

StringBuffer：字符创变量

StringBuilder：字符创变量

从上面的名字可以看到，String是“字符创常量”，也就是不可改变的对象。对于这句话的理解你可能会产生这样一个疑问  ，比如这段代码：

1 String s = "abcd";

2 s = s+1;

3 System.out.print(s);// result : abcd1

我们明明就是改变了String型的变量s的，为什么说是没有改变呢? 其实这是一种欺骗，JVM是这样解析这段代码的：首先创建对象s，赋予一个abcd，然后再创建一个新的对象s用来执行第二行代码，也就是说我们之前对象s并没有变化，所以我们说String类型是不可改变的对象了，由于这种机制，每当用String操作字符串时，实际上是在不断的创建新的对象，而原来的对象就会变为垃圾被ＧＣ回收掉，可想而知这样执行效率会有多底。

而StringBuffer与StringBuilder就不一样了，他们是字符串变量，是可改变的对象，每当我们用它们对字符串做操作时，实际上是在一个对象上操作的，这样就不会像String一样创建一些而外的对象进行操作了，当然速度就快了。

一个特殊的例子：

String str = “This is only a” + “ simple” + “ test”;

StringBuffer builder = new StringBuilder(“This is only a”).append(“simple”).append(“ test”);

你会很惊讶的发现，生成str对象的速度简直太快了，而这个时候StringBuffer居然速度上根本一点都不占优势。其实这是JVM的一个把戏，实际上：

String str = “This is only a” + “ simple” + “test”;

其实就是：

String str = “This is only a simple test”;

所以不需要太多的时间了。但大家这里要注意的是，如果你的字符串是来自另外的String对象的话，速度就没那么快了，譬如：

String str2 = “This is only a”;

String str3 = “ simple”;

String str4 = “ test”;

String str1 = str2 +str3 + str4;

这时候JVM会规规矩矩的按照原来的方式去做。

4.StringBuilder与 StringBuffer：

StringBuilder：线程非安全的

StringBuffer：线程安全的

当我们在字符串缓冲去被多个线程使用是，JVM不能保证StringBuilder的操作是安全的，虽然他的速度最快，但是可以保证StringBuffer是可以正确操作的。当然大多数情况下就是我们是在单线程下进行的操作，所以大多数情况下是建议用StringBuilder而不用StringBuffer的，就是速度的原因。

对于三者使用的总结：

如果要操作少量的数据用 = String

单线程操作字符串缓冲区 下操作大量数据 = StringBuilder

多线程操作字符串缓冲区 下操作大量数据 = StringBuffer

* 什么是内部类？内部类的作用

内部类可以直接访问外部类中的成员(包括私有成员),  
而外部类要访问内部类中的成员必须要建立内部类的对象

* 抽象类和接口区别

设计思想不同,使用动机不同.

抽象类和接口设计的时候,设计思想不同. 设计抽象类是自下而上的过程,因为调用者子类需要某些属性和特有行为,所以调用者继承抽象类 设计接口是自上而下的过程,接口规范某一行为,我某类需要这个行为,调用者实现某接口

开发者使用的时候,使用动机不同. 开发者继承抽象类是为了使用抽象类的属性和行为; 开发者实现接口只是为了使用接口的行为.

区别总结:

区别一: 抽象类只能单继承,接口能多实现

区别二: 抽象类是一个类,可以被任意权限修饰符修饰,静态和非静态属性，final和非final属性，可以有抽象方法和非抽象方法；

接口只能被public,final修饰,只能有静态方法,即使没有显示的声明，而且是不可修改的；

区别三: 抽象的事物不同: 抽象类是对类的抽象,接口是对行为的抽象； 抽象类是对整个类抽象,包括属性,行为； 接口是对类的行为(局部)进行抽象；

区别四: 定义的时候,定义抽象类和接口的思想不同； 设计抽象类是自下而上的过程,我子类需要,所以我定义抽象类； 设计接口是自上而下的过程,我接口规范某一行为,我某类需要这个行为,我实现某接口；

核心区别:

调用者使用动机不同,实现接口是为了使用他规范的某一个行为；

继承抽象类是为了使用这个类属性和行为.

