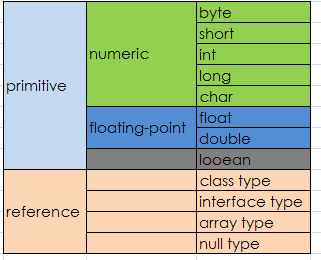
数据类型



容量单位

计算机中表储存容量大小的单位 bit、B（byte）、KB、MB、GB、TB（类似质量单位g、kg）

在计算机术语中，bit的中文含义是位。在计算机中的二进制数系统中，位，简记为b,也称为比特，每个0或1就是一个位(bit)。计算机中的CPU位数指的是CPU一次能处理的最大位数。

内存单位的换算：

（1）1KB=1024B；

（2）1B（byte，字节）= 8 bit；

（3）1KB（Kilobyte，千字节）=1024B= 10^3 B；

（4）1MB（Megabyte，兆字节，百万字节，简称“兆”）=1024KB= 10^6 B；

（5）1GB（Gigabyte，吉字节，十亿字节，又称“千兆”）=1024MB= 10^9 B；

（6）1TB（Terabyte，万亿字节，太字节）=1024GB= 10^12 B；

基本数据类型

变量就是申请内存来存储值。也就是说，当创建变量的时候，需要在内存中申请空间。内存管理系统根据变量的类型为变量分配存储空间，分配的空间只能用来储存该类型数据。为了合理利用内存空间，所以用到数据类型。如int a=2;不管a的是多少，都会向内存申请32bit的内存，如果a的是定值那可以用byte类型占8位内存节省空间。

Java语言提供了八种基本类型。六种数字类型（四个整数型，两个浮点型），一种字符类型，还有一种布尔型。Java代码中数字默认都是int类型，十进制,int类型/操作时得到的是只舍不进的整数。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 类型 | byte | short | int | long | float | double | boolean | char |
| 位数 | 8bit | 16bit | 32bit | 64bit | 32bit | 64bit | 1bit | 16bit |
| 默认值 | 0 | 0 | 0 | 0L | 0.0f | 0.0d | false | null |

整数型以byte为例，数据类型是8位、有符号的，以二进制补码表示的整数

原码，就是用二进制表示的原始编码，  
反码，就是除符号位外，其它位取反  
补码，正数的补码等于原码，负数的补码等于反码加1

byte a=5,二进制表示00000101(原码)，那么-5就是原码的反码再加1，反码加1叫补码

原码：00000101

反码：11111010

补码：11111011

byte a=5的二进制表示00000101，byte b=-5的二进制表示11111011

int a=5的二进制表示00000000000000000000000000000101，int a=-5的二进制表示1111111111111111111111111111111011

byte的最大值为01111111即2^7-1=127最小值为最大值的负数再减1即-2^7-1=-128.计算机中采用补码表示负数11111111：-1,10000000：-128，正数第一位是0，负数为1

byte a=127;  
byte b=4;  
byte c= (byte) (a+b);  
System.out.println(c);

结果：-125。代码中2个正整数相加变为负数就是因为二进制储存位数原因

浮点型与二进制转换

例如十进制22.8125等于二进制10110.1101的算法

整数除以2，商继续除以2，得到0为止，将余数逆序排列.

22 / 2 11 余 0

11 / 2 5 余 1

5 / 2 2 余 1

2 / 2 1 余 0

1 / 2 0 余 1

0结束

所以22的二进制是10110

小数乘以2，取整，小数部分继续乘以2，取整，得到小数部分0为止，将整数顺序排列

0.8125x2=1.625 取整1，小数部分是0.625

0.625x2=1.25 取整1，小数部分是0.25

0.25x2=0.5 取整0，小数部分是0.5

0.5x2=1.0 取整1，小数部分是0，结束

所以0.8125的二进制是0.1101

浮点型表示一个小数的时候存在精度不准确的问题

由上可知整数部分除以2结果一定为0余0或1，即整数部分永远可以用二进制精确表示，但小数部分就不一定了

如0.9用二进制表示

0.9\*2=1.8 取整数部分 1

0.8(1.8的小数部分)\*2=1.6 取整数部分 1

0.6\*2=1.2 取整数部分 1

0.2\*2=0.4 取整数部分 0

0.4\*2=0.8 取整数部分 0

0.8\*2=1.6 取整数部分 1

0.6\*2=1.2 取整数部分 0

......... 0.9二进制表示为(从上往下): 1100100100100......

