目录

[Object通用方法？ 2](#_Toc117607856)

[equals 作用 2](#_Toc117607857)

[equals 与 == 区别 3](#_Toc117607858)

[equals 的默认实现 & 重写 3](#_Toc117607859)

[Object.hashCode() 5](#_Toc117607860)

[散列集合包括啥以及添加新元素的逻辑 5](#_Toc117607861)

[equals 与 hashCode 重写的关系以及hashCode重写的逻辑 6](#_Toc117607862)

[Object.clone()的使用和底层原理 10](#_Toc117607863)

[深拷贝的3种实现 11](#_Toc117607864)

[如何深拷贝一个List集合 14](#_Toc117607865)

[序列化 14](#_Toc117607866)

[序列化的方式包括两种： 15](#_Toc117607867)

[Parcelable方式的实现原理？ 15](#_Toc117607868)

[序列化常用场景： 15](#_Toc117607869)

[Serializable接口 16](#_Toc117607870)

[transient 关键字 16](#_Toc117607871)

[Serializable和Parcelable比较 17](#_Toc117607872)

[内部类的好处 18](#_Toc117607873)

[内部类的种类 18](#_Toc117607874)

[成员内部类： 18](#_Toc117607875)

[局部内部类： 19](#_Toc117607876)

[为什么局部内部类访问局部变量需要final? 20](#_Toc117607877)

[匿名内部类： 20](#_Toc117607878)

[静态内部类： 21](#_Toc117607879)

[Java 异常 21](#_Toc117607880)

[Java 异常关键字 22](#_Toc117607881)

[面试：final/finally/finalize的作用？ 23](#_Toc117607882)

[throws 25](#_Toc117607883)

[Java 异常体系 27](#_Toc117607884)

[自定义Java异常 30](#_Toc117607885)

# Object通用方法？

Object.equals()

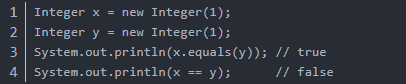
Object.hashCode()

Object.clone()

# equals 作用

Object.equals(Object obj)