再简单点说:

我们知道抽象类是从子类中发现公共部分，然后泛化成抽象类，子类继承该父类即可，但是接口不同。实现它的子类可以不存在任何关系，共同之处。例如猫、狗可以抽象成一个动物类抽象类，具备叫的方法。鸟、飞机可以实现飞Fly接口，具备飞的行为，这里我们总不能将鸟、飞机共用一个父类吧！所以说抽象类所体现的是一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在"[is](https://link.zhihu.com/?target=http%3A//www.mydown.com/soft/network/chat/475/444475.shtml)-a" 关系，即父类和派生类在概念本质上应该是相同的。对于接口则不然，并不要求接口的实现者和接口定义在概念本质上是一致的， 仅仅是实现了接口定义的规则而已。

* 抽象类的意义
* 抽象类与接口的应用场景

interface的应用场合

类与类之前需要特定的接口进行协调，而不在乎其如何实现。

作为能够实现特定功能的标识存在，也可以是什么接口方法都没有的纯粹标识。

需要将一组类视为单一的类，而调用者只通过接口来与这组类发生联系。

需要实现特定的多项功能，而这些功能之间可能完全没有任何联系。

abstract class的应用场合

一句话，在既需要统一的接口，又需要实例变量或缺省的方法的情况下，就可以使用它。最常见的有：

定义了一组接口，但又不想强迫每个实现类都必须实现所有的接口。可以用abstract class定义一组方法体，甚至可以是空方法体，然后由子类选择自己所感兴趣的方法来覆盖。

某些场合下，只靠纯粹的接口不能满足类与类之间的协调，还必需类中表示状态的变量来区别不同的关系。abstract的中介作用可以很好地满足这一点。

规范了一组相互协调的方法，其中一些方法是共同的，与状态无关的，可以共享的，无需子类分别实现；而另一些方法却需要各个子类根据自己特定的状态来实现特定的功能

如果你拥有一些方法并且想让它们中的一些有默认实现，那么使用抽象类吧。  
如果你想实现多重继承，那么你必须使用接口。  
由于Java不支持多继承，子类不能够继承多个类，但可以实现多个接口。因此你就可以使用接口来解决它。  
如果基本功能在不断改变，那么就需要使用抽象类。如果不断改变基本功能并且使用接口，那么就需要改变所有实现了该接口的类。

抽象类表示共有许要实现的方法 接口用来特定类有需要才实现 人都会吃饭 但有的人抽烟

抽象类是用来捕捉子类的通用特性的

抽象类是 is a关系；而接口是has a关系

看看jdk的api会发现大部分接口都是able结尾的，表示一种能力。马士兵曰过：当你不知道用抽象类还是接口好时就用接口

* 抽象类是否可以没有方法和属性

是有抽象属性或者抽象方法的必须为抽象类，相反不一定

* 泛型中extends和super的区别

<? extends T>和<? extends > 是Java泛型中的“**通配符（Wildcards）**”和“**边界（Bounds）**”的概念。

<? extends T>：是指“上界通配符（Upper Bounds Wildcards”

<? extends >： 是指“下界通配符（Lower Bounds Wildcards）”

1. 为什么要用通配符和边界？

使用泛型的过程中，经常出现一种很别扭的情况。比如我们有Fruit类和它的派生类Apple类。

class Fruit { }

class Apple extends Fruit { }

然后有一个最简单的容器：Plate类。盘子里可以放一个泛型的“东西”。我们可以对这个东西做最简单的“放”和“取”的动作：set()和get()方法。

class Plate<T> {

private T item;

public Plate(T t) {

item = t;

}

public void set(T t) {

item = t;

}

public T get() {

return item;