上面的计算过程循环了，也就是说\*2永远不可能消灭小数部分，这样算法将无限下去。很显然，小数的二进制表示有时是不可能精确的,实际应用中对小数精度要求比较的如金额要用BigDecimal类,

Long类型java中十进制最大数2^32-1，十进制为19位长度，而在js中为2^53十进制为16位长度,所以后台长度过长，那么前台会精度丢失，解决方法转为String类型，手动转活着第三方包注解转如jackson.databind。

boolean数据类型表示一位的信息，只有两个取值：true 和 false，默认值是 false，二进制储存ture为1，false为0

char类型是一个单一的 16 位 Unicode 字符，数据类型可以储存任何字符，如char a=’a’占8位，二进制一般为0000000010110001，char a=‘汉’占16位，二进制一般为0100101010101110，即一个汉字=2个英文字母=2字节，

一个字符储存什么样的二进制则需要一套编码机制，每一个字符都对应一个唯一的二进制数，乱码的原因就是字符编码和解码用的不是同一套编码，假如‘汉’用gbk编码储存为010，UTF-8编码为001，编码时用gbk，解码时用utf导致乱码。有的编码不是包含所有汉字，所以有的特殊生僻字没有对应的二进制导致乱码。

reference类型在32位系统上每个占用4bytes, 在64位系统上每个占用8bytes。

基本类型的包装类

虽然 Java 语言是典型的面向对象编程语言，但其中的八种基本数据类型并不支持面向对象编程，基本类型的数据不具备“对象”的特性——不携带属性、没有方法可调用。 沿用它们只是为了迎合人类根深蒂固的习惯，并的确能简单、有效地进行常规数据处理。  
这种借助于非面向对象技术的做法有时也会带来不便，比如引用类型数据均继承了 Object 类的特性，要转换为 String 类型（经常有这种需要）时只要简单调用 Object 类中定义的toString()即可，而基本数据类型转换为 String 类型则要麻烦得多。为解决此类问题 ，Java为每种基本数据类型分别设计了对应的类，称之为包装类(Wrapper Classes)，也有教材称为外覆类或数据类型类。

包装类是引用类型所以默认值为null，如int 默认为0，Integer 默认null，Integer可以区分出未赋值和值为0的区别，int则无法表达出未赋值的情况

比如在查询条件时，

<if test="taskStatus!=null">

AND tsm.task\_status = #{taskStatus,jdbcType=INTEGER}

</if>

int类型不可以，Integer则可以，包装类型里包含了基本类型的各种属性，如Integer.SIZE、Integer.MAX\_VALUE、Integer.MIN\_VALUE

|  |  |
| --- | --- |
| 基本数据类型及对应的包装类 | |
| 基本数据类型 | 对应的包装类 |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| char | Character |
| float | Float |
| double | Double |
| boolean | Boolean |

每个包装类的对象可以封装一个相应的基本类型的数据，并提供了其它一些有用的方法。包装类对象一经创建，其内容（所封装的基本类型数据值）不可改变。  
基本类型和对应的包装类可以相互装换：

由基本类型向对应的包装类转换称为装箱，例如把 int 包装成 Integer 类的对象；

包装类向对应的基本类型转换称为拆箱，例如把 Integer 类的对象重新简化为 int。

int m = 500;

Integer obj = new Integer(m); // 手动装箱

int n = obj.intValue(); // 手动拆箱

Java 1.5 之后可以自动拆箱装箱，也就是在进行基本数据类型和对应的包装类转换时，系统将自动进行，这将大大方便程序员的代码书写

int m = 500;

