**一、关键字**  
Java语言中一些具有特殊用途的单词被称作关键字，定义标识符时不要让标识符和关键字相同，以免引起错误，Java中的关键字一般都是小写，Java一共包含50个关键字，根据他们的用途，可以划分成4类。  
1、标识数据类型、对象：boolean、byte、char、double、false、float、int、long、new、null、short、true、void、instanceof；  
2、语句控制：break、case、catch、continue、default、do、else、for、if、return、switch、try、while、finally、throw；  
3、修饰功能：this、super、abstract、final、native、private、protected、public、static、synchronizet、transient、volatile；  
4、类、接口、方法、包和异常等的声明，定义要求：class、extends、implements、interface、package、import、throws。  
这些关键字我们将会在后期的文章当中分享到，现在不需要死记硬背，在后期的学习当中慢慢理解，自然而然就会掌握的熟练。  
**二、标识符**  
在Java当中，包名、类名、接口名、方法名、对象名、常量名、变量名统称为标识符。  
Java中的标识符 必须是字母、下划线、美元符号开头，后续符号还可以是数字、Unicode字符集中序号大于0xC0的所有符号，尽可作为标识符的一部分，注意关键字不能 作为关键字，只能作为标识符的一部分。  
此外，在Java中还有一些命名规则，可以让自己的程序更加要规范易懂，例如：  
1、包名通常为小写，如Java.lang 　[java.io](http://java.io/)  
2、类名、接口名的首字母通常都为大写，例如：System　、　Math  
3、方法名的第一个字母通常为小写：例如：main（）　print（）　output（）  
4、当类名、接口名、方法名由多个单词组成时，后面单词的首字母都应当大写，例如：DataOutput、getSum。（即为驼峰分布）  
5、用户声明的变量名、一个类的对象名通常为小写，例如：temp、str  
**三、常量**  
常量是指在Java程序中，值永远不发生变化的量，除了我们上一期分享的数值常量，字符常量，布尔常量之外，我们还可以用符号常量来表示，符号常量要用关键字final来定义。  
格式：final　数据类型　常量名=值  
例如：final　double　PI　=　3.1415926

四丶JAVA的基本数据类型

在JAVA中一共有八种基本数据类型，他们分别是   
byte、short、int、long、float、double、char、boolean

五丶关系运算符

比较运算符|关系运算符

* > < >= <=
  + 操作的类型只能为 byte short int long double float char
* == !=
  + 可以比较基本数据类型,可以比较引用数据类型
* 最终的结果肯定是一个bollean类型的值

六丶构造方法相关概念

概念：表现在一个类中，与类的名称相关，不声明返回值的类型的方法

public class Car (){

publicCar(){

}

}

1，构造方法在创建类的对象时，被执行。例如：

Car car=new Car();

2，构造方法可以用于决定类的对象被初始化时需要执行的任务，例如：

public class Car{

public srting color;

        public String brand;

public Car (string carColor,string carBrand){

color=carColor;

brand=carBrand;

}

}

public class Test{

public static void main(String[] args) {

   Car car = new Car("black", "BMW");

 }

}

3，每个类中，如果没有显式的声明构造方法，则编译器会默认自动添加无参的构造方法。

public class Car{

}

public class Car{

public Car(){

}

}

以上两个了中的声明的效果完全是一致的。一旦开发者在设计类时，显式的声明一个或者多个构造方法，则编译器不会再自动添加无参构造方法。

4，在每个子类的构造方法中，编译器都会默认调用父类的无参的构造方法

public class Car{

}

publci class Car{

public Car(){

}

}

publci class Car{

public Car(){

 super();  
 }

}

以上三种类的声明完全是等效的。

5，假设某个类中，只有带参数的构造方法

public class Animal{

public Animal (String name);

}

当子类继承这个animal类时，必须显式的调用Animal 类带参数的构造方法

public class Dog extends Animal｛

public Dog(){

super("Wangcai");

}

｝

构造方法允许重载，同一个类中，可以存在多个构造方法。所以当某个类存在多个构造方法，一旦子类继承它，子类的任何构造方法中必须调用父类中的任意一个构造方法。

# 7.java 类、属性、方法修饰符总结

修饰符可以分为访问修饰符（访问控制符）和非访问修饰符（非访问控制符）

访问修饰符：顾名思义如果在访问范围内可以控制访问。

非访问修饰符：实现一些其他的功能。

**一、类修饰符**

       访问修饰符：public、默认修饰符            非访问修饰符有： abstract 、 final

      1、public（公共类修饰符/权限修饰符） ： 一个java文件可以写多个类，但是只能有一个类是用public进行修饰的（也就

            是所说的主类），且这个类需要和文件名保持一致。表示所有包中的类都可以访问。

      2 、abstract （抽象类修饰符）：用 abstract 修饰符修饰类表示该类是一个抽象类。所谓抽象类是指没有实例的抽象概念

           类 （没有具体对象）。

            注意：一个抽象类中可以没有抽象方法，但是有抽象方法的类必须是一个抽象类

      3、 final （最终类修饰符）：如果一个类被final进行修饰表示该类是最终类，不能被继承。

      4、类缺省(没有public)访问控制符：如果一个类没有访问控制符，说明它具有缺省的访问控制符特性。表示该类只能被本

           包中的其他类来进行访问。这一访问特性又称为包访问性。

在定义一个类的成员(包括属性（变量）和方法)时，可以使用private（私有修饰符）、protected（保护访问修饰符）、public（权限修饰符）等关键字说明这个成员的访问权限。

**访问范围：**

        public：可被所有类引用；  
        protected：可被该类自身、子类(只能在子类中通过子类对象来调用)、同一包中的其他类引用；  
        缺省(没有访问控制符)：可被该类自身、本包中的类引用；  
        private：仅可被该类自身引用和修改，不能被其他任何类(包括子类)引用。

**二、属性修饰符**

  访问修饰符： public、 private、 protected、默认修饰符

  非访问修饰符： static、 final 、 volatile 、 transient

      1、 public（权限修饰符） ：用 public 修饰的域称为公共域。如果公共域属于一个公共类，则可以被所有其它类所引用。

            由于 public 修饰符会降低运行的安全性和数据的封装性，所以一般应减少 public 域的使用。

      2、 private（私有修饰符） ： 仅可被该类自身引用和修改，不能被其他任何类(包括子类)引用。

      3、 protected （保护访问修饰符）可被该类自身、子类(只能在子类中通过子类对象来调用)、同一包中的其他类引用。

      4、缺省(没有访问控制符)：可被该类自身、本包中的类引用。

      5、static （静态域修饰符 ）：类成员变量，对所有的实例一致，引用是前缀可使用类名或对象名。

注意：用 static修饰的成员变量仅属于类的变量，而不属于任何一个具体的对象，静态成员变量的值是保存在类的内存区域的公共存储单元，任何一个类的对象访问它时取到的都是相同的数据；任何一个类的对象修改它时 , 也都是对同一个内存单元进行操作。

      6、 final （最终域修饰符）：数值不变常量，定义同时应对其进行初始化，不一定，也可以利用构造函数给不同的对象赋

            不同的值。

      7、 volatile （易失 ( 共享 ) 域修饰符）：是用来说明这个成员变量可能被几个线程所控制和修改。在使用中要注意这种成

            员变量取值的变化。通常 volatile 用来修饰接受外部输入的域。

      8、 transient （暂时性域修饰符）：用修饰符transient 限定的暂时性变量，将指定 Java虚拟机认定该暂时性变量不属于

             永久状态，以实现不同对象的存档功能。否则，类中所有变量都是对象的永久状态的一部分，存储对象时必须同时

            保存这些变量。

**三、方法修饰符**

   访问修饰符：public 、 private、 protected 、 默认修饰符

   非访问修饰符：abstract、static、 final、 native 、synchronized

      1、 abstract （抽象方法控制符）：抽象方法，只有方法的定义而没有方法体和操作实现。需要在子类中通过重写该方法

          来实现。

      2、 static（静态方法控制符） ：静态方法，又称类方法。前缀可使用类名或对象名。在方法体中只能使用类的static变量

          和static方法。静态方法是属于整个类的类方法；而不使用static 修饰、限定的方法是属于某个具体类对象的方法。

      3、 final （最终方法修饰符）：所有被 private 修饰符限定为私有的方法，以及所有包含在 final 类 中的方法，都被认为是

           最终方法。最终方法是功能和内部语句不能更改的方法，即最终方法不能重载（子类中不能重新定义该方法）。固定了

           这个方法所具有的功能和操作，防止当前类的子类对父类关键方法的错误定义，保证了程序的安全性和正确性。

      4、 native（本地方法修饰符）：为了提高程序的运行速度，需要用其它的高级语言书写程序的方法体，那么该方法可定义

          为本地方法用修饰符 native 来修饰。

      5、 synchronized（同步方法修饰符） ：该修饰符主要用于多线程共存的程序中的协调和同步。

# 八丶this和super

### this关键字

* 表示当前对象
* 区分同名变量
* 作为方法名初始化对象，放在构造方法第一句，相当于调用同类的其他构造方法（重载的构造器）
* 作为参数传递，需要在某些完全分离的类中调用一个方法，并将当前对象的一个引用作为参数传递时

### super关键字

* 访问父类的变量和方法
* 调用父类中的某一个构造函数（应该为构造函数中的第一条语句，并且只能调用一次）。

### this和super的区别

* super()主要是对父类构造函数的调用，this()是对重载构造函数的调用
* super()主要是在继承了父类的子类的构造函数中使用，是在不同类中的使用；this()主要是在同一类的不同构造函数中的使用

# 九丶上转型对象声明和使用

## 1. 定义

如果B类是A类的子类或间接子类，当用B类创建对象b并将这个对象b的引用赋给A类对象a时，如：

Parent p;  
Son s = new Son();

p=s;

则称Parent**类对象p是子类Son对象s的上转型对象。**

## 2. 性质

对象b的上转型a的实体是有子类B创建的，但是上转型对象会失去子类B的一些属性和功能。上转型对象具有以下特点：

1. 上转型对象不能操作子类新增加的成员变量，不能使用子类新增的方法。即为较子类B失去一些属性和功能，这些属性和功能是新增的。
2. 上转型对象可以操作子类继承或隐藏的成员变量，也可以使用子类继承的或重写的方法。即为上转型对象可以操纵父类原有的属性和功能，无论这些方法是否被重写。
3. 上转型对象调用方法时，就是调用子类继承和重写过的方法。而不会是新增的方法，也不是父类原有的方法。
4. 可以将对象的上转型对象再强制转换到一个子类对象，强制转换过的对象具有子类所有属性和功能。

# [java中File类的常用所有方法及其应用](https://www.cnblogs.com/wangjinlong-padan/p/6382924.html)

创建：  
createNewFile()在指定位置创建一个空文件，成功就返回true，如果已存在就不创建，然后返回false。  
mkdir()  在指定位置创建一个单级文件夹。  
mkdirs()  在指定位置创建一个多级文件夹。  
renameTo(File dest)如果目标文件与源文件是在同一个路径下，那么renameTo的作用是重命名， 如果目标文件与源文件不是在同一个路径下，那么renameTo的作用就是剪切，而且还不能操作文件夹。