}

}

现在我们可以定义一个“水果盘子”，逻辑上水果盘子当然可以装苹果。

Plate<Fruit> p = new Plate<Apple>(new Apple());

但实际上Java编译器不允许这个操作，会报错。

Type mismatch: cannot convert from Plate<Apple> to Plate<Fruit>

实际上，编译器认定的逻辑是这样的：

苹果is-a水果

装苹果的盘子not-is-a装水果的盘子

所以，就算容器里装的东西之间有继承关系，但**容器之间是没有继承关系的**。所以我们不可以把Plate<Apple>的引用传递给Plate<Fruit>。

为了让泛型用起来更舒服，于是就有了<? extends T>和< super T>的办法，来让“水果盘子”和“苹果盘子”有联系。

2 什么是上界

下面代码就是“**上界通配符**”：

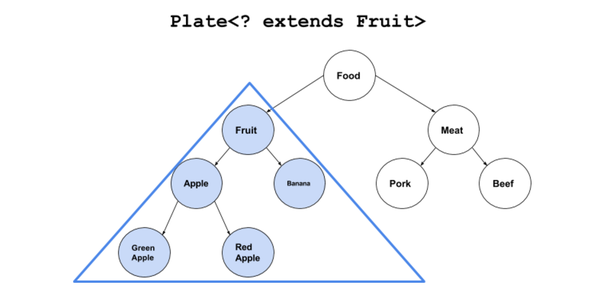
Plate<? extends Fruit>

意思是：**一个能放水果以及一切是水果派生类的盘子**。这和我们人类的逻辑就比较接近了。Plate<? extends Fruit>和Plate<Apple>最大的区别就是：**Plate<? extends Fruit>是Plate<Fruit>以及Plate<Apple>的基类**。直接的好处就是，我们可以用“苹果盘子”给“水果盘子”赋值。

Plate<? extends Fruit> p = new Plate<Apple>(new Apple());

如果把Fruit和Apple的例子再扩展一下，食物分成水果和肉类，水果还有苹果和香蕉，肉类有猪肉和牛肉，苹果还有两种青苹果和红苹果。

在这个继承体系中，下界通配符Plate<? extends Fruit> 覆盖下图中蓝色的区域。



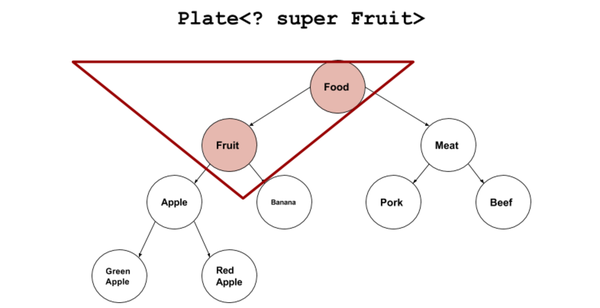
继承体系图

3. 什么是下界

相应的，“下界通配符”：

Plate<? super Fruit>

表达的就是相反的概念：一个能放水果以及一切是水果基类的盘子。Plate<?super Fruit>是Plate<Fruit>的基类，但不是Plate<Apple>的基类。对应刚才那个例子，Plate<? super Fruit>覆盖下图中红色的区域：



继承体系图

4. 上下界通配符的副作用

边界让Java不同泛型之间的转换更容易了。但不要忘记，这样的转换也有一定的副作用。那就是**容器的部分功能可能失效**。

还是以刚才的Plate为例。我们可以对盘子做两件事，往盘子里set( )新东西，以及从盘子里get()东西。

class Plate<T> {

private T item;

public Plate(T t) {

item = t;

}

public void set(T t) {

item = t;

}

public T get() {

return item;

}

}

4.1 上界<? extends T>不能往里存，只能往外取

<? extends Fruit>会使往盘子里放东西的set( )方法失效。但取东西get( )方法还有效。比如下面例子里两个set()方法，插入Apple和Fruit都报错。

Plate<? extends Fruit> p = new Plate<Apple>(new Apple());