Integer obj = m; // 自动装箱

int n = obj; // 自动拆箱

自动装箱和拆箱的原理

自动装箱时编译器调用valueOf将原始类型值转换成对象，同时自动拆箱时，编译器通过调用类似intValue(),doubleValue()这类的方法将对象转换成原始类型值。

明白自动装箱和拆箱的原理后，我们带着上面的疑问进行分析下Integer的自动装箱的实现源码。如下：

public static Integer valueOf(int i) {

  //判断i是否在-128和127之间，如果不在此范围，则从IntegerCache中获取包装类的实例。否则new一个新实例

  if (i >= IntegerCache.low && i <= IntegerCache.high)

    return IntegerCache.cache[i + (-IntegerCache.low)];

  return new Integer(i);

}

//使用享元模式，来减少对象的创建（亨元设计模式大家有必要了解一下，我认为是最简单的设计模式，也许大家经常在项目中使用，不知道他的名字而已

private static class IntegerCache {

  static final int low = -128;

  static final int high;

  static final Integer cache[];

  //静态方法，类加载的时候进行初始化cache[],静态变量存放在常量池中

  static {

    // high value may be configured by property

    int h = 127;

    String integerCacheHighPropValue =

      sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");

    if (integerCacheHighPropValue != null) {

      try {

        int i = parseInt(integerCacheHighPropValue);

        i = Math.max(i, 127);

        // Maximum array size is Integer.MAX\_VALUE

        h = Math.min(i, Integer.MAX\_VALUE - (-low) -1);

      } catch( NumberFormatException nfe) {

        // If the property cannot be parsed into an int, ignore it.

      }

    }

    high = h;

    cache = new Integer[(high - low) + 1];

    int j = low;

    for(int k = 0; k < cache.length; k++)

      cache[k] = new Integer(j++);

    // range [-128, 127] must be interned (JLS7 5.1.7)

    assert IntegerCache.high >= 127;

  }

  private IntegerCache() {}

}

Integer i1 = 40; 自动装箱，相当于调用了Integer.valueOf(40);方法。首先判断i值是否在-128和127之间，如果在-128和127之间则直接从IntegerCache.cache缓存中获取指定数字的包装类；不存在则new出一个新的包装类。IntegerCache内部实现了一个Integer的静态常量数组，在类加载的时候，执行static静态块进行初始化-128到127之间的Integer对象，存放到cache数组中。cache属于常量，存放在java的方法区中。

引用类型

数组、String、所有的类变量都是引用类型，默认值都为null，理论上Object类是所有引用类型的父类，即直接或间接的继承java.lang.Object类。由于所有的类包括数组都继承了Object类，因此省略了extends Object关键字。对Object类的学习参考源码及注释。

异常机制

异常机制已经成为判断一门编程语言是否成熟的标准，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异常机制主要依赖于try、catch、finally、throw、throws五个关键字。

1.try：它里面放置可能引发异常的代码

2.catch：后面对应异常类型和一个代码块，用于表明该catch块用于处理这种类型的代码块，可以有多个catch块。

3.finally：主要用于回收在try块里打开的物力资源（如数据库连接、网络连接和磁盘文件），异常机制总是保证finally块总是被执行，除非在try块或者catch块中调用了退出虚拟机的方法即System.exit(1)。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，所以一般情况下，不要再finally块中使用renturn或throw等导致方法终止的语句，因为一旦使用，将会导致try块、catch块中的return、throw语句失效。

4.throw：用于抛出一个实际的异常，可以单独作为语句使用，抛出一个具体的异常对象。

5.throws：用在方法签名中，用于声明该方法可能抛出的异常，如果某段代码调用了一个带throws声明的方法，该方法声明抛出了Checked异常，这表明该方法希望它的调用者来处理该异常。那么这段代码要么放在try块中显示捕获该异常，要么这段代码处于另一个带throws声明抛出的方法中，如检查性异常io异常，使用的方法后有throws 。

Java的异常分为两种，checked异常（编译时异常）和Runtime异常（运行时异常）

1.       java认为checked异常都是可以再编译阶段被处理的异常，所以它强制程序处理所有的checked异常，而Runtime异常无须处理，java程序必须显式处理checked异常，如果程序没有处理，则在编译时会发生错误，无法通过编译。