Object的equals方法主要用于判断引用的对象的内容是否相等。



# equals 与 == 区别

==比较的是栈中的值。

equals比较的是堆中对象的内容。

基本数据类型（byte,short,char,int,long,float,double,boolean）用==比较的是两个数据的值是否相等。

引用类型（类、接口、数组）用==比较的是它们在内存中的存放地址是否相等（两个变量是否引用同一个对象）。

对象是存放在堆中的，栈中存放的是对象的引用（地址）。因此==是对栈中的值进行比较的。如果要比较堆中对象的内容是否相同，那么就要重写equals方法了。

# equals 的默认实现 & 重写

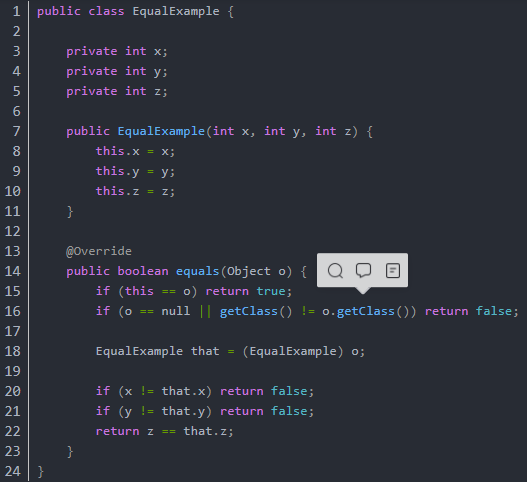
默认实现：

检查是否为同一个对象的引用，如果是直接返回 true；

检查是否是同一个类型，如果不是，直接返回 false；

将 Object 对象进行转型；

判断每个关键域是否相等。



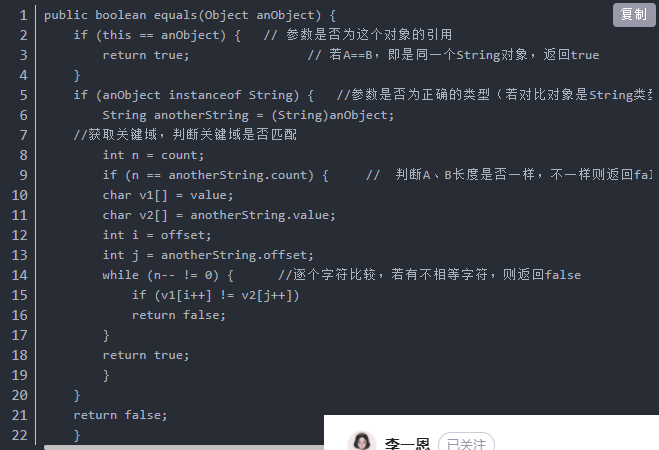
覆盖equals方法一般通过比较对象的内容是否相等来判断对象是否相等。如下为String类对equals方法进行重写。（不同类有不同等价方法的实现）

检查是否为同一个对象的引用，如果是直接返回 true；

检查是否是String类型，如果是则继续；

判断长度是否一样，不一样返回false

逐个字符进行比较，不同则false



5//则继续

# Object.hashCode()

（1）定义 & 散列集合 唯一性原理

hashCode方法返回一个hash码（int），主要作用是在对对象进行散列时作为key输入，因此需要每个对象的hashCode尽可能不同，这样才能保证散列的存取性能。事实上，Object类提供的默认实现确保每个对象的hash码不同（在对象的内存地址基础上经过特定算法返回一个hashCode）

# 散列集合包括啥以及添加新元素的逻辑

HashSet、HashMap以及HashTable

1调用hashCode定位到物理位置上判断是否可以存储在这个位置

2若不可以，调用equals判断是否相同

3相同则不存，不同则散列其他地址存储

hashCode用于配合基于散列的集合一起正常运行，这样的散列集合包括HashSet、HashMap以及HashTable。

散列集合中元素不可重复，Java则依据元素的hashCode来判断两个元素是否重复。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（放入对象的hashcode与集合中任一元素的hashcode不相等）；如果这个位置上已经有元素了（hashcode相等），就调用它的equals方法与新元素进行比较，相同的话就不存，不相同就散列其它的地址。（通过调用equals解决冲突）

# equals 与 hashCode 重写的关系以及hashCode重写的逻辑

重写equals（）必须重写hashCode（），使得equals（）相等的两个对象hashCode（）必须相等。

HashCode重写：重写散列函数，使内容一样的两个对象计算出的hashCode也相同。

等价的两个对象散列值一定相同

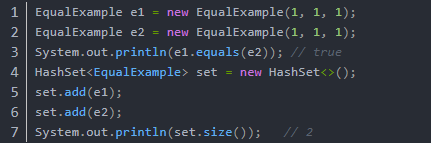
若重写equals(Object obj)方法，有必要重写hashcode()方法，确保通过equals(Object obj)方法判断结果为true的两个对象具备相等的hashcode()返回值。

散列值相同的两个对象不一定等价

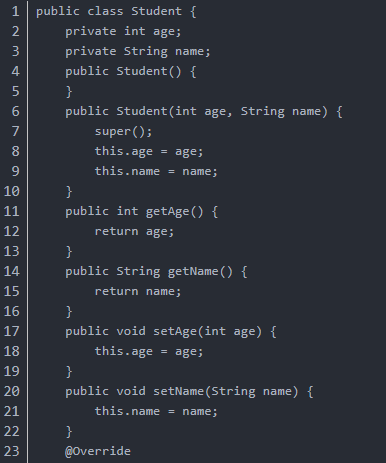
如果equals(Object obj)返回false，即两个对象“不相同”，并不要求对这两个对象调用hashcode()方法得到两个不相同的数。