//不能存入任何元素

p.set(new Fruit());//Error

p.set(new Apple());//Error

//读取出来的东西只能存放在Fruit或它的基类里

Fruit other1 = p.get();

Object other2 = p.get();

Apple other3 = p.get();//Error

原因是编译器只知道容器内是Fruit或者它的派生类，但具体是什么类型不知道。可能是Fruit？可能是Apple？也可能是Banana，RedApple，GreenApple？编译器在看到后面用Plate<Apple>赋值以后，盘子里没有被标上“苹果”。而是标上一个占位符：CAP#1，来表示捕获一个Fruit或Fruit子类，具体是什么类不知道，代号CAP#1.然后无论是想往里插入Apple或者Meat或者Fruit编译器都不知道能不能和这个CAP#1匹配，所以就都不允许。

所以通配符<?>和类型参数<T>的区别就在于，对编译器来说所有的T都代表同一种类型。比如下面这个泛型方法里，三个T都指代同一个类型，要么都是String，要么都是Integer。

public <T> List<T> fill(T... t);

但通配符<?>没有这种约束，Plate<?>单纯的就表示：盘子里放了一个东西，是什么我不知道。

所以Plate<？ extends Fruit>里什么都放不进去。

4.2 下界<? super T>不影响往里存，但往外取只能放在Object对象里

Plate<? super Fruit> p = new Plate<Apple>(new Apple());

//存入元素正常

p.set(new Fruit());

p.set(new Apple());

//读取出来的东西只能存放在Object类里

Fruit other1 = p.get();//Error

Apple other2 = p.get();//Error

Object other3 = p.get();

使用下界<? super Fruit>会使从盘子里取东西的get( )方法部分失效，只能存放到Object对象里。set( )方法正常。

因为下界规定了元素的最小粒度的下限，实际上是放松了容器元素的类型控制。既然元素是Fruit的基类，那往里存粒度比Fruit小的都可以。但往外读取元素就费劲了，只有所有类的基类Object对象才能装下。但这样的话，元素的类型信息就全部丢失。

5. PECS原则

1) T<? super B>

对于这个泛型，?代表容器里的元素类型，由于只规定了元素必须是B的超类，导致元素没有明确统一的“根”（除了Object这个必然的根），所以这个泛型你其实无法使用它，对吧，除了把元素强制转成Object。所以，对把参数写成这样形态的函数，你函数体内，只能对这个泛型做插入操作，而无法读

2) T<? extends B>

由于指定了B为所有元素的“根”，你任何时候都可以安全的用B来使用容器里的元素，但是插入有问题，由于供奉B为祖先的子树有很多，不同子树并不兼容，由于实参可能来自于任何一颗子树，所以你的插入很可能破坏函数实参，所以，对这种写法的形参，禁止做插入操作，只做读取

最后看一下什么是PECS（Producer Extends Consumer Super）原则，已经很好理解了：

**频繁往外读取内容的，适合用上界Extends**。

**经常往里插入的，适合用下界Super**。

* 父类的静态方法能否被子类重写

不能

因为静态方法从程序开始运行后就已经分配了内存，也就是说已经写死了。所有引用到该方法的对象（父类的对象也好子类的对象也好）所指向的都是同一块内存中的数据，也就是该静态方法。子类中如果定义了相同名称的静态方法，并不会重写，而应该是在内存中又分配了一块给子类的静态方法，没有重写这一说

* 进程和线程的区别

进程和线程的关系

（1）一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。

（2）资源分配给进程，同一进程的所有线程共享该进程的所有资源。

（3）处理机分给线程，即真正在处理机上运行的是线程。

（4）线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的办法实现同步。线程是指进程内的一个执行单元,也是进程内的可调度实体.