删除：  
delete()  删除文件或者一个空文件夹，不能删除非空文件夹，马上删除文件，返回一个布尔值。  
deleteOnExit()jvm退出时删除文件或者文件夹，用于删除临时文件，无返回值。  
 判断：  
exists()  文件或文件夹是否存在。  
isFile()  是否是一个文件，如果不存在，则始终为false。  
isDirectory()  是否是一个目录，如果不存在，则始终为false。  
isHidden()  是否是一个隐藏的文件或是否是隐藏的目录。  
isAbsolute()  测试此抽象路径名是否为绝对路径名。  
 获取：  
getName()  获取文件或文件夹的名称，不包含上级路径。  
getAbsolutePath()获取文件的绝对路径，与文件是否存在没关系  
length()  获取文件的大小（字节数），如果文件不存在则返回0L，如果是文件夹也返回0L。  
getParent()  返回此抽象路径名父目录的路径名字符串；如果此路径名没有指定父目录，则返回null。  
lastModified()获取最后一次被修改的时间。

文件夹相关：  
static File[] listRoots()列出所有的根目录（Window中就是所有系统的盘符）  
list()  返回目录下的文件或者目录名，包含隐藏文件。对于文件这样操作会返回null。  
listFiles()  返回目录下的文件或者目录对象（File类实例），包含隐藏文件。对于文件这样操作会返回null。  
list(FilenameFilter filter)返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回null。  
listFiles(FilenameFilter filter)返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回null。

### 二、Comparator简介

　　Comparator是比较接口，我们如果需要控制某个类的次序，而该类本身不支持排序(即没有实现Comparable接口)，那么我们就可以建立一个“该类的比较器”来进行排序，这个“比较器”只需要实现Comparator接口即可。也就是说，我们可以通过实现Comparator来新建一个比较器，然后通过这个比较器对类进行排序。该接口定义如下：

package java.util;

public interface Comparator<T>

{

    int compare(T o1, T o2);

    boolean equals(Object obj);

}

注意：1、若一个类要实现Comparator接口：它一定要实现compare(T o1, T o2) 函数，但可以不实现 equals(Object obj) 函数。

　　　2、int compare(T o1, T o2) 是“比较o1和o2的大小”。返回“负数”，意味着“o1比o2小”；返回“零”，意味着“o1等于o2”；返回“正数”，意味着“o1大于o2”。

　　现在假如上面的Person类没有实现Comparable接口，该如何比较大小呢？我们可以新建一个类，让其实现Comparator接口，从而构造一个“比较器"。

[复制代码](javascript:void(0);)

public class PersonCompartor implements Comparator<Person>

{

@Override

public int compare(Person o1, Person o2)

{

return o1.getAge()-o2.getAge();

}

}

[复制代码](javascript:void(0);)

　　现在我们就可以利用这个比较器来对其进行排序：

[复制代码](javascript:void(0);)

public class Person

{

String name;

int age;

public Person(String name, int age)

{

super();

this.name = name;

this.age = age;

}

public String getName()

{

return name;

}

public int getAge()

{

return age;

}

public static void main(String[] args)

{

Person[] people=new Person[]{new Person("xujian", 20),new Person("xiewei", 10)};

System.out.println("排序前");

for (Person person : people)

{

System.out.print(person.getName()+":"+person.getAge());

}

Arrays.sort(people,new PersonCompartor());

System.out.println("\n排序后");

for (Person person : people)

{

System.out.print(person.getName()+":"+person.getAge());

}

}

}

[复制代码](javascript:void(0);)

final 在 [Java](http://c.biancheng.net/java/" \t "_blank) 中的意思是最终，也可以称为完结器，表示对象是最终形态的，不可改变的意思。final 应用于类、方法和变量时意义是不同的，但本质是一样的，都表示不可改变，类似 [C#](http://c.biancheng.net/csharp/" \t "_blank) 里的 sealed 关键字。  
  
使用 final 关键字声明类、变量和方法需要注意以下几点：

* final 用在变量的前面表示变量的值不可以改变，此时该变量可以被称为常量。
* final 用在方法的前面表示方法不可以被重写（子类中如果创建了一个与父类中相同名称、相同返回值类型、相同参数列表的方法，只是方法体中的实现不同，以实现不同于父类的功能，这种方式被称为方法重写，又称为方法覆盖。这里了解即可，教程后面我们会详细讲解）。
* final 用在类的前面表示该类不能有子类，即该类不可以被继承。

## final 修饰变量

final 修饰的变量即成为常量，只能赋值一次，但是 final 所修饰局部变量和成员变量有所不同。

1. final 修饰的局部变量必须使用之前被赋值一次才能使用。
2. final 修饰的成员变量在声明时没有赋值的叫“空白 final 变量”。空白 final 变量必须在构造方法或静态代码块中初始化。

注意：final 修饰的变量不能被赋值这种说法是错误的，严格的说法是，final 修饰的变量不可被改变，一旦获得了初始值，该 final 变量的值就不能被重新赋值。

1. **public** **class** FinalDemo {
2. void doSomething() {
3. // 没有在声明的同时赋值
4. **final** int e;
5. // 只能赋值一次
6. e = 100;
7. System.out.print(e);
8. // 声明的同时赋值
9. **final** int f = 200;
10. }
11. // 实例常量
12. **final** int a = 5; // 直接赋值
13. **final** int b; // 空白final变量
14. // 静态常量
15. **final** **static** int c = 12;// 直接赋值
16. **final** **static** int d; // 空白final变量
17. // 静态代码块
18. **static** {
19. // 初始化静态变量
20. d = 32;
21. }
22. // 构造方法
23. FinalDemo() {
24. // 初始化实例变量
25. b = 3;
26. // 第二次赋值，会发生编译错误
27. // b = 4;
28. }
29. }

上述代码第 4 行和第 6 行是声明局部常量，其中第 4 行只是声明没有赋值，但必须在使用之前赋值（见代码第 6 行），其实局部常量最好在声明的同时初始化。代码第 13、14、16 和 17 行都声明成员常量。代码第 13 和 14 行是实例常量，如果是空白 final 变量（见代码第 14 行），则需要在构造方法中初始化（见代码第 27 行）。代码第 16 和 17 行是静态常量，如果是空白 final 变量（见代码第 17 行），则需要在静态代码块中初始化（见代码第 21 行）。  
  
另外，无论是那种常量只能赋值一次，见代码第 29 行为 b 常量赋值，因为之前 b 已经赋值过一次，因此这里会发生编译错误。

#### final 修饰基本类型变量和引用类型变量的区别

当使用 final 修饰基本类型变量时，不能对基本类型变量重新赋值，因此基本类型变量不能被改变。 但对于引用类型变量而言，它保存的仅仅是一个引用，final 只保证这个引用类型变量所引用的地址不会改变，即一直引用同一个对象，但这个对象完全可以发生改变。  
  
下面程序示范了 final 修饰数组和 Person 对象的情形。

1. import java.util.Arrays;
2. **class** Person {
3. **private** int age;
4. **public** Person() {
5. }
6. // 有参数的构造器
7. **public** Person(int age) {
8. **this**.age = age;
9. }
10. // 省略age的setter和getter方法
11. // age 的 setter 和 getter 方法
12. }
13. **public** **class** FinalReferenceTest {
14. **public** **static** void main(String[] args) {
15. // final修饰数组变量，iArr是一个引用变量
16. **final** int[] iArr = { 5, 6, 12, 9 };
17. System.out.println(Arrays.toString(iArr));
18. // 对数组元素进行排序，合法
19. Arrays.sort(iArr);
20. System.out.println(Arrays.toString(iArr));
21. // 对数组元素赋值，合法
22. iArr[2] = -8;
23. System.out.println(Arrays.toString(iArr));
24. // 下面语句对iArr重新赋值,非法
25. // iArr = null;
26. // final修饰Person变量，p是一个引用变量
27. **final** Person p = **new** Person(45);
28. // 改变Person对象的age实例变量，合法
29. p.setAge(23);
30. System.out.println(p.getAge());
31. // 下面语句对P重新赋值，非法
32. // p = null;
33. }
34. }

从上面程序中可以看出，使用 final 修饰的引用类型变量不能被重新赋值，但可以改变引用类型变量所引用对象的内容。例如上面 iArr 变量所引用的数组对象，final 修饰后的 iArr 变量不能被重新赋值，但 iArr 所引用数组的数组元素可以被改变。与此类似的是，p 变量也使用了 final 修饰，表明 p 变量不能被重新赋值，但 p 变量所引用 Person 对象的成员变量的值可以被改变。  
  
注意：在使用 final 声明变量时，要求全部的字母大写，如 SEX，这点在开发中是非常重要的。  
  
如果一个程序中的变量使用 public static final 声明，则此变量将称为全局变量，如下面的代码：

public static final String SEX= "女";

## final修饰方法

final 修饰的方法不可被重写，如果出于某些原因，不希望子类重写父类的某个方法，则可以使用 final 修饰该方法。  
  
Java 提供的 Object 类里就有一个 final 方法 getClass()，因为 Java 不希望任何类重写这个方法，所以使用 final 把这个方法密封起来。但对于该类提供的 toString() 和 equals() 方法，都允许子类重写，因此没有使用 final 修饰它们。  
  
下面程序试图重写 final 方法，将会引发编译错误。

1. **public** **class** FinalMethodTest {
2. **public** **final** void test() {
3. }
4. }
5. **class** Sub **extends** FinalMethodTest {
6. // 下面方法定义将出现编译错误，不能重写final方法
7. **public** void test() {
8. }
9. }

上面程序中父类是 FinalMethodTest，该类里定义的 test() 方法是一个 final 方法，如果其子类试图重写该方法，将会引发编译错误。  
  
对于一个 private 方法，因为它仅在当前类中可见，其子类无法访问该方法，所以子类无法重写该方法——如果子类中定义一个与父类 private 方法有相同方法名、相同形参列表、相同返回值类型的方法，也不是方法重写，只是重新定义了一个新方法。因此，即使使用 final 修饰一个 private 访问权限的方法，依然可以在其子类中定义与该方法具有相同方法名、相同形参列表、相同返回值类型的方法。  
  
下面程序示范了如何在子类中“重写”父类的 private final 方法。

1. **public** **class** PrivateFinalMethodTest {
2. **private** **final** void test() {
3. }
4. }
5. **class** Sub **extends** PrivateFinalMethodTest {
6. // 下面的方法定义不会出现问题
7. **public** void test() {
8. }
9. }

上面程序没有任何问题，虽然子类和父类同样包含了同名的 void test() 方法，但子类并不是重写父类的方法，因此即使父类的 void test() 方法使用了 final 修饰，子类中依然可以定义 void test() 方法。  
  
final 修饰的方法仅仅是不能被重写，并不是不能被重载，因此下面程序完全没有问题。

1. **public** **class** FinalOverload {
2. // final 修饰的方法只是不能被重写，完全可以被重载
3. **public** **final** void test(){}
4. **public** **final** void test(String arg){}
5. }

## final修饰类

final 修饰的类不能被继承。当子类继承父类时，将可以访问到父类内部数据，并可通过重写父类方法来改变父类方法的实现细节，这可能导致一些不安全的因素。为了保证某个类不可被继承，则可以使用 final 修饰这个类。  
  
下面代码示范了 final 修饰的类不可被继承。

1. **final** **class** SuperClass {
2. }
3. **class** SubClass **extends** SuperClass { //编译错误
4. }

因为 SuperClass 类是一个 final 类，而 SubClass 试图继承 SuperClass 类，这将会引起编译错误。

## final 修饰符使用总结

#### 1. final 修饰类中的变量

表示该变量一旦被初始化便不可改变，这里不可改变的意思对基本类型变量来说是其值不可变，而对对象引用类型变量来说其引用不可再变。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；二是在构造方法中。这两个地方只能选其一，要么在定义时给值，要么在构造方法中给值，不能同时既在定义时赋值，又在构造方法中赋予另外的值。

#### 2. final 修饰类中的方法

说明这种方法提供的功能已经满足当前要求，不需要进行扩展，并且也不允许任何从此类继承的类来重写这种方法，但是继承仍然可以继承这个方法，也就是说可以直接使用。在声明类中，一个 final 方法只被实现一次。

#### 3. final 修饰类

# 表示该类是无法被任何其他类继承的，意味着此类在一个继承树中是一个叶子类，并且此类的设计已被认为很完美而不需要进行修改或扩展。 对于 final 类中的成员，可以定义其为 final，也可以不是 final。而对于方法，由于所属类为 final 的关系，自然也就成了 final 型。也可以明确地给 final 类中的方法加上一个 final，这显然没有意义。

.