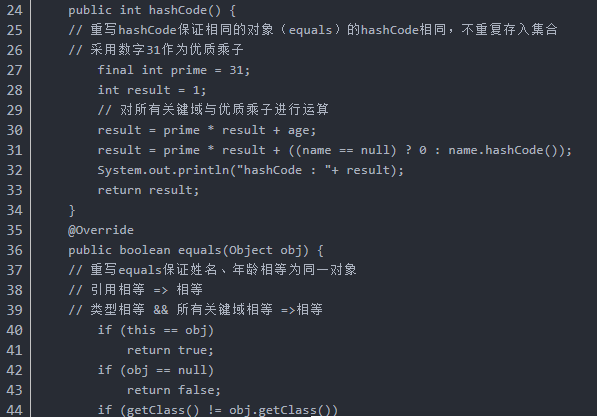
为了满足上述规范，覆盖equals方法时总要覆盖hashCode，这样该类才能结合所有基于散列的集合（如HashMap、HashSet、HashTable）一起正常运作。

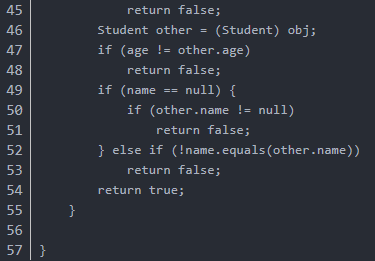
下面的代码中，新建了两个等价的对象，并将它们添加到 HashSet 中。我们希望将这两个对象当成一样的，只在集合中添加一个对象，但是因为 EqualExample 没有实现 hashCode() 方法，因此这两个对象的散列值是不同的，最终导致集合添加了两个等价的对象。



重写实例：Student类同时覆盖hashCode与equals方法





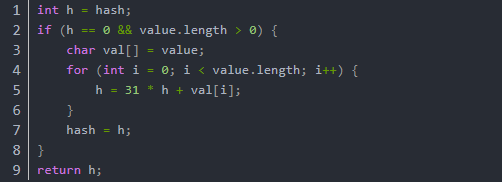


散列函数构造

理想的散列函数应当具有均匀性，即不相等的对象应当均匀分布到所有可能的散列值上。这就要求了散列函数要把所有域的值都考虑进来。可以将每个域都当成 R 进制的某一位，然后组成一个 R 进制的整数。R 一般取 31，因为它是一个奇素数，如果是偶数的话，当出现乘法溢出，信息就会丢失，因为与 2 相乘相当于向左移一位。

一个数与 31 相乘可以转换成移位和减法：31\*x == (x<<5)-x，编译器会自动进行这个优化。

String hashCode 方法实现：



# Object.clone()的使用和底层原理

使用步骤

1实现Cloneable接口

2 重写clone（）方法

原理：

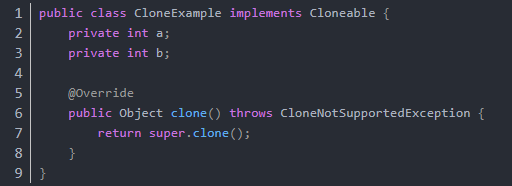
浅拷贝

重写clone()方法

clone() 是 Object 的 protected 方法，它不是 public，一个类不显式去重写 clone()，其它类就不能直接去调用该类实例的 clone() 方法。

实现Cloneable接口

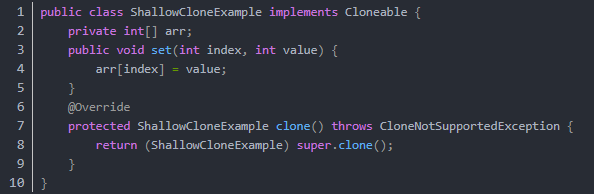
Cloneable 接口只是规定，如果一个类没有实现 Cloneable 接口又调用了 clone() 方法，就会抛出 CloneNotSupportedException。

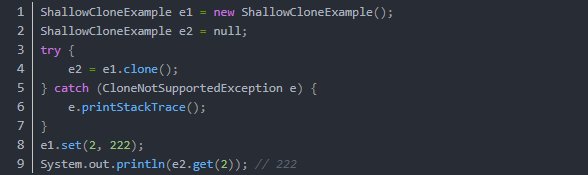


浅拷贝（super.clone()）

拷贝对象和原始对象的引用类型引用同一个对象。

创建一个新对象，然后将当前对象的非静态字段复制到该新对象，如果字段是值类型（基本类型）的，那么对该字段值执行复制；如果该字段是引用类型（对象、String）的话，则复制引用但不复制引用的对象。因此，原始对象及其副本对象引用同一个对象，如果其中一个对象改变了这个地址，就会影响到另一个对象。