进程与线程的区别

（1）调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位。

（2）并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可并发执行。

（3）拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源。

（4）系统开销：在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销。

* final，finally，finalize的区别

1、final 修饰符（关键字）

  final用于控制成员、方法或者是一个类是否可以被重写或者继承等功能，如果要学好java，那么这个关键字很重要，必须掌握。如果类被声明为final，意味着它不能再派生出新的子类，不能作为父类被继承。将变量或者方法声明为final，可以保证他们在使用中不被改变。其初始化可以在两个地方：一是其定义处，也就是说，在final变量定义时直接给其赋值；二是构造函数中。这2个地方只能选其一，要么在定义处直接给其赋值，要么在构造函数中给值，并且在以后的引用中，只能读取，不可修改。被声明为final的方法也同样只能使用，不能重写。

2、finally(用于异常处理)

    一般是用于异常处理中，提供finally块来执行任何的清楚操作，try{} catch(){} finally{}。finally关键字是对java异常处理模型的最佳补充。finally结构使代码总会执行，不关有无异常发生。使得finally可以维护对象的内部状态，并可以清理非内存资源。

finally在try,catch中可以有，可以没有。如果trycatch中有finally则必须执行finally快中的操作。一般情况下，用于关闭文件的读写操作，或者是关闭数据库的连接等等。

3、finalize（用于垃圾回收）

finalize这个是方法名。在java中，允许使用finalize()方法在垃圾收集器将对象从内存中清理出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用是对这个对象调用的。它是Object类中定义的，因此，所有的类都继承了它。finalize()方法是在垃圾收集器删除对象之前对这个对象调用的。

* 序列化的方式
* Serializable 和Parcelable 的区别
* 静态属性和静态方法是否可以被继承？是否可以被重写？以及原因？
* 静态内部类的设计意图

1）首先，用内部类是因为内部类与所在外部类有一定的关系，往往只有该外部类调用此内部类。所以没有必要专门用一个Java文件存放这个类。

2）静态都是用来修饰类的内部成员的。比如静态方法，静态成员变量，静态常量。它唯一的作用就是随着类的加载（而不是随着对象的产生）而产生，以致可以用类名+静态成员名直接获得。

这样静态内部类就可以理解了，因为这个类没有必要单独存放一个文件，它一般来说只被所在外部类使用。并且它可以直接被用 外部类名+内部类名 获得。

以下举例说明怎么使用：

Student类有个叫School的内部类（非静态）

Student stu = new Student();

stu.School sch = new stu.School();

sch就是School的一个对象。

假如School是内部静态类：

Student.School sch = new Student.School();

需要注意的地方：

内部类就像一个实例成员一样存在于外部类中。

内部类可以访问外部类的所有成员就想访问自己的成员一样没有限制。

内部类中的this指的是内部类的实例对象本身，如果要用外部类的实例对象就可以用类名.this的方式获得。

内部类对象中不能有静态成员，原因很简单，内部类的实例对象是外部类实例对象的一个成员。

* 成员内部类、静态内部类、局部内部类和匿名内部类的理解，以及项目中的应用

内部类，多重继承

class OuterClass {

...

static class StaticNestedClass {

...// 静态内部类

}

class InnerClass {

...// 内部类

}

function a(){

class PhoneNumber {

//局部内部类

}

}

Runnable r = new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

//匿名内部类

}

};

}

为什么匿名内部类和局部内部类引用外部的变量必要是final的呢？

“匿名内部类来自外部闭包环境的自由变量必须是final的”

外部环境如果持有内部函数所使用的自由变量，就会对内部函数形成“闭包”

https://zhuanlan.zhihu.com/p/29245059

* 闭包和局部内部类的区别

Ruby之父松本行弘在《代码的未来》一书中解释的最好：闭包就是把函数以及变量包起来，使得变量的生存周期延长。闭包跟面向对象是一棵树上的两条枝，实现的功能是等价的。