2.       checked异常体现了java设计哲学：没有完善处理的代码根本不会被执行，体现了java的严谨性，

对于构造大型、健壮、可维护的应用系统而言，错误处理是整个应用需要考虑的重要方面。Java异常处理机制，在程序运行出现意外时，系统会生成一个Exception对象，来通知程序，从而实现将“业务功能实现代码”和“错误处理代码”分离，提供更好的可读性。

如果执行try块里的业务逻辑代码时出现异常，系统会自动生成一个异常对象，该异常对象被提交给运行环境，这个过程被称为抛出（throw）异常。Java环境收到异常对象时，会寻找合适的catch块，如果找不到，java运行环境就会终止，java程序将退出。

     不同的catch块，视为了针对不同的异常类，提供不同的处理方法。

对于错误处理机制，主要有如下的两个缺点：

1.无法穷举所有异常情况：因为人类的知识是有限的，异常情况总比可以考虑到的情况多，总有漏网之鱼

2.错误处理代码和业务实现代码混杂严重影响程序的可读性，会增加程序维护的难度。

1.使用try...catch捕获异常

java提出了一种假设，如果程序可以顺利完成，那么一切正常，把系统的业务实现代码放在try块中定义，所有的异常处理逻辑放在catch块中进行处理。

即：try{

//业务实现代码

...

}

catch(Exception e){

输入不合法

}

上面的格式中try块和catch块后的{...}都是不可以省略的！

执行步骤：

1.如果执行try块中的业务逻辑代码时出现异常，系统自动生成一个异常对象，该异常对象被提交给java运行环境，这个过程称为抛出（throw）异常。

2.当java运行环境收到异常对象时，会寻找能处理该异常对象的catch块，如果找到合适的cathc块并把该异常对象交给catch块处理，那这个过程称为捕获（catch）异常；如果java运行时环境找不到捕获异常的catch块，则运行时环境终止，jav程序也将退出。

注意1：不管程序代码块是否处于try块中，甚至包括catch块中代码，只要执行该代码时出现了异常，系统都会自动生成一个异常对象，如果程序没有为这段代码定义任何catch块，java运行环境肯定找不到处理该异常的catch块，程序肯定在此退出。

注意2：try块后可以有多个catch块，try块后使用多个catch块是为了针对不同异常类提供的不同的异常处理方式。当系统发生不同意外情况时，系统会生成不同的异常对象，java运行时就会根据该异常对象所属的异常类来决定使用哪个catch块来处理该异常。

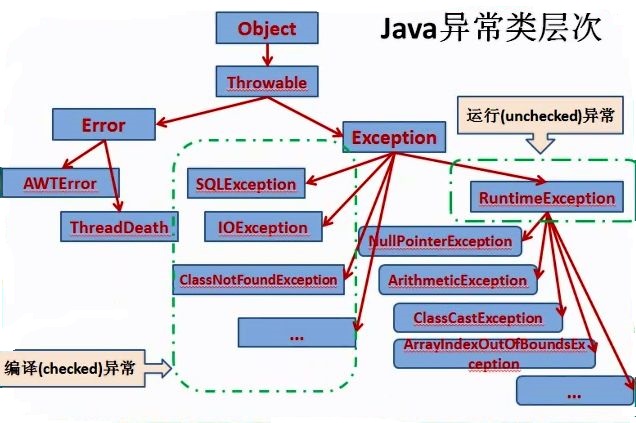
注意3：通常情况下，如果try块被执行一次，则try块后只有一个catch块会被执行，绝不可能有多个catch块被执行，除非在循环中使用类continue开始下一次循环，下一次循环又重新运行了try块，这才可能导致多个catch块被执行。