[Java](http://c.biancheng.net/java/" \t "_blank) 语言提供了两种类，分别为具体类和抽象类。前面学习接触的类都是具体类。这一节介绍一下抽象类。  
  
在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，那么这样的类称为抽象类。  
  
在 Java 中抽象类的语法格式如下：

<abstract>class<class\_name> {

<abstract><type><method\_name>(parameter-iist);

}

其中，abstract 表示该类或该方法是抽象的；class\_name 表示抽象类的名称；method\_name 表示抽象方法名称，parameter-list 表示方法参数列表。  
  
如果一个方法使用 abstract 来修饰，则说明该方法是抽象方法，抽象方法只有声明没有实现。需要注意的是 abstract 关键字只能用于普通方法，不能用于 static 方法或者构造方法中。  
  
抽象方法的 3 个特征如下：

1. 抽象方法没有方法体
2. 抽象方法必须存在于抽象类中
3. 子类重写父类时，必须重写父类所有的抽象方法

注意：在使用 abstract 关键字修饰抽象方法时不能使用 private 修饰，因为抽象方法必须被子类重写，而如果使用了 private 声明，则子类是无法重写的。  
  
抽象类的定义和使用规则如下：

1. 抽象类和抽象方法都要使用 abstract 关键字声明。
2. 如果一个方法被声明为抽象的，那么这个类也必须声明为抽象的。而一个抽象类中，可以有 0~n 个抽象方法，以及 0~n 个具体方法。
3. 抽象类不能实例化，也就是不能使用 new 关键字创建对象。

#### 例 1

不同几何图形的面积计算公式是不同的，但是它们具有的特性是相同的，都具有长和宽这两个属性，也都具有面积计算的方法。那么可以定义一个抽象类，在该抽象类中含有两个属性（width 和 height）和一个抽象方法 area( )，具体步骤如下。  
  
1）首先创建一个表示图形的抽象类 Shape，代码如下所示。

1. **public** **abstract** **class** Shape {
2. **public** int width; // 几何图形的长
3. **public** int height; // 几何图形的宽
4. **public** Shape(int width, int height) {
5. **this**.width = width;
6. **this**.height = height;
7. }
8. **public** **abstract** double area(); // 定义抽象方法，计算面积
9. }

2）定义一个正方形类，该类继承自形状类 Shape，并重写了 area( ) 抽象方法。正方形类的代码如下：

1. **public** **class** Square **extends** Shape {
2. **public** Square(int width, int height) {
3. **super**(width, height);
4. }
5. // 重写父类中的抽象方法，实现计算正方形面积的功能
6. @Override
7. **public** double area() {
8. **return** width \* height;
9. }
10. }

3）定义一个三角形类，该类与正方形类一样，需要继承形状类 Shape，并重写父类中的抽象方法 area()。三角形类的代码实现如下：

1. **public** **class** Triangle **extends** Shape {
2. **public** Triangle(int width, int height) {
3. **super**(width, height);
4. }
5. // 重写父类中的抽象方法，实现计算三角形面积的功能
6. @Override
7. **public** double area() {
8. **return** 0.5 \* width \* height;
9. }
10. }

4）最后创建一个测试类，分别创建正方形类和三角形类的对象，并调用各类中的 area() 方法，打印出不同形状的几何图形的面积。测试类的代码如下：

1. **public** **class** ShapeTest {
2. **public** **static** void main(String[] args) {
3. Square square = **new** Square(5, 4); // 创建正方形类对象
4. System.out.println("正方形的面积为：" + square.area());
5. Triangle triangle = **new** Triangle(2, 5); // 创建三角形类对象
6. System.out.println("三角形的面积为：" + triangle.area());
7. }
8. }

在该程序中，创建了 4 个类，分别为图形类 Shape、正方形类 Square、三角形类 Triangle 和测试类 ShapeTest。其中图形类 Shape 是一个抽象类，创建了两个属性，分别为图形的长度和宽度，并通过构造方法 Shape( ) 给这两个属性赋值。  
  
在 Shape 类的最后定义了一个抽象方法 area( )，用来计算图形的面积。在这里，Shape 类只是定义了计算图形面积的方法，而对于如何计算并没有任何限制。也可以这样理解，抽象类 Shape 仅定义了子类的一般形式。  
  
正方形类 Square 继承抽象类 Shape，并实现了抽象方法 area( )。三角形类 Triangle 的实现和正方形类相同，这里不再介绍。

在测试类 ShapeTest 的 main( ) 方法中，首先创建了正方形类和三角形类的实例化对象 square 和 triangle，然后分别调用 area( ) 方法实现了面积的计算功能。  
  
5）运行该程序，输出的结果如下：

正方形的面积为：20.0

三角形的面积为：5.0

重载(Overloading)

 方法重载是让类以统一的方式处理不同类型数据的一种手段。多个同名函数同时存在，具有不同的参数个数/类型。

重载Overloading是一个类中多态性的一种表现。

[Java](http://java.chinaitlab.com/" \t "_blank)的方法重载，就是在类中可以创建多个方法，它们具有相同的名字，但具有不同的参数和不同的定义。

调用方法时通过传递给它们的不同参数个数和参数类型来决定具体使用哪个方法, 这就是多态性。

 重载的时候，方法名要一样，但是参数类型和个数不一样，返回值类型可以相同也可以不相同。无法以返回型别作为重载函数的区分标准。

父类方法被默认修饰时，只能在同一包中，被其子类被重写，如果不在同一包则不能重写。

父类的方法被protoeted时，不仅在同一包中，被其子类被重写，还可以不同包的子类重写。

重写方法的规则：

1）、参数列表必须完全与被重写的方法相同，否则不能称其为重写而是重载。

2）、返回的类型必须一直与被重写的方法的返回类型相同，否则不能称其为重写而是重载。

3）、访问修饰符的限制一定要大于被重写方法的访问修饰符（public>protected>default>private）

4）、重写方法一定不能抛出新的检查异常或者比被重写方法申明更加宽泛的检查型异常。例如：

父类的一个方法申明了一个检查异常IOException，在重写这个方法是就不能抛出Exception,只能抛出IOException的子类异常，可以抛出非检查异常。

而重载的规则：

1）、必须具有不同的参数列表；

2）、可以有不同的返回类型，只要参数列表不同就可以了；

3）、可以有不同的访问修饰符；

4）、可以抛出不同的异常；

**重载和重写（覆盖）的特点：**

1. Override 特点

　　1、覆盖的方法的标志必须要和被覆盖的方法的标志完全匹配，才能达到覆盖的效果；

　　2、覆盖的方法的返回值必须和被覆盖的方法的返回一致；

　　3、覆盖的方法所抛出的异常必须和被覆盖方法的所抛出的异常一致，或者是其子类；

　　4、被覆盖的方法不能为private，否则在其子类中只是新定义了一个方法，并没有对其进行覆盖。

2.Overload 特点

　　1、在使用重载时只能通过不同的参数样式。例如，不同的参数类型，不同的参数个数，不同的参数顺序（当然，同一方法内的几个参数类型必须不一样，例如可以是fun(int, float)， 但是不能为fun(int, int)）；

　　2、不能通过访问权限、返回类型、抛出的异常进行重载；

　　3、方法的异常类型和数目不会对重载造成影响；

　　4、对于继承来说，如果某一方法在父类中是访问权限是priavte，那么就不能在子类对其进行重载，如果定义的话，也只是定义了一个新方法，而不会达到重载的效果。

总结：

override（重写）

　　 1、方法名、参数、返回值相同。

　　 2、子类方法不能缩小父类方法的访问权限。

　　 3、子类方法不能抛出比父类方法更多的异常(但子类方法可以不抛出异常)。

　　 4、存在于父类和子类之间。

　　 5、方法被定义为final不能被重写。

　overload（重载）

　　1、参数类型、个数、顺序至少有一个不相同。

　　2、不能重载只有返回值不同的方法名。

　　3、存在于父类和子类、同类中。

## 抽象类

### 定义

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

#### 解释

比如一个**food**类，我们知道他是一个食物，但是不知道它的形状、大小、味道等等，所以它是抽象的，需要一个具体的饼干，面条来给它特定的描述。

#### 特点

* 1） 不能被实例化

这个我考虑很久，查阅了一些资料，个人理解有几点：

1. **空间分配，内存垃圾问题**。在实例化对象的时候(**new food()**)会开辟一个堆内存空间，而它没有实现的方法，是个东西，但没用。那就是垃圾。
2. **安全**。如果调用未实现的类就会报异常。比如

public abstract class Food{

public void print(){

System.out.println("un\_abstract method");

}

public abstract void abstractmethod();

}

假设我们**new**一个对象出来，**Food food = new Food()**,那么执行**food.abstractmethod**呢？  
3. **面向对象思想不允许**。可能也是最重要的吧。面向对象领域的一切都是对象，抽象在问题领域是没有概念的。在映射到现实社会的模拟，抽象是没有意义的。简单的来说，你实例化一个抽象的东西能干吗，没有实际的意义，违背了面向对象思想。

**关于这三点，属于个人观点，希望得到探讨和批评，不保证正确性**。

* 2） 抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

public abstract class Person{

private String name; // 变量

private String address;

public Employee(String name, String address){ // 构造方法

this.name = name;

this.address = address;

}

public abstract void cry(); // 抽象方法

public void jump(){ // 普通方法

System.out.println(this.name + "跳起来了")

}

// getter/setter 略

}

* 3）类中有抽象方法就必须定义为抽象类,抽象类中不一定包含抽象方法;

关于**抽象类中不一定包含抽象方法**，这个也有点钻牛角尖问题。如果一个抽象类不包含任何抽象方法，为何还要设计为抽象类？所以暂且记住这个概念吧，不必去深究为什么。

