# 1.Java对象的3种特征

## 封装

封装就是把同一类事物的共性归到同一类中方便使用。

1. 什么是封装？

封装就是利用抽象的数据类型把数据和数据的操作封装起来，成为一个不可分割的部分。数据隐藏在抽象类的内部，只留一些接口与外界交互，隐藏数据细节和具体的实现细节。

(2) 封装的作用。

首先封装能够实现更专业的分工，这是肯定的。其次良好的封装能够减少代码的耦合性。可以隐藏细节，仅提供接口。

## 继承

（1）什么是继承？

继承是从已有的类中派生出新的类，然后新的类能够拥有已有类的数据属性和行为，也可以扩展新的数据属性和行为。

（2）继承的作用

继承避免了对一般类和特殊类之间共同属性的重复描述，提供了软件复用的功能。其次，继承通过增强一致性来减少模块间的接口和界面，大大加强了程序的易维护性。

## 多态

（1）什么是多态?

多态就是对于同一种行为，不同的对象能做出不同的动作。方法的重写，重载与动态连接构成多态性。通过类的向上转型来实现。

（2）多态的作用。

多态的作用是加强代码的可扩展性，减少代码的耦合。

# 2.Java的4大引用类型

## 强引用

强引用是指new一个对象产生的引用，只要强引用存在，则gc永远不会回收强引用。

## 软引用

软引用是指一些有用，但是非必须的引用，在系统将要发生内存溢出时，会回收这些对象。释放内存资源。软引用可以用来实现内存敏感的高速缓存。

## 弱引用

弱引用就是比软引用更弱一级的引用，它的生命周期只能到第二次gc操作之前。一旦发现就会被gc回收，回收之后会被添加到一个与之关联的队列中去。

## 虚引用

虚引用时一种最弱的引用关系，它时可有可无的，在被回收之后，会添加到一个与之关联的队列，虚引用的作用在于跟踪对象的回收，清理被回收对象的相关资源。

# 3.static和final

## final类

就是表示当前类不能被继承，final方法不能够被重写。

## static内部类和非static内部类

static类只能是内部类，static内部类能够只能访问外部类的static方法和static域，而非static内部类可以访问外部类的所有的方法和域。Static内部类的实例化不需要先实例化外部类，而非static内部类的实例化，必须先实例化一个外部类对象。

## static方法和非static方法

staic修饰的方法会先一步被加载到方法区，以便共享使用。Static方法中不能使用this和super，因为static方法在创建对象之前就创建了，而this，和super是针对对象的。非static方法是可以使用this和super的。Static方法只加载一次，而非static方法，有多少对象，它就拷贝多少次。因为static方法只有一个，所以调用static方法时要考虑线程安全的问题。同一个类中的static方法只能访问staic的域。

## static域和非static域

static变量在加载的时候先于main方法加载在内存中的方法区，而非static变量，是要创建变量才加载到堆内存中的。Static变量是单独划分一块空间，不与对象绑定。Static变量只加载一次，而非static变量会加载很多次。

# 4.String,StringBuffer与StringBuilder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）  
 简要的说， String 类型和 StringBuffer 类型的主要性能区别其实在于 String 是不可变的对象, 因此在每次对 String 类型进行改变的时候其实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，所以经常改变内容的字符串最好不要用 String ，因为每次生成对象都会对系统性能产生影响，特别当内存中无引用对象多了以后， JVM 的 GC 就会开始工作，那速度是一定会相当慢的。  
 而如果是使用 StringBuffer 类则结果就不一样了，每次结果都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，再改变对象引用。所以在一般情况下我们推荐使用 StringBuffer ，特别是字符串对象经常改变的情况下。而在某些特别情况下， String 对象的字符串拼接其实是被 JVM 解释成了 StringBuffer 对象的拼接，所以这些时候 String 对象的速度并不会比 StringBuffer 对象慢，而特别是以下的字符串对象生成中， String 效率是远要比 StringBuffer 快的：  
 String S1 = “This is only a” + “ simple” + “ test”;  
 StringBuffer Sb = new StringBuilder(“This is only a”).append(“ simple”).append(“ test”);  
 你会很惊讶的发现，生成 String S1 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，在 JVM 眼里，这个  
 String S1 = “This is only a” + “ simple” + “test”; 其实就是：  
 String S1 = “This is only a simple test”; 所以当然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 S2 = “This is only a”;  
String S3 = “ simple”;  
String S4 = “ test”;  
String S1 = S2 +S3 + S4;  
这时候 JVM 会规规矩矩的按照原来的方式去做