注意4：进行异常捕获时，一定要记住先捕获小的异常，再捕获大的异常。

异常类

封装异常信息的类

java内置的异常类IMG_256

IMG_256IMG_256IMG_256

IMG_256IMG_256IMG_256

  java把所有非正常情况分成两种：异常（Exception）和错误（Error），都是继承自Throwable父类。

 Error错误：一般是指虚拟机相关的问题，如系统崩溃，虚拟机出错误等，这种错误无法恢复或不可能捕获，将导致应用程序中断，通常不处理。

         Throwable（）：Throwable 类是 Java 语言中所有错误或异常的超类。只有当对象是此类（或其子类之一）的实例时，才能通过 Java 虚拟机或者 Java throw 语句抛出。类似地，只有此类或其子类之一才可以是 catch 子句中的参数类型。

         1.Error（错误）：一般是指java虚拟机相关的问题，如系统崩溃、虚拟机出错误、动态链接失败等，这种错误无法恢复或不可能捕获，将导致应用程序中断，通常应用程序无法处理这些错误，因此应用程序不应该捕获Error对象，也无须在其throws子句中声明该方法抛出任何Error或其子类。

         2.Exception：Exception 类及其子类是 Throwable 的一种形式，它指出了合理的应用程序想要捕获的条件

         （1）. SQLException：该异常提供关于数据库访问错误或其他错误的信息。

         （2）. RuntimeException 是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类

         （3）.IOException：此类为异常的通用类，它是由失败的或中断的 I/O 操作生成的。

异常对象包含的常用方法：

1.       getMessage（）；返回该异常的详细描述字符

2.       printStackTrace（）：将该异常的跟踪栈信息输出到标准错误输出。

3.       printStackTrace（PrintStream s）：将该异常的跟踪栈信息输出到指定的输出流

4.       getStackTrace（）：返回该异常的跟踪栈信息。

自定义异常类

用户自定义异常都应该继承Exception基类，如果希望自定义Runtime异常，则应该继承RuntimeException基类。

异常在事务管理中的应用

|  |
| --- |
| Connection conn = getConnection();  try {  conn.commit();1  } catch (Exception e) {  conn.rollback();2  }finally {  conn .close() ;3  } |

原理是1处操作发生异常，程序抛出一个异常被2处扑捉到进行回滚。如果在1处再套一层try catch如下,这样2处则不会执行

|  |
| --- |
| Connection conn = getConnection();  try {  try{  conn.commit();1  }catch(Exception e){  e.printStackTrace();  }  } catch (Exception e) {  conn.rollback();2  }finally {  conn .close() ;3  } |

在实际开发中一般用spring声明式事务，1处的操作就是service层代码，为了事务的正常生效，当有try catch后捕获了异常，事务不会回滚，如果不得不在service层写try catch 需要catch后 throw new RuntimeException() 让事务回滚,如下

|  |
| --- |
| Connection conn = getConnection();  try {  try{  conn.commit();1  }catch(Exception e){  throw new  [Global](C:/Program%20Files/Dict/8.5.1.0/resultui/html/index.html" \l "/javascript:;)Exception(“统一异常处理”)  }  } catch (Exception e) {  conn.rollback();2  }finally {  conn .close() ;3  } |

spring通过异常进行事务回滚的机制：

1、spring 的默认事务机制，当出现unchecked异常时候回滚，checked异常的时候不会回滚；

2、我们有时为了打印日志，会在service层抓住Exception并打印日志，这时我们的所有异常都会被认为成checked异常。

3、为了事务的正常生效：当有try catch后捕获了异常，事务不会回滚，如果不得不在service层写try catch 需要catch后 throw new RuntimeException 让事务回滚；

例子：

try {

} catch (Exception e) {

e.printStackTrace();

logger.error("发生异常");

throw new RuntimeException();

}

如何保证在有返回值的方法中进行异常的捕获？

方法1：

因为要保证当前方法具有返回值，在前端界面展示操作状态。所以我们service层在添加了try catch日志打印后，抛出的runtime类异常需要在controller层进行捕获，捕获之后，在catch中编写操作失败后返回值的信息。