* string 转换成 integer的方式及原理

public static int parseInt(String s, int radix)  
  throws NumberFormatException  
    {  
        if (s == null) {  
            throw new NumberFormatException("null");  
        }

 if (radix < Character.MIN\_RADIX) {  
     throw new NumberFormatException("radix " + radix +  
         " less than Character.MIN\_RADIX");  
 }

 if (radix > Character.MAX\_RADIX) {  
     throw new NumberFormatException("radix " + radix +  
         " greater than Character.MAX\_RADIX");  
 }

 int result = 0;  
 boolean negative = false;  
 int i = 0, max = s.length();  
 int limit;  
 int multmin;  
 int digit;

 if (max > 0) {  
     if (s.charAt(0) == '-') {  
  negative = true;  
  limit = Integer.MIN\_VALUE;  
  i++;  
     } else {  
  limit = -Integer.MAX\_VALUE;  
     }

     if (i < max) {  
  digit = Character.digit(s.charAt(i++),radix);  
  if (digit < 0) {  
      throw NumberFormatException.forInputString(s);  
  } else {  
      result = -digit;  
  }  
     }  
     while (i < max) {  
  // Accumulating negatively avoids surprises near MAX\_VALUE  
  digit = Character.digit(s.charAt(i++),radix);  
  if (digit < 0) {  
      throw NumberFormatException.forInputString(s);   
  }  
  if (result < multmin) {  
      throw NumberFormatException.forInputString(s);  异常1  
  }  
  result \*= radix;  
  if (result < limit + digit) {  
      throw NumberFormatException.forInputString(s);  异常2  
  }  
  result -= digit;  
     }  
 } else {  
     throw NumberFormatException.forInputString(s);  
 }

 if (negative) {  
     if (i > 1) {  
  return result;  
     } else { /\* Only got "-" \*/  
  throw NumberFormatException.forInputString(s);  
     }  
 } else {  
     return -result;  
 }  
    }

很显然,该方法的第二个参数表示是基数(最常用的是十进制,还有十六机制,八进制等等).  
如果字符串是空指针,直接抛出异常.  
如果基础小于2或者大于36的话,抛出异常(这种情况一般不会出现,因为我们用的最多就是十进制的了).  
如果是空字符串,也抛出异常,也就是max=0的情况了.  
我们来关注下面的转换过程:  
这里使用了Character中的静态方法digit,这个方法比较复杂,这里先说明它的功能:对于给定的基数,如果是合法的字符(可以转化为数字),返回该数字值,否则返回-1.比如digit('3',10)返回3,digit('a',10)返回-1.

* 哪些情况下的对象会被垃圾回收机制处理掉？

https://www.zhihu.com/question/19912197

* 讲一下常见编码方式？

Unicode 做为一个国际标准，其实只是定义了每个字符对应的一个数字（实际上 Unicode 包含很多复杂的内容，这里只说最简单的部分），比如 20031 这个数字就代表“中”这个字，25991 代表“文”这个字。但其实 Unicode 并没有说你要怎么保存一个字。

ASCII 码, ISO-8859-1, GB2312, GBK, UTF-8是实现

* utf-8编码中的中文占几个字节；int型几个字节

中文占3或4个字节，数字占1个字节

* 静态代理和动态代理的区别，什么场景使用？

静态代理

在静态代理模式中，代理与被代理对象必须实现同一个接口，代理专注于实现日志记录需求，并在合适的时候，调用被代理对象，这样被代理对象就可以专注于执行业务逻辑。

改进上面那个例子  
首先定义一个接口

IHello.java

package Reflection;

public interface IHello {

public void hello(String name);

}

然后专注于业务逻辑实现HelloSpeaker实现上面这个接口：

HelloSpeaker.java

package Reflection;

public class HelloSpeaker implements IHello {

@Override

public void hello(String name) {

System.out.println("Hello, " + name);

}

}

可以看到，在这个类中没有日志记录的代码，其只需要专注于实现业务功能，而记录日志的工作则可以交给代理对象来实现，代理对象也要实现Ihello接口：

HelloProxy.java

package Reflection;

import java.util.logging.\*;

public class HelloProxy implements IHello {

private Logger logger =

Logger.getLogger(this.getClass().getName());

private IHello helloObject;

public HelloProxy(IHello helloObject) {

this.helloObject = helloObject;