在大部分情况下 StringBuffer > String  
StringBuffer  
Java.lang.StringBuffer线程安全的可变字符序列。一个类似于 String 的字符串缓冲区，但不能修改。虽然在任意时间点上它都包含某种特定的字符序列，但通过某些方法调用可以改变该序列的长度和内容。  
可将字符串缓冲区安全地用于多个线程。可以在必要时对这些方法进行同步，因此任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致。  
StringBuffer 上的主要操作是 append 和 insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串缓冲区中。append 方法始终将这些字符添加到缓冲区的末端；而 insert 方法则在指定的点添加字符。  
例如，如果 z 引用一个当前内容是“start”的字符串缓冲区对象，则此方法调用 z.append("le") 会使字符串缓冲区包含“startle”，而 z.insert(4, "le") 将更改字符串缓冲区，使之包含“starlet”。  
在大部分情况下 StringBuilder > StringBuffer

java.lang.StringBuilde  
java.lang.StringBuilder一个可变的字符序列是5.0新增的。此类提供一个与 StringBuffer 兼容的 API，但不保证同步。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。两者的方法基本相同。

# 5.关于java中的Time和date

1. 使用SystemClock.elapsedRealtime()可以获取系统开机至今所经历的毫秒数。

使用System.currentTimeMillis()可以获取1970年1月1日至今所经历的毫秒数。

1. 设置Date类型数据的格式，使用SimpleDateFormat类：

//先定义一个Date类型对象

Date date=new Date(System.currentTimeMillis());

//实例化一个SimpleDateFormat对像，并传入你设置好的日期格式

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

//sdf.format（date）返回String类型的数据

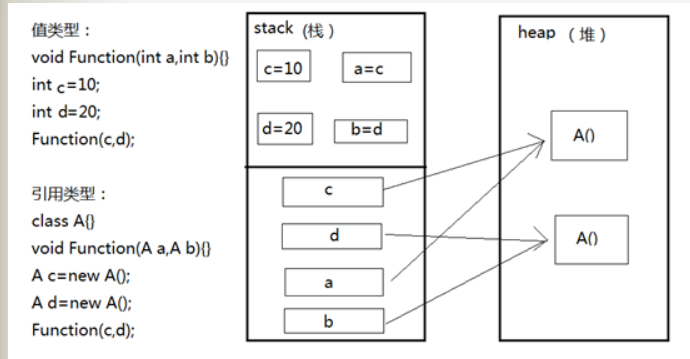
String time=sdf.format(date)

# 6. 基本数据类型和引用类型

基本数据类型和引用类型。

基本数据类型包括：byte(8),shot(16),int(32),long(64),float(32),double(64),char(16)

基本数据类型存放在栈中。

引用数据类型：引用数据类型是除了基本数据类型之外的其他数据类型。引用即别名存放在栈中，而对象存放在堆中，就是说每新建一个新的对象，会在堆中分配存储空间。

**7.抽象类和接口的区别以及使用场景（记）**

* [抽象类和接口的区别以及使用场景记](https://blog.csdn.net/lamyuqingcsdn/article/details/50501871#%E6%8A%BD%E8%B1%A1%E7%B1%BB%E5%92%8C%E6%8E%A5%E5%8F%A3%E7%9A%84%E5%8C%BA%E5%88%AB%E4%BB%A5%E5%8F%8A%E4%BD%BF%E7%94%A8%E5%9C%BA%E6%99%AF%E8%AE%B0)
  + [相同点](https://blog.csdn.net/lamyuqingcsdn/article/details/50501871#1-%E7%9B%B8%E5%90%8C%E7%82%B9)
  + [不同点](https://blog.csdn.net/lamyuqingcsdn/article/details/50501871#2-%E4%B8%8D%E5%90%8C%E7%82%B9)
  + [interface的应用场合](https://blog.csdn.net/lamyuqingcsdn/article/details/50501871#3-interface%E7%9A%84%E5%BA%94%E7%94%A8%E5%9C%BA%E5%90%88)
  + [abstract class的应用场合](https://blog.csdn.net/lamyuqingcsdn/article/details/50501871#4-abstract-class%E7%9A%84%E5%BA%94%E7%94%A8%E5%9C%BA%E5%90%88)