方法2：

不依赖于spring的异常捕获机制进行事务回滚，通过手动的session.rollback进行异常捕获后回滚事务也可。

修饰符

修饰符用来定义类、方法或者变量，通常放在语句的最前端

访问控制修饰符

| **修饰符** | **当前类** | **同包** | **子类** | **其他包** |
| --- | --- | --- | --- | --- |
| public | √ | √ | √ | √ |
| protected | √ | √ | √ | × |
| default | √ | √ | × | × |
| private | √ | × | × | × |

这个范围指的是实例所在的位置，如下person在Main中，所有person的private不可调用，public在哪都可以。

|  |
| --- |
| public class Main {  public static void main(String[] args) {  Person person = new Person();  person.say1();  person.say2();//编译无法通过  int age1 = person.age1;  int age2 = person.age2;//编译无法通过  }  }  class Person {  private String name;  public int age1;  private int age2;  public void say1() {  System.out.println("say1");  }  private void say2() {  System.out.println("say2");  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  } |

Jdk中有的类的构造器是用private修饰的，目的就是不让用户在其他地方实例化，如Math类，这种类要么是其中的方法都是静态的直接调用不需要实例化，要么就是为了保证单例， 通过类中的单例模式获取实例。

访问控制和继承

1. 父类中声明为public的方法在子类中也必须为public。
2. 父类中声明为protected的方法在子类中要么声明为protected，要么声明为public。不能声明为private。很少用protectedObject类中对clone方法的声明即用到了protected访问修饰符，这是因为Object类的clone方法只能实现浅克隆，而并不能实现常使用的深克隆，这就要求子类在需要克隆对象时尽量重写clone方法，此时即声明为protected的，以保证在需要克隆对象时，必须要求待克隆对象所在的类实现Cloneable接口并重写clone方法。（浅：引用，深：新的对象）
3. 父类中声明为private的方法，不能够被继承。

非访问修饰符

为了实现一些其他的功能，Java也提供了许多非访问修饰符

static

　“static方法就是没有this的方法。在static方法内部不能调用非静态方法，反过来是可以的。而且可以在没有创建任何对象的前提下，仅仅通过类本身来调用static方法。这实际上正是static方法的主要用途。”

　　这段话虽然只是说明了static方法的特殊之处，但是可以看出static关键字的基本作用，简而言之，一句话来描述就是：

　　方便在没有创建对象的情况下来进行调用（方法/变量）。

　　很显然，被static关键字修饰的方法或者变量不需要依赖于对象来进行访问，只要类被加载了，就可以通过类名去进行访问。

　　static可以用来修饰类的成员方法、类的成员变量，另外可以编写static代码块来优化程序性能。

1）static方法

　　static方法一般称作静态方法，由于静态方法不依赖于任何对象就可以进行访问，因此对于静态方法来说，是没有this的，因为它不依附于任何对象，既然都没有对象，就谈不上this了。并且由于这个特性，在静态方法中不能访问类的非静态成员变量和非静态成员方法，因为非静态成员方法/变量都是必须依赖具体的对象才能够被调用。

　　但是要注意的是，虽然在静态方法中不能访问非静态成员方法和非静态成员变量，但是在非静态成员方法中是可以访问静态成员方法/变量的。

而对于非静态成员方法，它访问静态成员方法/变量显然是毫无限制的。

　　因此，如果说想在不创建对象的情况下调用某个方法，就可以将这个方法设置为static。我们最常见的static方法就是main方法，至于为什么main方法必须是static的，现在就很清楚了。因为程序在执行main方法的时候没有创建任何对象，因此只有通过类名来访问。

2）static变量

　　static变量也称作静态变量，静态变量和非静态变量的区别是：静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。而非静态变量是对象所拥有的，在创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响。static是不允许用来修饰局部变量。不要问为什么，这是Java语法的规定.

　　static成员变量的初始化顺序按照定义的顺序进行初始化。

3）static代码块

　　static关键字还有一个比较关键的作用就是 用来形成静态代码块以优化程序性能。static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。