* 4）抽象类必须被继承使用。子类必须重写父类所有抽象方法或者声明自己为抽象类（实现部分抽象方法）;

public class Employee extends Person{

private int number;

public Employee(String name, String address, int number){

super(name,address);

this.number = number;

}

@Override

public void cry(){ //

System.out.println(this.number + "在叫")

}

// public abstract void \*();

// 子类也可以继续声明自己的抽象方法，继续被继承实现

}

* 5）abstract不能与final并列修饰同一个类。
* 6）abstract 不能与private、static、final或native并列修饰同一个方法。
* 7）一个类只能继承一个抽象类，而一个类却可以实现多个接口。

### 为什么需要抽象类？

抽象方法和抽象类看上去是多余的，对于抽象方法，不知道如何实现，定义一个空方法体不就行了吗，而抽象类不让创建对象，看上去只是增加了一个不必要的限制。

引入抽象方法和抽象类，是Java提供的一种语法工具，对于一些类和方法，引导使用者正确使用它们，减少被误用。

使用抽象方法，而非空方法体，子类就知道他必须要实现该方法，而不可能忽略。

使用抽象类，类的使用者创建对象的时候，就知道他必须要使用某个具体子类，而不可能误用不完整的父类。

无论是写程序，还是平时做任何别的事情的时候，每个人都可能会犯错，减少错误不能只依赖人的优秀素质，还需要一些机制，使得一个普通人都容易把事情做对，而难以把事情做错。抽象类就是Java提供的这样一种机制。

## 接口

在软件工程中，接口泛指供别人调用的方法或者函数。从这里，我们可以体会到 Java 语言设计者的初衷，它是对行为的抽象。在 Java 中，定一个接口的形式如下：

[public] interface InterfaceName {

}

#### 特征

* 1）接口中可以含有变量和方法

但是要注意，接口中变量只能是 public static final变量，用其他报错。

方法且只能是 public abstract 方法，用其他关键字，比如 private、protected、static、 final 等修饰会报编译错误

* 2）接口中的方法必须都是抽象方法
* 3）允许一个类遵循多个特定的接口

class ClassName implements Interface1,Interface2,[....]{

}

如果一个非抽象类遵循了某个接口，就必须实现该接口中的所有方法。对于遵循某个接口的抽象类，可以不实现该接口中的抽象方法。

## 抽象类和接口的区别

### 1. 语法层面上

* 抽象类可以提供成员方法的实现细节，而接口中只能存在public abstract 方法；
* 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是public static final类型的；
* 接口中不能含有静态代码块以及静态方法，而抽象类可以有静态代码块和静态方法；
* 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

说明抽象类只是类的另一种形式。而接口是一个公共的方法集合，不能有别的，一旦用了一个方法还得全部都得用，很纯粹，很牛XX

### 2. 设计层面上

要真正理解和区分，就要在设计层面上下功夫了，这样才能知道在什么场景下用它。

* 抽象类是对一种事物的抽象，即对类抽象，而接口是对行为的抽象。

举个栗子，飞机和鸟是不同的事物，但有一个共同的特征就是会飞。因此我们可以设计的时候，设计**Airplan**和**Bird**两个类，但**飞**是一个动作，一种行为，不是一类事物的抽象描述。所以就可以设计一个接口**Fly**,包含方法**fly()**，然后Airplane和Bird分别根据自己的需要实现Fly这个接口。

然后至于有不同种类的飞机，比如战斗机、民用飞机等直接继承Airplane即可，对于鸟也是类似的，不同种类的鸟直接继承Bird类即可。

从这里可以看出，类继承属于‘是不是’的问题，你属不属于飞机类，鸟类。接口实现是‘有没有’的问题，你有没有飞这个功能。

* 抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

我认为这就话就是精髓所在。直接再举起来一个栗子：门和报警

门都有**open()**和**close()**两个动作，但现在要给门加上**alarm()**，这就要分析一下了。

**Door** 的 **open()** 、**close()** 和 **alarm()** 属于两个不同范畴内的行为，open() 和 close() 属于门本身固有的行为特性，而 alarm() 属于延伸的附加行为。门也分很多种，玻璃门、防盗门、推拉门等等。不是所有的都具有报警功能。以后添加报警装置，也只是增加了一个报警功能。

因此最好的解决办法是单独将报警设计为一个接口，包含 **alarm()** 行为，**Door** 设计为单独的一个抽象类，包含 **open** 和 **close** 两种行为。再设计一个报警门继承 **Door** 类和实现 **Alarm** 接口。

即：

public abstract class Door{

public abstract void open();

public abstract void close();

}

interface Alarm{

void alarm();

}

class AlarmDoor extends Door implements Alarm{

@Override

void oepn() {

//....

}

@Override

void close() {

//....

}

@Override

void alarm() {

//....

}

}

参考资料：

[https://www.runoob.com/w3cnote/java-abstract-interface-different.html](https://www.runoob.com/w3cnote/java-abstract-interface-different.html" \t "_blank)

[https://www.runoob.com/java/java-abstraction.html](https://www.runoob.com/java/java-abstraction.html" \t "_blank)

[https://blog.csdn.net/chenssy/article/details/12858267](https://blog.csdn.net/chenssy/article/details/12858267" \t "_blank)

### sleep() 和 wait() 有什么区别？

* sleep() 是 Thread 类的静态本地方法；wait() 是Object类的成员本地方法
* sleep() 方法可以在任何地方使用；wait() 方法则只能在同步方法或同步代码块中使用，否则抛出异常Exception in thread "Thread-0" java.lang.IllegalMonitorStateException
* sleep() 会休眠当前线程指定时间，释放 CPU 资源，不释放对象锁，休眠时间到自动苏醒继续执行；wait() 方法放弃持有的对象锁，进入等待队列，当该对象被调用 notify() / notifyAll() 方法后才有机会竞争获取对象锁，进入运行状态
* JDK1.8 sleep() wait() 均需要捕获 InterruptedException 异常

测试代码

1. public class TestWaitSleep {
3. private static Object obj = new Object();
5. public static void main(String[] args) {
7. *//测试sleep()*
8. *//测试 RunnableImpl1 wait(); RunnableImpl2 notify()*
9. Thread t1 = new Thread(new RunnableImpl1(obj));
10. Thread t2 = new Thread(new RunnableImpl2(obj));
11. t1.start();
12. t2.start();
14. *//测试RunnableImpl3 wait(long timeout)方法*
15. Thread t3 = new Thread(new RunnableImpl3(obj));
16. t3.start();
17. }

20. }
22. class RunnableImpl1 implements Runnable {
24. private Object obj;
26. public RunnableImpl1(Object obj) {
27. this.obj = obj;
28. }
30. public void run() {
31. System.out.println("run on RunnableImpl1");
32. synchronized (obj) {
33. System.out.println("obj to wait on RunnableImpl1");
34. try {
35. obj.wait();
36. } catch (InterruptedException e) {
37. e.printStackTrace();
38. }
39. System.out.println("obj continue to run on RunnableImpl1");
40. }
41. }
42. }
44. class RunnableImpl2 implements Runnable {
46. private Object obj;
48. public RunnableImpl2(Object obj) {
49. this.obj = obj;
50. }
52. public void run() {
53. System.out.println("run on RunnableImpl2");
54. System.out.println("睡眠3秒...");
55. try {
56. Thread.sleep(3000);
57. } catch (InterruptedException e) {
58. e.printStackTrace();
59. }
60. synchronized (obj) {
61. System.out.println("notify obj on RunnableImpl2");
62. obj.notify();
63. }
64. }
65. }
67. class RunnableImpl3 implements Runnable {
69. private Object obj;
71. public RunnableImpl3(Object obj) {
72. this.obj = obj;
73. }
75. public void run() {
76. System.out.println("run on RunnableImpl3");
77. synchronized (obj) {
78. System.out.println("obj to wait on RunnableImpl3");
79. try {
80. obj.wait(1000);
81. } catch (InterruptedException e) {
82. e.printStackTrace();
83. }
84. System.out.println("obj continue to run on RunnableImpl3");
85. }
86. }
87. }

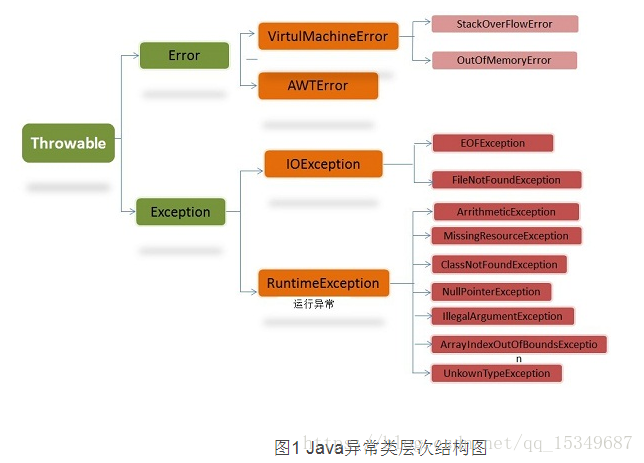
打印结果

1. run on RunnableImpl2
2. 睡眠3秒...
3. run on RunnableImpl1
4. obj to wait on RunnableImpl1
5. run on RunnableImpl3
6. obj to wait on RunnableImpl3
7. obj continue to run on RunnableImpl3
8. notify obj on RunnableImpl2
9. obj continue to run on RunnableImpl1

### ****一、什么是java异常？****

java异常指在程序运行时可能出现的一些错误，如：文件找不到、网络连接失败、非法参数等。异常是一个事件，它发生在程序运行期间，中断了正在执行的程序的正常指令流。Java通过API中Throwable类的众多子类描述各种不同的异常。因而，Java异常都是对象，是Throwable子类的实例，描述了出现在一段编码中的错误条件。当条件生成时，错误将引发异常。

Java异常类的层次结构图，如下所示（盗的图）：



Java所有异常类都是 Throwable的子类。它包括Java异常处理的两个重要子类：Error和Exception.