可以看出e1与e2的arr指向了同一个对象

# 深拷贝的3种实现

（1）实现Cloneable接口重写clone（）进行深拷贝

（2）自定义拷贝函数

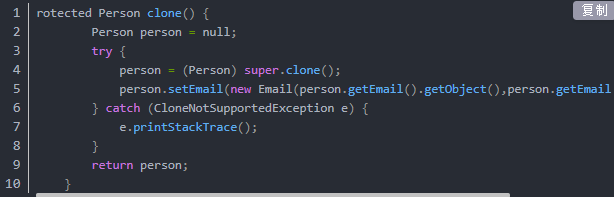
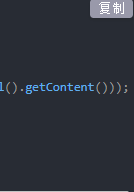
（3）利用序列化完成对象深拷贝

拷贝对象和原始对象的引用类型引用不同对象。

深拷贝会拷贝所有的属性,并拷贝属性指向的动态分配的内存。当对象和它所引用的对象一起拷贝时即发生深拷贝。深拷贝相比于浅拷贝速度较慢并且花销较大。

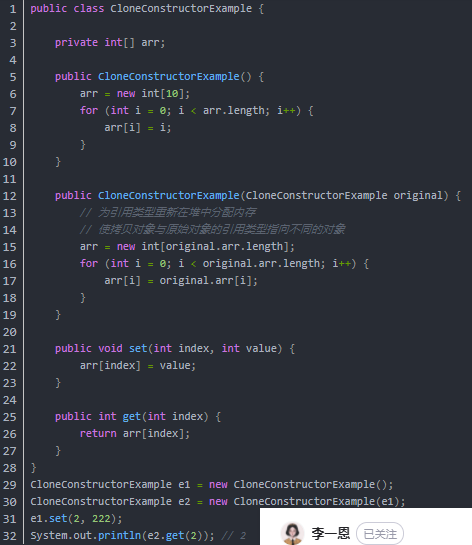
（1）实现Cloneable接口进行深拷贝

实现clone方法，并且在clone方法内部，把该对象引用的其他对象也要clone一份 ， 这就要求这个被引用的对象必须也要实现Cloneable接口并且实现clone方法。

（2）自定义拷贝函数

使用 clone() 方法来拷贝一个对象即复杂又有风险，它会抛出异常，并且还需要类型转换。Effective Java 书上讲到，最好不要去使用 clone()，可以使用拷贝构造函数或者拷贝工厂来拷贝一个对象。



（3）利用序列化完成对象深拷贝

将原对象（需要实现Serializable接口）写入到一个字节流中（outputStream.writeObject），再从字节流中将其读取出来创建一个新的对象（inputStream.readObject）