　　为什么说static块可以用来优化程序性能，是因为它的特性:只会在类加载的时候执行一次。

static加载顺序问题

**public class** Test {  
 Person **person** = **new** Person(**"Test"**);  
 **static**{  
 System.***out***.println(**"test static"**);  
 }  
 **public** Test() {  
 System.***out***.println(**"test constructor"**);  
 }  
 **public static void** main(String[] args) {  
 **new** MyClass();  
 }  
}  
**class** Person{  
 **static**{  
 System.***out***.println(**"person static"**);  
 }  
 **public** Person(String str) {  
 System.***out***.println(**"person "**+str);  
 }  
}  
  
**class** MyClass **extends** Test {  
 Person **person** = **new** Person(**"MyClass"**);  
 **static**{  
 System.***out***.println(**"myclass static"**);  
 }  
 **public** MyClass() {  
 System.***out***.println(**"myclass constructor"**);  
 }  
}

分析：执行Test类中main方法，首先加载Test类，因此会执行Test类中的static块。接着执行new MyClass()，而MyClass类还没有被加载，因此需要加载MyClass类。在加载MyClass类的时候，发现MyClass类继承自Test类，但是由于Test类已经被加载了，所以只需要加载MyClass类，那么就会执行MyClass类的中的static块。在加载完之后，就通过构造器来生成对象。而在生成对象的时候，必须先初始化父类的成员变量，因此会执行Test中的Person person = new Person()，而Person类还没有被加载过，因此会先加载Person类并执行Person类中的static块，接着执行父类的构造器，完成了父类的初始化，然后就来初始化自身了，因此会接着执行MyClass中的Person person = new Person()，最后执行MyClass的构造器。

final

A、修饰类(class)。   
     1、该类不能被继承。   
     2、类中的方法不会被覆盖，因此默认都是final的。   
     3、用途：设计类时，如果该类不需要有子类，不必要被扩展，类的实现细节不允许被改变，那么就设计成final类

B、修饰方法(method)   
     1、该方法可以被继承，但是不能被覆盖。   
     2、用途：一个类不允许子类覆盖该方法，则用final来修饰   
     3、好处：可以防止继承它的子类修改该方法的意义和实现；更为高效，编译器在遇到调用fianal方法转入内嵌机制，提高了执行效率。   
     4、注意：父类中的private成员方法不能被子类覆盖，因此，private方法默认是final型的(可以查看编译后的class文件)

C、修饰变量(variable)   
1、用final修饰后变为常量。包括静态变量、实例变量和局部变量这三种。   
2、特点：可以先声明，不给初值，这种叫做final空白。但是使用前必须被初始化,可以在域的定义处或每个构造器中用表达式为final赋值，一旦被赋值，将不能再被改变。

3、注意：基本数据类型是不可以被修改了,包括String，对于其他refer 型的变量，可以修改这种对像里面的内容，也就是说final引用对象的引用不能改变，但是里面的值可以改变，.如Map,List等,即使你用final修饰了,依然可以put和add

1. 修饰参数(arguments)   
        1、用final修饰参数时，可以读取该参数，但是不能对其作出修改。

abstract修饰符

抽象类：

抽象类不能用来实例化对象，声明抽象类的唯一目的是为了将来对该类进行扩充。抽象类是存在抽象方法的，如果能让抽象类创建对象的话，那么使用抽象的对象调用抽象方法是没有任何意义的

一个类不能同时被abstract和final修饰。如果一个类包含抽象方法，那么该类一定要声明为抽象类，否则将出现编译错误。抽象类可以包含抽象方法和非抽象方法。

抽象方法

抽象方法是一种没有任何实现的方法，该方法的的具体实现由子类提供。抽象方法不能被声明成final和static。任何继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类。如果一个类包含若干个抽象方法，那么该类必须声明为抽象类。抽象类可以不包含抽象方法。

synchronized修饰符

synchronized关键字声明的方法同一时间只能被一个线程访问

transient修饰符

序列化的对象包含被transient修饰的实例变量时，java虚拟机(JVM)跳过该特定的变量。该修饰符包含在定义变量的语句中，用来预处理类和变量的数据类型。

volatile修饰符

volatile修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。一个volatile对象引用可能是null。

native

java本地接口，如Object类中 public final native Class<?> getClass();public native int hashCode();这些方法是调用外部的方法，一般是c语言写的。