**Error**：Error及其子类用来描述Java运行系统中的内部错误以及资源耗尽的错误，是程序无法处理的错误，这类错误比较严重。这类的大多数错误与代码编写者执行的操作无关，如，运行代码时，JVM（Java虚拟机）出现的问题，例如，Java虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。 这些错误是不可查的，因为它们在应用程序的控制和处理能力之外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。

**Exception**：可以通过捕捉处理使程序继续执行，是程序自身可以处理的异常，也称为非致命性异常类。根据错误发生原因可分为RuntimeException异常和除RunTimeException之外的异常，如IOException异常。RuntimeException 类及其子类表示“JVM 常用操作”引发的错误。例如，若试图使用空值对象引用、除数为零或数组越界，则分别引发运行时异常（NullPointerException、ArithmeticException）和 ArrayIndexOutOfBoundException。

简单来说：异常与错误的区别是：异常可以通过程序自身捕捉处理，而错误是程序自身无法处理的。

Java异常包括Exception类和Error类，其可分为可查异常和不可查异常：

**可查异常**：编译器要求必须处理的异常，这类异常的发生在一定程度上是可以预计的，而且这类异常一旦发生，就必须采用某种方式进行处理。除了RuntimeException及其子类以外的其它异常类都属于可查异常。这种异常的特点是Java编译器会检查它，也就是说，出现这种异常，要么用try-catch语句捕捉它，要么用throws语句声明抛出它，否则编译不通过。

**不可查异常**：编译器不要求强制处理的异常，包括运行时异常（RuntimeException与其子类）和错误（Error） 。

**Exception 异常**又可分为两大类：运行时异常和非运行时异常(编译异常)。程序中应当尽可能去处理这些异常。

**运行时异常**：RuntimeException类及其子类异常，如NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是不可查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

运行时异常的特点：Java编译器不会检查它，也就是说，当程序中出现这类异常时，也会编译通过。

**非运行时异常 （编译异常）**：是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

### ****二、异常处理机制****

 在 Java 应用程序中，异常处理机制为：抛出异常，捕捉异常。

**抛出异常**：当一个方法出现错误引发异常时，方法创建异常对象并交付运行时系统，异常对象中包含了异常类型和异常出现时的程序状态等异常信息。运行时系统负责寻找处置异常的代码并执行。

**捕捉异常**：在方法抛出异常之后，运行时系统将转为寻找合适的异常处理器（exception handler）。潜在的异常处理器是异常发生时依次存留在调用栈中的方法的集合。当异常处理器所能处理的异常类型与方法抛出的异常类型相符时，即为合适的异常处理器。运行时系统从发生异常的方法开始，依次回查调用栈中的方法，直至找到含有合适异常处理器的方法并执行。当运行时系统遍历调用栈而未找到合适的异常处理器，则运行时系统终止。同时，意味着Java程序的终止。

***对于运行时异常、错误或可查异常，Java技术所要求的异常处理方式有所不同。***

由于运行时异常的不可查性，为了更合理、更容易地实现应用程序，Java规定，运行时异常将由Java运行时系统自动抛出，允许应用程序忽略运行时异常。

对于方法运行中可能出现的Error，当运行方法不欲捕捉时，Java允许该方法不做任何抛出声明。因为，大多数Error异常属于永远不能被允许发生的状况，也属于合理的应用程序不该捕捉的异常。

对于所有的可查异常，Java规定：一个方法必须捕捉，或者声明抛出方法之外。也就是说，当一个方法选择不捕捉可查异常时，它必须声明将抛出异常。

能够捕捉异常的方法，需要提供相符类型的异常处理器。所捕捉的异常，可能是由于自身语句所引发并抛出的异常，也可能是由某个调用的方法或者Java运行时 系统等抛出的异常。也就是说，一个方法所能捕捉的异常，一定是Java代码在某处所抛出的异常。简单地说，异常总是先被抛出，后被捕捉的。

任何Java代码都可以抛出异常，如：自己编写的代码、来自Java开发环境包中代码，或者Java运行时系统。无论是谁，都可以通过Java的throw语句抛出异常。从方法中抛出的任何异常都必须使用throws子句。捕捉异常通过try-catch语句或者try-catch-finally语句实现。

总体来说，Java规定：对于可查异常必须捕捉、或者声明抛出。允许忽略不可查的RuntimeException和Error。

***抛出异常的方法：throws和throw***

**throws**：通常被用在声明方法时，用来指定方法可能抛出的异常，多个异常可使用逗号分隔。throws关键字将异常抛给上一级，如果不想处理该异常，可以继续向上抛出，但最终要有能够处理该异常的代码。

**throw**：通常用在方法体中或者用来抛出用户自定义异常，并且抛出一个异常对象。程序在执行到throw语句时立即停止，如果要捕捉throw抛出的异常，则必须使用try-catch语句块或者try-catch-finally语句。

例1：throws方法抛出异常

1. public class Shoot {
2. static void pop()throws NegativeArraySizeException{
3. int[] arr = new int[-3];
4. }
5. public static void main(String[] args) {
6. try{
7. pop();
8. }catch(NegativeArraySizeException e){
9. System.out.println("pop()方法抛出的异常");
10. }
11. }
13. }

运行结果：



例2：throw方法抛出异常

1. public class TestException {
2. public static void main(String[] args) {
3. int a = 6;
4. int b = 0;
5. try {
7. if (b == 0) throw new ArithmeticException(); *// 通过throw语句抛出异常*
8. System.out.println("a/b的值是：" + a / b);
9. }
10. catch (ArithmeticException e) { *// catch捕捉异常*
11. System.out.println("程序出现异常，变量b不能为0。");
12. }
13. System.out.println("程序正常结束。");
14. }
15. }

 运行结果：

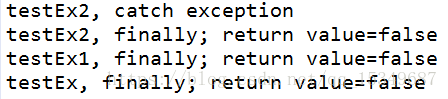


### ****三、理解try-catch-finally语句****

首先，我们来看一个例子（网上找的），大家想想运行之后的结果是什么？

1. package Exception;
3. public class exception4 {
4. public exception4() {
5. }
7. boolean testEx() throws Exception {
8. boolean ret = true;
9. try {
10. ret = testEx1();
11. } catch (Exception e) {
12. System.out.println("testEx, catch exception");
13. ret = false;
14. throw e;
15. } finally {
16. System.out.println("testEx, finally; return value=" + ret);
17. return ret;
18. }
19. }
21. boolean testEx1() throws Exception {
22. boolean ret = true;
23. try {
24. ret = testEx2();
25. if (!ret) {
26. return false;
27. }
28. System.out.println("testEx1, at the end of try");
29. return ret;
30. } catch (Exception e) {
31. System.out.println("testEx1, catch exception");
32. ret = false;
33. throw e;
34. } finally {
35. System.out.println("testEx1, finally; return value=" + ret);
36. return ret;
37. }
38. }
40. boolean testEx2() throws Exception {
41. boolean ret = true;
42. try {
43. int b = 12;
44. int c;
45. for (int i = 2; i >= -2; i--) {
46. c = b / i;
47. System.out.println("i=" + i);
48. }
49. return true;
50. } catch (Exception e) {
51. System.out.println("testEx2, catch exception");
52. ret = false;
53. throw e;
54. } finally {
55. System.out.println("testEx2, finally; return value=" + ret);
56. return ret;
57. }
58. }
60. public static void main(String[] args) {
61. exception4 testException1 = new exception4();
62. try {
63. testException1.testEx();
64. } catch (Exception e) {
65. e.printStackTrace();
66. }
67. }
68. }

运行结果：



哈哈！！！你的答案是否与正确答案一样呢？

大家可能刚看上去有这么多的try-catch-finally语句，感觉会比较懵！但实际上try-catch-finally语句理解很简单，只要牢记这些规则就好了：

对于try-catch-finally语句：先执行try 块中的代码，如果正常运行没有发生异常则执行完后执行finally 代码块中的代码；如若在try 中发生异常且被catch 捕捉到则执行catch 中的代码块，然后执行finally 块中的代码；

这样讲，大家都明白了吧，是不是很简单！！！

但也存在以下4种特殊情况，finally块不会被执行：

      在前面的代码中使用了System.exit()退出程序；

      在finally语句块中发生异常；

      程序所在的线程死亡；

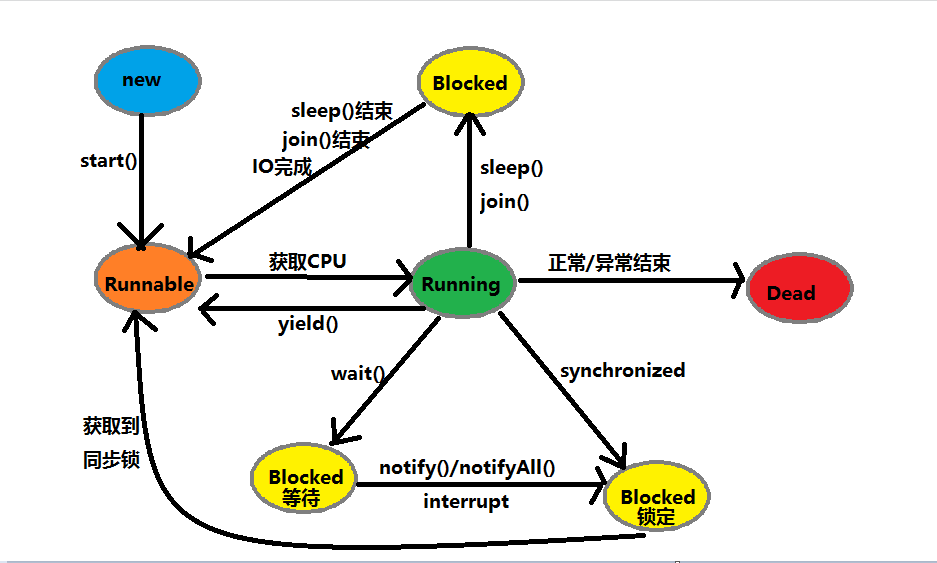
      关闭CPU。

想知道更多详细的内容，请参考如下链接地址：

https://blog.csdn.net/hguisu/article/details/6155636?utm\_source=copy

**线程的生命周期一共五个状态：新建、就绪、运行、堵塞、死亡**  
**（1）新建 new Thread**  
Thread thread=new Thread(）；  
当创建一个Thread对象时，这个线程就进入了新建状态，有了自己的内存空间，但还没有启动，此时线程还不是活的。  
**（2）就绪 runable**  
线程已经启动，正在等待CPU时间片。（通过线程的start()方法启动线程后进入就绪状态），此时具备运行条件，但由于没有CPU，所以处于线程就绪队列，此时线程是活着的。  
**（3）运行 running**  
线程执行(也就是进入了run()方法)，此时除非线程自动放弃CPU或者优先级更高的线程进入，否则会一直执行，此时线程是活着的。  
**（4）堵塞 blocked**  
由于某种原因导致正在运行的线程让出自己的CPU并暂停自己的执行，此时进入都塞状态。此时线程仍是活着的。  
**（5）死亡 dead**  
线程执行完毕或被其他线程杀死。此时线程是不活着的。  
*自然终止：run()执行完毕*  
*异常终止：调用了Stop()方法*

**线程常用的方法**  
**void run()**:线程必须覆盖的方法。  
**void start()**:开启一个线程。  
**static void sleep(long time)**:当前线程睡眠time毫秒。*sleep()方法释放了cpu的执行权，不释放锁。作用：保持对象锁，让出CPU，调用的目的就是让当前线程让出他获得CPU资源，以便给其他线程执行的机会。*  
**final void wait()**:进入等待池。*wait()方法释放cpu执行权，暂时失去对象的机锁，wait()结束后还要返还对象锁，****当前线程必须拥有当前对象的锁，所以wait()方法必须在synchronized block中调用****。*  
**final void notify()**:唤醒当前对象等待池中的第一个线程，notify()必须拥有相同的对象锁。  
**final void notifyAll()**:唤醒当前对象等待池中的第一个线程，notifyAll()必须拥有相同的对象锁。  
**static void yied()**:对当前对象临时暂停(但不保证会被虚拟机在次调度进入running状态)，让出CPU的执行权，给其他线程执行的机会，让同等优先级的线程运行，如果没有同等优先级的线程，则该方法不会起作用。