**1. 相同点**

1. 两者都是抽象类，都不能实例化。
2. interface实现类及abstrct class的子类都必须要实现已经声明的抽象方法。

**2. 不同点**

1. interface需要实现，要用implements，而abstract class需要继承，要用extends。
2. 一个类可以实现多个interface，但一个类只能继承一个abstract class。
3. interface强调特定功能的实现，而abstract class强调所属关系。
4. 尽管interface实现类及abstrct class的子类都必须要实现相应的抽象方法，但实现的形式不同。interface中的每一个方法都是抽象方法，都只是声明的(declaration, 没有方法体)，实现类必须要实现。而abstract class的子类可以有选择地实现:

这个选择有两点含义：

* + Abastract class中并非所有的方法都是抽象的，只有那些冠有abstract的方法才是抽象的，子类必须实现。那些没有abstract的方法，在Abstrct class中必须定义方法体。
  + abstract class的子类在继承它时，对非抽象方法既可以直接继承，也可以覆盖；而对抽象方法，可以选择实现，也可以通过再次声明其方法为抽象的方式，无需实现，留给其子类来实现，但此类必须也声明为抽象类。既是抽象类，当然也不能实例化。

1. abstract class是interface与Class的中介。
2. interface是完全抽象的，只能声明方法，而且只能声明pulic的方法，不能声明private及protected的方法，不能定义方法体，也不能声明实例变量。然而，interface却可以声明常量变量，并且在JDK中不难找出这种例子。但将常量变量放在interface中违背了其作为接口的作用而存在的宗旨，也混淆了interface与类的不同价值。如果的确需要，可以将其放在相应的abstract class或Class中。   
   abstract class在interface及Class中起到了承上启下的作用。一方面，abstract class是抽象的，可以声明抽象方法，以规范子类必须实现的功能；另一方面，它又可以定义缺省的方法体，供子类直接使用或覆盖。另外，它还可以定义自己的实例变量，以供子类通过继承来使用。
3. 以前抽象类或者抽象方法默认是protected，jdk1.8以后改成默认default了。

JDK 1.8以前，接口中的方法必须是public的

JDK 1.8时，接口中的方法可以是public的，也可以是default的

JDK 1.9时，接口中的方法可以是private的

**3. interface的应用场合**

1. 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。
2. 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。
3. 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。
4. 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。

**4. 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：

1. 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。
2. 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。
3. 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

# 8.逆波兰表达式

1. 逆波兰计算器

逆波兰表达式是一种十分有用的表达式，它将复杂表达式转换为可以依靠简单的操作得到计算结果的表达式。例如(a+b)\*(c+d)转换为ab+cd+\*。

1. 逆波兰表达式

如果当前字符为变量或者数字就入栈，如果是运算符，则将栈顶两个元素弹出，做相应的运算，表达式扫描完后，栈里的就是结果。

1. 为什么要使用逆波兰表达式

因为计算机在计算普通表达式时，要对运算符的优先级进行判断，过于复杂的表达式会令计算机崩溃，而逆波兰表达式只需要进行简单的入栈出栈操作就可以完成了。

# 9.方法重写的两同两小一大原则

所谓子类方法重写父类方法遵循“两同两小一大”的规则  
  
两同：  
1)  
方法名  
2)  
形参列表  
两小：  
1）  
返回值类型比父类更小或相等  
2）  
异常比父类方法更小或相等  
一大：  
子类权限比父类大或相等

# 10.自动拆、装箱

## 自动拆箱

### “==”运算符

基本类型和基本类型的封装类型进行“==”运算符的比较时，基本类型的封装类型将会自动拆箱为基本类型来进行比较。

## 自动装箱

### equals()比较

基本类型封装类型调用equals（）方法和基本类型进行比较时，会先把基本类型进行装箱，再进行比较。比较方式使先看类型是否相同，若相同再比较值。

# 11.Java的保留关键字

**goto和const**是保留字也是关键字。

1，Java **关键字**列表 (依字母排序 共50组)：

abstract, assert, boolean, break, byte, case, catch, char, class, **const**（保留关键字）, continue, default, do, double, else, enum, extends, final, finally, float, for, **goto**（保留关键字）, if, implements, import, instanceof, int, interface, long, native, new, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while

2，**保留字**列表 (依字母排序 共14组)，Java保留字是指现有Java版本尚未使用，但以后版本可能会作为关键字使用：

byValue, cast, **false**, future, generic, inner, operator, outer, rest, **true**, var, **goto** （保留关键字） , **const** （保留关键字） , **null**