}

public void hello(String name) {

// 日誌服務

log("hello method starts....");

// 執行商務邏輯

helloObject.hello(name);

// 日誌服務

log("hello method ends....");

}

private void log(String msg) {

logger.log(Level.INFO, msg);

}

}

我们可以看到在hello方法的实现中，前后插入了日志记录的方法。  
下面我们就测试一下

public class ProxyDemo {

public static void main(String[] args) {

IHello proxy =

new HelloProxy(new HelloSpeaker());

proxy.hello("Justin");

}

}

Paste\_Image.png

程序中执行hello方法的是代理对象，实例化代理对象的时候，必须传入被代理对象，而且声明代理对象的时候，必须使用代理对象和被代理对象共同实现的接口，以便实现多态。

代理对象将代理真正执行hello方法的被代理对象来执行hello,并在执行的前后加入日志记录的操作这样就可以使业务代码专注于业务实现。

这就是静态代理

动态代理

jdk1.3加入了动态代理相关的API，从上面静态代理的例子我们知道，静态代理，需要为被代理对象和方法实现撰写特定的代理对象，显然这样做并不灵活，我们希望可以有一个公用的代理，可以动态的实现对不同对象的代理，这就需要利用到反射机制和动态代理机制。  
在动态代理中，一个handler可以代理服务各种对象，首先，每一个handler都必须继承实现java.lang.reflect.InvocationHandler接口，下面具体实例说明，依然是上面那个记录日志的例子

LogHandler.java

package Reflection;

import java.util.logging.\*;

import java.lang.reflect.\*;

public class LogHandler implements InvocationHandler {

private Logger logger =

Logger.getLogger(this.getClass().getName());

private Object delegate;

public Object bind(Object delegate) {

this.delegate = delegate;

return Proxy.newProxyInstance(

delegate.getClass().getClassLoader(),

delegate.getClass().getInterfaces(),

this);

}

public Object invoke(Object proxy, Method method,

Object[] args) throws Throwable {

Object result = null;

try {

log("method starts..." + method);

result = method.invoke(delegate, args);

logger.log(Level.INFO, "method ends..." + method);

} catch (Exception e){

log(e.toString());

}

return result;

}

private void log(String message) {

logger.log(Level.INFO, message);

}

}

具体来说就是使用Proxy.newProxyInstance()静态方法new一个代理对象出来，底层会使用反射机制，建立代理对象的时候，需要传入被代理对象的class，以及被代理对象的所实现的接口，以及代理方法调用的调用程序 InvocationHandler，即实现 InvocationHandler接口的对象。这个对象会返回一个指定类指定接口，指定 InvocationHandler的代理类实例，这个实例执行方法时，每次都会调用 InvocationHandler的invoke方法，invoke方法会传入被代理对象的方法与方法参数，实际方法的执行会交给method.invoke().所以我们就可以在其前后加上日志记录的工作。

接下来我们就来测试一下，使用logHandler的bind方法来绑定代理对象：

package Reflection;

import java.lang.reflect.Proxy;

public class ProxyDemo {

public static void main(String[] args) {

LogHandler logHandler = new LogHandler();

IHello helloProxy =

(IHello) logHandler.bind(new HelloSpeaker());

helloProxy.hello("baba");

}

}

* 修改对象A的equals方法的签名，那么使用HashMap存放这个对象实例的时候，会调用哪个equals方法？

先求出key的hashcode(),比较其值是否相等，若相等再比较equals(),若相等则认为他们是相等 的。若equals()不相等则认为他们不相等。如果只重写hashcode()不重写equals()方法，当比较equals()时只是看他们是否为 同一对象（即进行内存地址的比较）,所以必定要两个方法一起重写。HashMap用来判断key是否相等的方法，其实是调用了HashSet判断加入元素 是否相等。