**线程的状态和方法总结下来就是：**  


# Java中字符串的常用方法

**1、equals()：比较两个字符串是否相等**  
它具有如下的一般形式：boolean equals(Object str)  
str是一个用来与调用字符串（String）对象做比较的字符串（String）对象。如果两个字符串具有相同的字符和长度，它返回true，否则返回false。这种比较是区分大小写的。  
**2、equalsIgnoreCase( )：忽略大小写的两个字符串是否相等比较**  
当比较两个字符串时，它会认为A-Z和a-z是一样的。  
其一般形式如下：boolean equalsIgnoreCase(String str)  
str是一个用来与调用字符串（String）对象做比较的字符串（String）对象。如果两个字符串具有相同的字符和长度，它也返回true，否则返回false。  
**3、toString()：转换成String类型**  
Object object = getObject();  
System.out.println(object.toString());  
注意：必须保证object不是null值，否则将抛出NullPointerException异常。  
采用这种方法时，通常派生类会覆盖Object里的toString（）方法。  
**4、String：转换成String类型**  
（String）object：这是标准的类型转换，将object转成String类型的值。  
注意：类型必须能转成String类型。因此最好用instanceof做个类型检查，以判断是否可以转换。否则容易抛出CalssCastException异常。  
因定义为Object 类型的对象在转成String时语法检查并不会报错，这将可能导致潜在的错误存在。  
如： Object obj = new Integer(100);  
String strVal = (String)obj;  
在运行时将会出错，因为将Integer类型强制转换为String类型，无法通过。  
但是， Integer obj = new Integer(100);  
String strVal = (String)obj;  
如是格式代码，将会报语法错误。此外，因null值可以强制转换为任何java类类型，(String)null也是合法的。  
**5、String.valueOf()：转换成String类型（不用担心object是否为null值这一问题）**  
注意：当object为null 时，String.valueOf（object）的值是字符串”null”，而不是null。  
**6、split()：分隔符**  
1、如果用“.”作为分隔的话,必须是如下写法,String.split("\\.")  
2、如果用“|”作为分隔的话,必须是如下写法,String.split("\\|")  
“.”、“|”、"\*" 和"+"都是转义字符,必须得加"\\";  
3、如果在一个字符串中有多个分隔符,可以用“|”作为连字符,比如,“acount=? and uu =? or n=?”,把三个都分隔出来,可以用String.split("and|or");  
例如：String[] aa = "aaa|bbb|ccc".split("\\\*");  
for (int i = 0 ; i <aa.length ; i++ ) {  
System.out.println("--"+aa[i]);  
}  
**7、subString()：截取字符串中的一段字符串**  
String str;  
（1）str＝str.substring(int beginIndex);  
截取掉str从首字母起长度为beginIndex的字符串，将剩余字符串赋值给str；  
（2）str＝str.substring(int beginIndex，int endIndex);  
截取str中从beginIndex开始至endIndex结束时的字符串，并将其赋值给str;  
**8、charAt()：返回指定索引处char值**  
public char charAt(int index)  
char s = str.charAt(1);  
**9、toLowerCase()：将所有在此字符串中的字符转化为小写（使用默认语言环境的规则）**  
public String toLowerCase()  
String newStr = str.toLowerCase();  
**10、indexOf()：指出 String 对象内子字符串的开始位置**  
1、int indexOf(String str) ：返回第一次出现的指定子字符串在此字符串中的索引。  
2、int indexOf(String str, int startIndex)：从指定的索引处开始，返回第一次出现的指定子字符串在此字符串中的索引。  
3、int lastIndexOf(String str) ：返回在此字符串中最右边出现的指定子字符串的索引。  
4、int lastIndexOf(String str, int startIndex) ：从指定的索引处开始向后搜索，返回在此字符串中最后一次出现的指定子字符串的索引。  
注意：如果没有找到子字符串，则返回-1。  
如果 startindex 是负数，则 startindex 被当作零。如果它比最大的字符位置索引还大，则它被当作最大的可能索引。  
例如：String s = "xXccxxxXX";  
// 从头开始查找是否存在指定的字符 //结果如下  
System.out.println(s.indexOf("c")); //2  
// 从第四个字符位置开始往后继续查找，包含当前位置  
System.out.println(s.indexOf("c", 3)); //3  
//若指定字符串中没有该字符则系统返回-1  
System.out.println(s.indexOf("y")); //-1  
System.out.println(s.lastIndexOf("x")); //6  
**11、replace和replaceAll**  
（1）replace的参数是char和CharSequence，即可以支持字符的替换，也支持字符串的替换（CharSequence即字符串序列的意思,说白了也就是字符串）；  
（2）replaceAll的参数是regex，即基于规则表达式的替换，比如：可以通过replaceAll("\\d", "\*")把一个字符串所有的数字字符都换成星号；  
相同点：都是全部替换，即把源字符串中的某一字符或字符串全部换成指定的字符或字符串；  
不同点：（1）replaceAll支持正则表达式，因此会对参数进行解析（两个参数均是），如replaceAll("\\d", "\*")，而replace则不会，replace("\\d","\*")就是替换"\\d"的字符串，而不会解析为正则。  
（2）“\”在java中是一个转义字符，所以需要用两个代表一个。例如System.out.println( "\\" ) ;只打印出一个"\"。但是“\”也是正则表达式中的转义字符，需要用两个代表一个。所以：\\\\被java转换成\\，\\又被正则表达式转换成\，因此用replaceAll替换“\”为"\\"，就要用replaceAll("\\\\","\\\\\\\\")，而replace则replace("\\","\\\\")。  
（3）如果只想替换第一次出现的，可以使用replaceFirst()，这个方法也是基于规则表达式的替换，但与replaceAll()不同的是，只替换第一次出现的字符串。  
说到正则表达式，有个例子就能很好的解释replaceAll的用法。即替换空格  
String test = "wa n\tg\_p\\te\\tn　g";  
test = test.replaceAll("\\t|\\\\t|\u0020|\\u3000", "");//去掉空格  
System.out.println(test);  
其中test = test.replaceAll("\\t|\\\\t|\u0020|\\u3000", "")  
与test = Pattern.compile("\\t|\\\\t|\u0020|\\u3000").matcher(test).replaceAll("")  
是等效的，  
因此用正则表达式仅仅是替换全部或替换第一个的话，用replaceAll或replaceFirst即可。  
**12、getBytes()：得到一个系统默认的编码格式的字节数组**  
都是将一个string类型的字符串转换成byte类型并且存入一个byte数组中。在java中的所有数据底层都是字节，字节数据可以存入到byte数组。  
UTF-8每个汉字转成3bytes，而GBK转成2bytes，所以要说明编码方式，否则用缺省编码。  
String.getBytes(String decode)  
byte[] b\_gbk = "中".getBytes("GBK");  
byte[] b\_utf8 = "中".getBytes("UTF-8");  
byte[] b\_iso88591 = "中".getBytes("ISO8859-1");  
将分别返回"中"这个汉字在GBK、UTF-8和ISO8859-1编码下的byte数组表示,此时  
b\_gbk的长度为2,  
b\_utf8的长度为3,  
b\_iso88591的长度为1。  
new String(byte[], decode)实际是使用指定的编码decode来将byte[]解析成字符串.  
String s\_gbk = new String(b\_gbk,"GBK");  
String s\_utf8 = new String(b\_utf8,"UTF-8");  
String s\_iso88591 = new String(b\_iso88591,"ISO8859-1");  
通过输出s\_gbk、s\_utf8和s\_iso88591,会发现s\_gbk和s\_utf8都是"中",而只有s\_iso88591是一个不被识别的字符（可以理解为乱码）,为什么使用ISO8859-1编码再组合之后,无法还原"中"字？原因很简单,因为ISO8859-1编码的编码表根本就不包含汉字字符,当然也就无法通过"中".getBytes("ISO8859-1");来得到正确的"中"字在ISO8859-1中的编码值了,所以，再通过new String()来还原就更是无从谈起。因此,通过String.getBytes(String decode)方法来得到byte[]时,一定要确定decode的编码表中确实存在String表示的码值,这样得到的byte[]数组才能正确被还原。  
**13、StringBuffer的append方法**  
StringBuffer buf=new StringBuffer("Hard ");  
String aString = "Waxworks";  
buf.append(aString,3,4);  
原文说明：这个操作将aString的从索引位置3开始的由四个字符组成的子串追加到StringBuffer对象buf中。然后buf对象就会包含字符 串"Hard work"。  
请注意，这个代码的实际运行结果是： buf对象包含的字符串为"Hard w"。  
具体原因引用源代码：  
public synchronized StringBuffer append(CharSequence s, int start, int end)  
{  
super.append(s, start, end);  
return this;  
}  
根据运行结果分析，StringBuffer对象的append（）方法的参数，如果是String类型，那么，后面取子串的操作实际是从索引3开始，取值到索引4之前的串。如果append的语句改成 buf.append(aString,3,3); ,那么没有添加aString的子串，即 buf包含的字符实际还是"Hard "。如果此语句再改成 buf.append(aString3,2); ，那么系统会抛出"IndexOutOfBoundsException"的异常！  
但是，如果append()的参数是字符数组（char[])，那么结果就如原文所述，buf将包含串"Hard work". 代码如下：  
StringBuffer buf=new StringBuffer("Hard ");  
char[] text ={'W','a','x','w','o','r','k','s'};  
buf.append(text ,3,4); // buf包含串"Hard work"  
具体原因引用源代码：  
public synchronized StringBuffer append(char str[], int offset, int len)  
{  
super.append(str, offset, len);  
return this;  
}  
JAVA 中 Stringbuffer 有append()方法  
　　Stringbuffer其实是动态字符串数组  
　　append()是往动态字符串数组添加，跟“xxxx”+“yyyy”相当那个‘+’号  
　　跟String不同的是Stringbuffer是放一起的  
　　String1+String2 和Stringbuffer1.append("yyyy")虽然打印效果一样，但在内存中表示却不一样  
　　String1+String2 存在于不同的两个地址内存  
　　Stringbuffer1.append(Stringbuffer2)放再一起

# Java集合框架

## <Java集合框架的四大接口>

Collection：存储无序的、不唯一的数据；其下有List和Set两大接口。

List：存储有序的、不唯一的数据；

Set：存储无序的、唯一的数据；

Map：以键值对的形式存储数据，以键取值。键不能重复，但值可以重复。

接口的常用实现类：ArrayList、LinkedList、Vector、HashSet、LinkedHashSet、TreeSet、HashMap、LinkedHashMap、TreeMap、HashTable