# 12.Java8新特性

## 接口中的default方法的作用：

在java8之前我们如果要修改接口就会导致接口的所有实现类都会被修改。由于实现类不必实现default方法，所以我们通过修改或添加default方法的方式来修改接口，从而达到修改功能的目的。

# 13.Integer和Int

JVM中一个字节以下的整型数据会在JVM启动的时候加载进内存，除非用new Integer()显式的创建对象，否则都是同一个对象

所有只有i04是一个新对象，其他都是同一个对象

Integer i01=59 的时候，会调用 Integer 的 valueOf 方法，

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static Integer valueOf(int i) {       assert IntegerCache.high>= 127;       if (i >= IntegerCache.low&& i <= IntegerCache.high)       return IntegerCache.cache[i+ (-IntegerCache.low)];       return new Integer(i); } |

这个方法就是返回一个 Integer 对象，只是在返回之前，看作了一个判断，判断当前 i 的值是否在 [-128,127] 区别，且 IntegerCache 中是否存在此对象，如果存在，则直接返回引用，否则，创建一个新的对象。

在这里的话，因为程序初次运行，没有 59 ，所以，直接创建了一个新的对象。

int i02=59 ，这是一个基本类型，存储在栈中。

Integer i03 =Integer.valueOf(59); 因为 IntegerCache 中已经存在此对象，所以，直接返回引用。

Integer i04 = new Integer(59) ；直接创建一个新的对象。

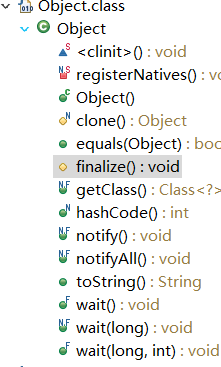
System. out .println(i01== i02); i01 是 Integer 对象， i02 是 int ，这里比较的不是地址，而是值。 Integer 会自动拆箱成 int ，然后进行值的比较。所以，为真。

System. out .println(i01== i03); 因为 i03 返回的是 i01 的引用，所以，为真。

System. out .println(i03==i04); 因为 i04 是重新创建的对象，所以 i03,i04 是指向不同的对象，因此比较结果为假。

System. out .println(i02== i04); 因为 i02 是基本类型，所以此时 i04 会自动拆箱，进行值比较，所以，结果为真。

# 14.Object的方法



## Clone（）方法

**protected** **native** Object clone() **throws** CloneNotSupportedException;

一个native方法，使用native方法比使用非native方法要快。所以clone（）,构造一个对象比使用new要快，而且clone()方法复制的对象已经拥有了原有对象的值，这在某些情况下节省了很多重新赋值的时间。

我们自己定义的对象要使用clone（）方法需要满足两个条件，一个是继承Object类，一个是实现Cloneable接口。第一个条件默认满足，所以我们只需要实现这个接口就可以了，如果不实现这个接口，就使用clone()方法的话，会抛出异常。

### 深复制与浅复制

所谓深复制和浅复制时针对拥有对象成员变量的对象来说的。对于这样的对象来说，我们直接使用Object的clone()方法，并不能完全复制它的对象成员变量，复制的只是对象成员变量的引用，这就是浅复制。如果想要复制它的对象成员变量，也就是深复制，我们有两种方法实现，一种是让它的对象成员变量也继承Cloneable接口，然后重写clone（）方法，再clone()方法里面调用object类的clone方法以后，对于得到的新对象的对象成员变量再调用一次clone()方法。第二种是序列化与反序列化。

## Equals()方法

**public** **boolean** equals(Object obj) {

**return** (**this** == obj);

}

Equals方法是通过“==”符号来比较的，比较的是地址值。

## HashCode和getClass()方法

**public** **native** **int** hashCode();

这是一个本地方法，返回的是对象的地址值。

**public** **final** **native** Class<?> getClass();

返回的一个class对象。

## toString()方法

**public** String toString() {

**return** getClass().getName() + "@" + Integer.toHexString(hashCode());

}

返回类名和地址值。

## Wait(),Notify()和NotifyAll()

Wait（）方法是让线程放弃锁，进入等待。等待notify（）的唤醒。NotifyAll()是唤醒所有的wait的线程。

## Finalize()

对象被回收时，如果对象重写了这个方法，在被回收时会被加入一个队列，然后有一个专门的线程来执行这个队列中对象的finalize（），不过只会执行一次。也就是说你可以在这个finalize（）方法中自救一次，让自己重写连上root链，但是仅此一次，第二次回收时就不会再执行这个方法。