# 如何深拷贝一个List集合

重写clone法：重写clone方法并遍历即可

序列化法：将List集合写入字节流，再读取出来解析成一个List集合

Json拷贝法：将List集合映射为一个Json字符串，再解析成一个List集合

# 序列化

序列化用于将对象状态转换为字节流，之后进行读写操作，或在网络中传输。

从字节流转换为对象的过程称为反序列化。

# 序列化的方式包括两种：

1实现Serializable接口（Java自带）

2实现Parcelable接口（Android专用、效率更高）

# Parcelable方式的实现原理？

将一个完整的对象进行分解，而分解后的每一部分都是Intent所支持的数据类型，从而实现传递对象的功能。

# 序列化常用场景：

1将对象状态存储到文件或数据库中

2在网络上传输对象

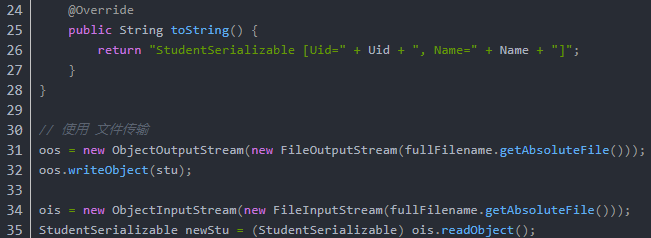
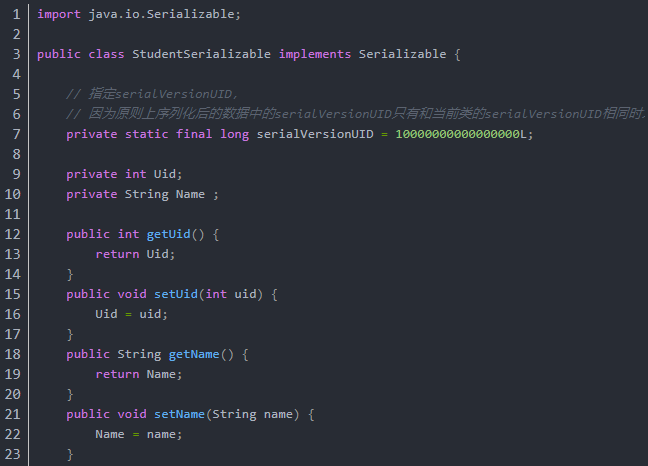
3在Activity/Fragment之间通过Intent传递对象（Android）

注意：

静态成员变量属于类不属于对象，所以不会参与序列化(对象序列化保存的是对象的“状态”，也就是它的成员变量，因此序列化不会关注静态变量)

用transient关键字标记的成员变量不参与序列化(在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null)

# Serializable接口



6//才能被正常的反序列化

# transient 关键字

只要一个类实现了Serilizable接口，这个类的所有属性和方法都会自动序列化。

为了安全起见，不希望在网络操作（主要涉及到序列化操作，本地序列化缓存也适用）中被传输，这些信息对应的变量就可以加上transient关键字。换句话说，这个字段的生命周期仅存于调用者的内存中而不会写到磁盘里持久化。