## 一、List接口

是一个有序集合，继承自Collection接口。现已知常用实现类有：ArrayList、LinkedList、Vector。

### 1. 常用方法：

add（）：在列表的最后添加元素；

add（index，obj）：在列表的指定位置添加元素；

size（）：返回当前列表的元素个数

get（int index）：返回下标为index的元素；

clear（）：清除列表中所有元素；

isEmpty（）：检测列表是否为空；返回true/false；

contains（）：传入一个对象，检测列表中是否含有该对象；

indexOf（）：传入一个对象，返回该对象在列表中首次出现的地址；

lastInsevOf（）：传入一个对象，返回该对象在列表中最后一次出现的地址；

remove（）：传入一个下标，或者一个对象，删除指定元素；如果删除指定对象，需要重写equals（）方法，比对传入的对象是不是属于原列表，如果属于，结果返回true；

set（index，obj）：用新传入的对象，将指定位置的元素替换掉，返回被替换前的元素对象；

subList（）：截取一个子列表返回List类型；

toArray（）：将列表转为数组，返回一个Object[]类型；

iterator（）：使用迭代器遍历。

### 2. 如何输出List列表？

a. for循环；

b. foreach遍历（比较常用）；

c. 使用迭代器遍历列表；

### 3. List接口的常用实现类：

ArrayList和LinkedList实现类是继承自List接口的，所以List的常用方法它们也是适用的。

#### a. ArrayList

ArrayList实现了一个长度可变的数组，在内存空间中开辟一串连续的空间。

#### b. LinkedList

LinkedList使用链表结构存储数据，在插入和删除元素是速度非常快。

[复制代码](javascript:void(0);)

1 //ArrayList实现类

2 ArrayList<String> list1=new ArrayList<String>();

3 list1.add("第一条数据");

4 list1.add("第二条数据");

5 list1.add("第三条数据");

6 list1.add("第四条数据");

7 list1.add("第五条数据");

8 list1.add("第三条数据");

9 list1.add(2,"第六条数据"); //在第三条数据后添加“第六条数据”

10

11 System.out.println(list1.size()); //返回列表长度

12 System.out.println(list1.get(2)); //返回“第六条数据”

13 //list1.clear(); //清除所有数据

14 System.out.println(list1.isEmpty()); //返回false

15 System.out.println(list1.contains("第一条数据")); //返回true

16 System.out.println(list1.indexOf("第三条数据")); //返回3

17 System.out.println(list1.lastIndexOf("第三条数据")); //返回6

18 System.out.println(list1.remove(1)); //删除“第二条数据”

19 System.out.println(list1.set(0, "替换第一条数据")); //替换第一条数据

20 List list=list1.subList(2, 5); //截取第三到第五条数据，返回List

21 System.out.println(list1.toString()); //转成数组

22

23 //LinkedList实现类

24 LinkedList<News> list2=new LinkedList<News>();

25 list2.add(new News(1,"xxxxxxx","赵"));

26 list2.add(new News(2,"sssssss","钱"));

27 list2.add(new News(3,"yyyyyyy","孙"));

28 list2.add(new News(4,"nnnnnnn","李"));

29 list2.add(new News(5,"rrrrrrr","周"));

30 System.out.println(list2.contains(new News(3,"yyyyyyy","孙"))); //返回为true；重写equals()方法；如果不重写equals()方法，返回为false；

31 System.out.println(list2.remove(new News(1,"xxxxxxx","赵"))); //返回为true

32

33 /\*

34 \* 使用for循环遍历

35 \*/

36 for (int i = 0; i < list1.size(); i++) {

37 if(list1.get(i) instanceof String){ //判断传入的数据属不属于String类型

38 String str=list1.get(i);

39 System.out.println(str);

40 }

41 }

42 /\*

43 \* foreach遍历

44 \*/

45 for (News news : list2) {

46 System.out.println(news.getId()+" "+news.getTitle()+"\t"+news.getAuthor());

47 }

48 /\*

49 \* 迭代器遍历

50 \*/

51 Iterator<String> iter=list.iterator();

52 while(iter.hasNext()){ //判断list有没有下一条

53 String s=iter.next(); //字符串取到下一条

54 System.out.println(s);

55 }

56

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 /\*\*

2 \*News类

3 \*/

4 class News{

5 private int id;

6 private String title;

7 private String author;

8

9 @Override

10 public boolean equals(Object obj) {

11 if (this == obj)

12 return true;

13 if (obj == null)

14 return false;

15 if (getClass() != obj.getClass())

16 return false;

17 News other = (News) obj;

18 if (author == null) {

19 if (other.author != null)

20 return false;

21 } else if (!author.equals(other.author))

22 return false;

23 if (id != other.id)

24 return false;

25 if (title == null) {

26 if (other.title != null)

27 return false;

28 } else if (!title.equals(other.title))

29 return false;

30 return true;

31 }

32 public News() {

33 super();

34 }

35 public News(int id, String title, String author) {

36 super();

37 this.id = id;

38 this.title = title;

39 this.author = author;

40 }

41 public int getId() {

42 return id;

43 }

44 public void setId(int id) {

45 this.id = id;

46 }

47 public String getTitle() {

48 return title;

49 }

50 public void setTitle(String title) {

51 this.title = title;

52 }

53 public String getAuthor() {

54 return author;

55 }

56 public void setAuthor(String author) {

57 this.author = author;

58 }

59

60

61 }

[复制代码](javascript:void(0);)

## 二、Set接口

### 1. 常用方法：

与List接口基本相同。但是Set接口中的元素是无序的，因此没有与下标有关的方法。例如：get（index）、remove（index）、add（index，obj）

### 2. Set接口的现已知常用实现类有：

HashSet、LinkedHashSet、TreeSet

### 3. HashSet

底层是HashMap的相关方法，传入数据后，根据数据的hashCode进行散列运算，得到一个散列值后再进行运算，确定元素在序列中存储的位置。

所以，使用HashSet存数据必须在实体类中重写hashCode和equals方法！！

[复制代码](javascript:void(0);)

1 //HashSet无序的

2 Set<String> set1=new HashSet<String>();

3 set1.add("set1");

4 set1.add("set2");

5 set1.add("set3");

6 set1.add("set4");

7 set1.add("set5");

8

9 //foreach遍历

10 for (String set : set1) {

11 System.out.println(set);

12 }

13

[复制代码](javascript:void(0);)

### 4. LinkedHashSet

在HashSet的基础上，新增了一个链表。用链表来记录HashSet种元素放入的顺序；HashSet依然是无序的，但链表会按照存入的顺序存储。

[复制代码](javascript:void(0);)

1 //LinkedHashSet按照放入的顺序输出

2 Set<Person> set2=new LinkedHashSet<Person>();

3 // set2.addAll(set1); //追加set1的所有数据

4 set2.add(new Person(1,"红"));

5 set2.add(new Person(2,"黄"));

6 set2.add(new Person(3,"蓝"));

7 set2.add(new Person(4,"绿"));

8 set2.add(new Person(4,"绿"));

9

10 //迭代器遍历

11 Iterator<Person> iter=set2.iterator();

12 while(iter.hasNext()){

13 Person p=iter.next();

14 System.out.println(p.getId()+"\t"+p.getName());

15 }

16

17 class Person{

18 private int id;

19 private String name;

20

21 /\*

22 \* 如果不重写hashCode和equals方法，存入相同数据时将无法比对

23 \*

24 \*/

25 @Override

26 public int hashCode() {

27 final int prime = 31;

28 int result = 1;

29 result = prime \* result + id;

30 result = prime \* result + ((name == null) ? 0 : name.hashCode());

31 return result;

32 }

33 @Override

34 public boolean equals(Object obj) {

35 if (this == obj)

36 return true;

37 if (obj == null)

38 return false;

39 if (getClass() != obj.getClass())

40 return false;

41 Person other = (Person) obj;

42 if (id != other.id)

43 return false;

44 if (name == null) {

45 if (other.name != null)

46 return false;

47 } else if (!name.equals(other.name))

48 return false;

49 return true;

50 }

51 public Person() {

52 super();

53 }

54 public Person(int id, String name) {

55 super();

56 this.id = id;

57 this.name = name;

58 }

59 public int getId() {

60 return id;

61 }

62 public void setId(int id) {

63 this.id = id;

64 }

65 public String getName() {

66 return name;

67 }

68 public void setName(String name) {

69 this.name = name;

70 }

71

72 }

[复制代码](javascript:void(0);)

### 5. TreeSet

将存入的数据，进行排序，然后输出。

如果传入的是一个实体对象，那么需要传入比较器：实体类实现Comparable接口，并重写CompareTo方法；

[复制代码](javascript:void(0);)

1 //TreeSet从小到大输出。存入元素，默认升序排序；存入实体对象，需要传入比较器

2 Set<Person> set3=new TreeSet<Person>();

3 set3.add(new Person(11,"a"));

4 set3.add(new Person(22,"b"));

5 set3.add(new Person(33,"c"));

6 set3.add(new Person(44,"d"));

7

8 //迭代器遍历

9 Iterator<Person> iter=set3.iterator();

10 while(iter.hasNext()){

11 Person p=iter.next();

12 System.out.println(p.getId()+"\t"+p.getName());

13 }

14

15 class Person implements Comparable{ //实现

16 private int id;

17 private String name;

18

19 //compareTo方法判断传入的对象和已有对象

20 @Override

21 public int compareTo(Object o) {

22 Person p=null;

23 if(o instanceof Person){

24 p=(Person)o;

25 }else{

26 return 0;

27 }

28 return this.id -p.getId(); //为正数，升序；为负数，降序；等于0，位置不变

29 }

30

31 public Person() {

32 super();

33 }

34 public Person(int id, String name) {

35 super();

36 this.id = id;

37 this.name = name;

38 }

39 public int getId() {

40 return id;

41 }

42 public void setId(int id) {

43 this.id = id;

44 }

45 public String getName() {

46 return name;

47 }

48 public void setName(String name) {

49 this.name = name;

50 }

51

52

53 }

[复制代码](javascript:void(0);)

## 三、Map接口

### 1. Map接口的特点：

以键值对的形式存储数据，以键取值。键不能重复，值可以重复。

### 2. Map接口的现已知常用实现类有：

HashMap、HashTable、LinkedHashMap、TreeMap

### 3. 常用方法：

put（key，value）：向Map的最后追加一个键值对；

get（key）：通过键，取到一个值；

clear（）：清除Map中的所有数据；

containsKey（key）：检测是否包含指定的键；

containsValue（obj）：检测是否包含指定的值；

replace（）：替换指定键的值；

### 4. 遍历Map

a. keySet（）：返回Set，先取键，再取值；

b. values（）：返回Collection，直接取值；

c. 取代一个entry键值对，返回Set<Entry<>>；

[复制代码](javascript:void(0);)

1 Map<String, String> map1=new HashMap<String, String>();

2 map1.put("01", "aaaaaa");

3 map1.put("02", "bbbbbb");

4 map1.put("03", "cccccc");

5 map1.put("04", "dddddd");

6 // map1.clear(); //清除所有数据

7 System.out.println(map1.containsKey("01")); //返回true

8 System.out.println(map1.containsValue("aaaa")); //返回false

9 System.out.println(map1.replace("03", "dddddd")); //将cccccc替换为dddddd

10 System.out.println(map1.get("02")); //返回bbbbbb

11

12 /\*\*

13 \* 遍历Map的方式一，先取键，再取值

14 \*/

15 Set<String> keys=map1.keySet(); //取到所有键

16 Iterator<String> iter=keys.iterator();

17 while(iter.hasNext()){ //迭代器遍历取到每一个键

18 String key=iter.next();

19 System.out.println(key+" "+map1.get(key)); //key:取到键 map1.get(key):取到每个键对应的值

20 }

21

22 /\*\*

23 \* 遍历Map的方式二，直接取值；只能取到值

24 \*/

25 Collection<String> values = map1.values();

26 Iterator<String> iter1= values.iterator();

27 while(iter1.hasNext()){

28 System.out.println(iter1.next());

29 }

30

31 /\*\*

32 \* 遍历Map的方式三,取到一个entry键值对

33 \*/

34 Set<Entry<String,String>> set= map1.entrySet();

35 Iterator<Entry<String,String>> iter2=set.iterator();

36 while(iter2.hasNext()){

37 Entry<String,String> entry=iter2.next();

38 System.out.println(entry.getKey()+" "+entry.getValue());

39 }

40

[复制代码](javascript:void(0);)