一旦变量被transient修饰，变量将不再是对象持久化的一部分，该变量内容在序列化后无法获得访问。

transient关键字只能修饰变量，而不能修饰方法、类。

一个静态变量不管是否被transient修饰，均不能被序列化。

Serializable和Parcelable比较



# 内部类的好处

1更好地封装

2实现多继承

在Java中，定义在类内部的类被称为内部类。设计内部类的好处是：

1更好地封装：内部类中的属性和方法与其他类隔离，且即使是外部类也不能直接访问，相反内部类可以直接访问外部类的属性和方法。

2实现多继承：每个内部类都能独立地实现接口。无论外部类是否已实现了某个接口，对内部类都没有影响。

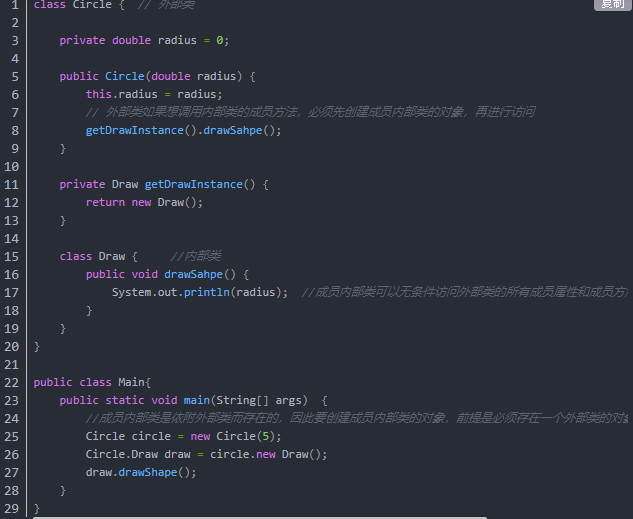
# 内部类的种类

内部类一般来说包括这四种：成员内部类、局部内部类、匿名内部类和静态内部类。

# 成员内部类：

无条件访问外部类，持有外部类的引用，依附于外部类存在

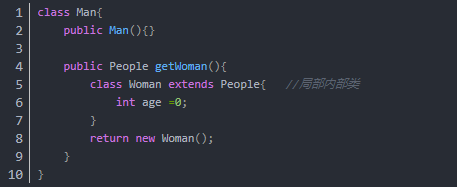
成员内部类定义为位于另一个类的内部，类似于一个成员。成员内部类可以无条件访问外部类的所有成员属性和成员方法（内部类编译后会默认持有外部类对象的一个引用，因此可以在成员内部类中随意访问外部类的成员。）。成员内部类是依附外部类而存在的，因此要创建成员内部类的对象，前提是必须存在一个外部类的对象（否则所持有的外部类对象的引用无法初始化赋值，也无法创建内部类对象）。



17*//成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员）*

# 局部内部类：

局部内部类是定义在一个方法或者一个作用域里面的类，它和成员内部类的区别在于局部内部类的访问仅限仅于方法内或者该作用域内；



# 为什么局部内部类访问局部变量需要final?

1解决生命周期不同的问题

2防止数据不同步问题

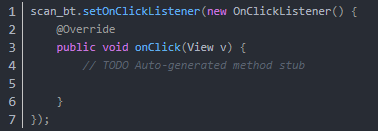
局部内部类引用局部变量，不添加final，会出现生命周期不同，导致非法引用问题。因为非final的局部变量的生命周期比局部内部类的生命周期短，当方法执行结束，非final的局部变量会被销毁，而局部内部类对局部变量的引用依然存在，当局部内部类调用局部变量时，会出错，出现非法引用。

而且直接拷贝会出现数据不同步问题，所以使用final（若定义为final，即拷贝了一个变量的副本，提供给局部内部类，这个副本的生命周期和局部内部类一样长，并且这个副本不可修改，保证了数据的同步）。

故使用final可以保证合法引用，而且数据不可修改。

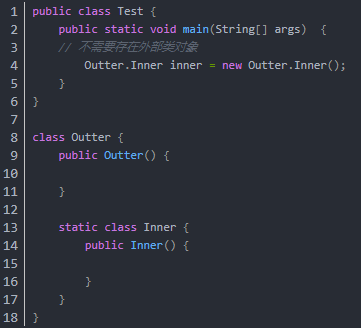
# 匿名内部类：

匿名内部类是直接使用 new 来生成一个对象的引用，创建匿名内部类时它会立即创建一个该类的实例，且仅能被使用一次，所以匿名内部类是不能够被重复使用;



# 静态内部类：

用static修饰的内部类。静态内部类调用时是不需要依赖于外部类的，不需要创建外部类，便能创建内部类。并且它不能使用外部类的非static成员变量或者方法，因为外部类的非static成员必须依附于具体的对象。（不持有外部类对象的引用）



# Java 异常

Java中异常体系是Java提供的一种定位错误以及响应错误的一种机制。

Java异常机制可以使程序中异常处理代码和正常业务代码分离，保证程序代码更加优雅，并提高程序健壮性。

# Java 异常关键字

try

用于监听。将要被监听的代码(可能抛出异常的代码)放在try语句块之内，当try语句块内发生异常时，异常就被抛出。

catch

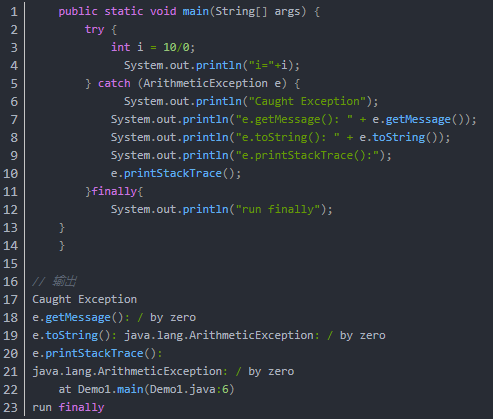
用于捕获异常。catch用来捕获try语句块中发生的异常。

finally

finally语句块总是会被执行。它主要用于回收在try块里打开的物理资源(如数据库连接、网络连接和磁盘文件)。只有finally块，执行完成之后，才会回来执行try或者catch块中的return或者throw语句，如果finally中使用了return或者throw等终止方法的语句，则就不会跳回执行，直接停止。

（1）被finally控制的语句体一定会执行（特殊情况：在执行到finally之前jvm退出，如：System.exit(0)），用于释放资源，在IO流操作和数据库操作中会见到。

（2）finally块中的内容会先于try中的return语句执行（return执行前会将finally语句执行完再执行return 进行返回）



在try语句块中有除数为0的操作，该操作会抛出java.lang.ArithmeticException异常。通过catch，对该异常进行捕获。

观察结果我们发现，并没有执行System.out.println(“i=”+i)。这说明try语句块发生异常之后，try语句块中的剩余内容就不会再被执行了。

finally语句块中的语句总是会执行。

# 面试：final/finally/finalize的作用？

final修饰符（关键字）

在Java中，final关键字可以用来修饰类、方法和变量。

修饰类

当用final修饰一个类时，表明这个类不能被继承。

修饰方法

当用final修饰一个方法时，该方法是不能被子类所覆盖的。

修饰变量

当final修饰一个基本数据类型时，表示该基本数据类型的值一旦在初始化后便不能发生变化；如果final修饰一个引用类型时，则在对其初始化之后便不能再让其指向其他对象了，但该引用所指向的对象的内容是可以发生变化的。本质上是一回事，因为引用的值是一个地址，final要求值，即地址的值不发生变化。

final变量必须要初始化，其初始化可在变量定义时直接赋值，也可在构造函数中赋值或者作为参数传递。且在之后的引用中只读取使用，无法修改。

finally（异常处理）

finally关键字一般用于异常处理中。finally结构使代码总会执行，不关有无异常发生。

finally在try,catch中可以有，可以没有。如果trycatch中有finally则必须执行finally块中的操作。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。

finalize（垃圾回收）

finalize方法是Object提供的实例方法，通过调用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。

finalize方法执行流程：

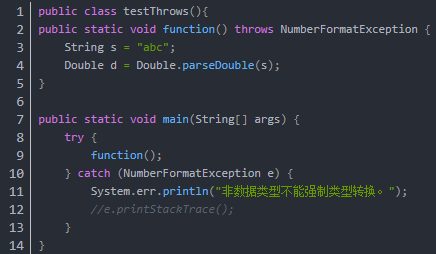
当对象变成(GC Roots)不可达时，GC会判断该对象是否覆盖了finalize方法，若未覆盖或finalize()方法已被调用，则直接将其回收（没有必要执行）。否则，将该对象放入F-Queue队列，由一低优先级Finalizer线程执行该队列中对象的finalize方法。（但并不承诺等待运行结束）执行finalize方法完毕后，GC会再次判断该对象是否可达，若不可达，则进行回收，否则，对象“复活”。

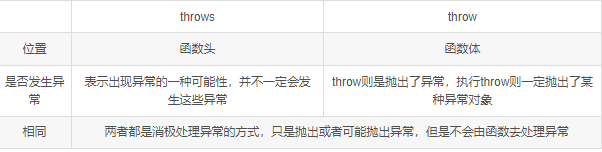
对象"复活"： finalize()方法是对象逃脱死亡命运的最后一次机会，稍后GC将对F-Queue中的对象进行第二次小规模的标记，如果对象要在finalize()中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this关键字）赋值给某个类变量或者对象的成员变量，那么在第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那么基本上它就真的被回收了。

因此，finalize()并不是必须要执行的，它只能执行一次或者0次。如果在finalize中建立对象关联，则当前对象可以复活一次。

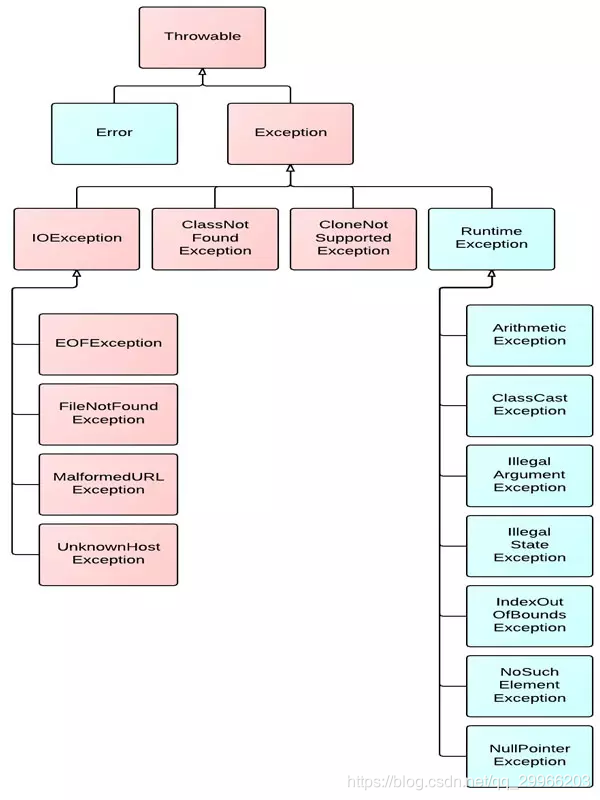
# throws

当某个方法可能会抛出某种异常时用于throws 声明可能抛出的异常，然后交给上层调用它的方法程序处理。





# Java 异常体系



Java异常以Throwable开始，扩展出Error和Exception。

Throwable

Throwable是 Java 语言中所有错误或异常的超类。Throwable包含两个子类: Error 和 Exception。它们通常用于指示发生了异常情况。

Throwable包含了其线程创建时线程执行堆栈的快照，它提供了printStackTrace()等接口用于获取堆栈跟踪数据等信息。

Error

Error是程序代码无法处理的错误，当资源不足、约束失败、或是其它程序无法继续运行的条件发生时，就产生错误。比如OutOfMemoryError、ThreadDeath等。和运行时异常一样，编译器也不会对错误进行检查。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止退出，其表示程序在运行期间出现了十分严重、不可恢复的错误，应用程序只能中止运行。

Exception

Exception及其子类是 Throwable 的一种形式，它指出了合理的应用程序想要捕获的条件。分为运行时异常和非运行时异常：

运行时异常（不检查异常）

运行时异常都是RuntimeException类及其子类异常，是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。编译器不会检查RuntimeException。程序代码中自行选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序代码应该从逻辑角度尽可能避免这类异常的发生。

举例常见运行时异常

java.lang.NullPointerException

空指针异常，调用了未经初始化的对象或者是不存在的对象（null对象）。

java.lang.ClassNotFoundException

指定的类不存在异常，一般类的名称和路径不正确。

java.lang.ArrayIndexOutOfBoundsException

数组下标越界异常，调用的下标超出了数组的范围。

java.lang.NoSuchMethodError

方法不存在错误，当应用试图调用某类未定义的某个方法。

java.lang.IndexOutOfBoundsException

索引越界异常。当访问某个序列的索引值小于0或大于等于序列大小时，抛出该异常。

java.lang.NumberFormatException（extends IllegalArgumentException ）

数字格式异常。当试图将一个String转换为指定的数字类型，而该字符串确不满足数字类型要求的格式时，抛出该异常。

java.sql.SQLException

Sql语句执行异常

java.io.IOException

输入输出异常

java.lang.IllegalArgumentException

方法参数错误

java.lang.IllegalAccessException

无访问权限异常

非运行时异常（检查异常）

所有继承Exception且不是RuntimeException的异常都是非运行时异常，也称检查异常，如上图中的IOException和ClassNotFoundException，编译器会对其作检查，故程序中一定会对该异常进行处理，处理方法要么在方法体中声明抛出checked Exception，要么使用catch语句捕获checked Exception进行处理，不然不能通过编译。

# 自定义Java异常

1为这个异常找一个近似的类作为父类；

2在该类中编写两个构造器：（1）默认构造器（2）带message参数的构造器