### 5. LinkedHashMap

可以使用列表，记录数据放入的次序，输出的顺序与放入的顺序一致。与LinkEDHashSet一样。

### 6. TreeMap

根据键的顺序，进行排序后输出。与TreeSet。

如果传入的是实体对象，必须重写比较函数。

### 7. HashMap和HashTable的区别？

a. HashTable是线程安全的，HashMap是线程不安全的；

b. HashTable的键不能为null，HashMap的键可以为null；

c. HashTable继承自Dirctionary字典类，HashMap继承自abstract类；

d. 扩容大小：HashTable两倍，HashMap两倍加一；

e. 初始容量：HashTable为11，HashMap为16；两者的填充因子默认都是0.75；

# Java时间日期的处理：Java Date类、Calendar类详解

在 [Java](http://c.biancheng.net/java/" \t "_blank) 中获取当前时间，可以使用 java.util.Date 类和 java.util.Calendar 类完成。其中，Date 类主要封装了系统的日期和时间的信息，Calendar 类则会根据系统的日历来解释 Date 对象。下面详细介绍这两个类的具体使用。

## Date 类

Date 类表示系统特定的时间戳，可以精确到毫秒。Date 对象表示时间的默认顺序是星期、月、日、小时、分、秒、年。

#### 1. 构造方法

Date 类有如下两个构造方法。

* Date()：此种形式表示分配 Date 对象并初始化此对象，以表示分配它的时间（精确到毫秒），使用该构造方法创建的对象可以获取本地的当前时间。
* Date(long date)：此种形式表示从 GMT 时间（格林尼治时间）1970 年 1 月 1 日 0 时 0 分 0 秒开始经过参数 date 指定的毫秒数。

这两个构造方法的使用示例如下：

1. Date date1 = **new** Date(); // 调用无参数构造函数
2. System.out.println(date1.toString()); // 输出：Wed May 18 21:24:40 CST 2016
3. Date date2 = **new** Date(60000); // 调用含有一个long类型参数的构造函数
4. System.out.println(date2); // 输出：Thu Jan 0108:01:00 CST 1970

Date 类的无参数构造方法获取的是系统当前的时间，显示的顺序为星期、月、日、小时、分、秒、年。  
  
Date 类带 long 类型参数的构造方法获取的是距离 GMT 指定毫秒数的时间，60000 毫秒是一分钟，而 GMT（格林尼治标准时间）与 CST（中央标准时间）相差 8 小时，也就是说 1970 年 1 月 1 日 00:00:00 GMT 与 1970 年 1 月 1 日 08:00:00 CST 表示的是同一时间。 因此距离 1970 年 1 月 1 日 00:00:00 CST 一分钟的时间为 1970 年 1 月 1 日 00:01:00 CST，即使用 Date 对象表示为 Thu Jan 01 08:01:00 CST 1970。

#### 2. 常用方法

Date 类提供了许多与日期和事件相关的方法，其中常见的方法如表 1 所示。

|  |  |
| --- | --- |
| 表1 Date类中的常用方法 | |
| **方法** | **描述** |
| boolean after(Date when) | 判断此日期是否在指定日期之后 |
| boolean before(Date when) | 判断此日期是否在指定日期之前 |
| int compareTo(Date anotherDate) | 比较两个日期的顺序 |
| boolean equals(Object obj) | 比较两个日期的相等性 |
| long getTime() | 返回自 1970 年 1 月 1 日 00:00:00 GMT 以来，此 Date 对象表示的毫秒数 |
| String toString() | 把此 Date 对象转换为以下形式的 String: dow mon dd hh:mm:ss zzz yyyy。 其中 dow 是一周中的某一天(Sun、Mon、Tue、Wed、Thu、Fri 及 Sat) |

#### 例 1

下面使用一个实例来具体演示 Date 类的使用。假设，某一天特定时间要去做一件事，而且那个时间已经过去一分钟之后才想起来这件事还没有办，这时系统将会提示已经过去了多 长时间。具体的代码如下：

1. import java.util.Date;
2. import java.util.Scanner;
3. **public** **class** Test11 {
4. **public** **static** void main(String[] args) {
5. Scanner input = **new** Scanner(System.in);
6. System.out.println("请输入要做的事情：");
7. String title = input.next();
8. Date date1 = **new** Date(); // 获取当前日期
9. System.out.println("[" + title + "] 这件事发生时间为：" + date1);
10. **try** {
11. Thread.sleep(60000);// 暂停 1 分钟
12. } **catch** (InterruptedException e) {
13. e.printStackTrace();
14. }
15. Date date2 = **new** Date();
16. System.out.println("现在时间为：" + date2);
17. **if** (date2.before(date1)) {
18. System.out.println("你还有 " + (date2.getTime() - date1.getTime()) / 1000 + " 秒需要去完成【" + title + "】这件事！");
19. } **else** {
20. System.out.println("【" + title + "】事情已经过去了 " + (date2.getTime() - date1.getTime()) / 1000 + " 秒");
21. }
22. }
23. }

在该程序中，分别使用 Date 类的无参数构造方法创建了两个 Date 对象。在创建完第一个 Date 对象后，使用 Thread.sleep() 方法让程序休眠 60 秒，然后再创建第二个 Date 对象，这样第二个 Date 对象所表示的时间将会在第一个 Date 对象所表示时间之后，因此“date2.before(date1)”条件表达式不成立，从而执行 else 块中的代码，表示事情已经发生过。  
  
运行该程序，执行结果如下所示。

请输入要做的事情：

收快递

【收快递】这件事发生时间为：Fri Oct 12 11:11:07 CST 2018

现在时间为：Fri Oct 12 11:12:07 CST 2018

【收快递】事情已经过去了 60 秒

## Calendar 类

Calendar 类是一个抽象类，它为特定瞬间与 YEAR、MONTH、DAY\_OF—MONTH、HOUR 等日历字段之间的转换提供了一些方法，并为操作日历字段（如获得下星期的日期） 提供了一些方法。  
  
创建 Calendar 对象不能使用 new 关键字，因为 Calendar 类是一个抽象类，但是它提供了一个 getInstance() 方法来获得 Calendar类的对象。getInstance() 方法返回一个 Calendar 对象，其日历字段已由当前日期和时间初始化。

1. Calendar c = Calendar.getInstance();

当创建了一个 Calendar 对象后，就可以通过 Calendar 对象中的一些方法来处理日期、时间。Calendar 类的常用方法如表 2 所示。

|  |  |
| --- | --- |
| 表 2 Calendar类的常用方法 | |
| **方法** | **描述** |
| void add(int field, int amount) | 根据日历的规则，为给定的日历字段 field 添加或减去指定的时间量 amount |
| boolean after(Object when) | 判断此 Calendar 表示的时间是否在指定时间 when 之后，并返回判断结果 |
| boolean before(Object when) | 判断此 Calendar 表示的时间是否在指定时间 when 之前，并返回判断结果 |
| void clear() | 清空 Calendar 中的日期时间值 |
| int compareTo(Calendar anotherCalendar) | 比较两个 Calendar 对象表示的时间值（从格林威治时间 1970 年 01 月 01 日 00 时 00 分 00 秒至现在的毫秒偏移量），大则返回 1，小则返回 -1，相等返回 0 |
| int get(int field) | 返回指定日历字段的值 |
| int getActualMaximum(int field) | 返回指定日历字段可能拥有的最大值 |
| int getActualMinimum(int field) | 返回指定日历字段可能拥有的最小值 |
| int getFirstDayOfWeek() | 获取一星期的第一天。根据不同的国家地区，返回不同的值 |
| static Calendar getInstance() | 使用默认时区和语言坏境获得一个日历 |
| static Calendar getInstance(TimeZone zone) | 使用指定时区和默认语言环境获得一个日历 |
| static Calendar getInstance(TimeZone zone, Locale aLocale) | 使用指定时区和语言环境获得一个日历 |
| Date getTime() | 返回一个表示此 Calendar 时间值（从格林威治时间 1970 年 01 月 01 日 00 时 00 分 00 秒至现在的毫秒偏移量）的 Date 对象 |
| long getTimeInMillis() | 返回此 Calendar 的时间值，以毫秒为单位 |
| void set(int field, int value) | 为指定的日历字段设置给定值 |
| void set(int year, int month, int date) | 设置日历字段 YEAR、MONTH 和 DAY\_OF\_MONTH 的值 |
| void set(int year, int month, int date, int hourOfDay, int minute, int second) | 设置字段 YEAR、MONTH、DAY\_OF\_MONTH、HOUR、 MINUTE 和 SECOND 的值 |
| void setFirstDayOfWeek(int value) | 设置一星期的第一天是哪一天 |
| void setTimeInMillis(long millis) | 用给定的 long 值设置此 Calendar 的当前时间值 |

Calendar 对象可以调用 set() 方法将日历翻到任何一个时间，当参数 year 取负数时表示公元前。Calendar 对象调用 get() 方法可以获取有关年、月、日等时间信息，参数 field 的有效值由 Calendar 静态常量指定。  
  
Calendar 类中定义了许多常量，分别表示不同的意义。

* Calendar.YEAR：年份。
* Calendar.MONTH：月份。
* Calendar.DATE：日期。
* Calendar.DAY\_OF\_MONTH：日期，和上面的字段意义完全相同。
* Calendar.HOUR：12小时制的小时。
* Calendar.HOUR\_OF\_DAY：24 小时制的小时。
* Calendar.MINUTE：分钟。
* Calendar.SECOND：秒。
* Calendar.DAY\_OF\_WEEK：星期几。

例如，要获取当前月份可用如下代码：

1. int month = Calendar.getInstance().get(Calendar.MONTH);

如果整型变量 month 的值是 0，表示当前日历是在 1 月份；如果值是 11，则表示当前日历在 12 月份。  
  
使用 Calendar 类处理日期时间的实例如下：

1. Calendar calendar = Calendar.getInstance(); // 如果不设置时间，则默认为当前时间
2. calendar.setTime(**new** Date()); // 将系统当前时间赋值给 Calendar 对象
3. System.out.println("现在时刻：" + calendar.getTime()); // 获取当前时间
4. int year = calendar.get(Calendar.YEAR); // 获取当前年份
5. System.out.println("现在是" + year + "年");
6. int month = calendar.get(Calendar.MONTH) + 1; // 获取当前月份（月份从 0 开始，所以加 1）
7. System.out.print(month + "月");
8. int day = calendar.get(Calendar.DATE); // 获取日
9. System.out.print(day + "日");
10. int week = calendar.get(Calendar.DAY\_OF\_WEEK) - 1; // 获取今天星期几（以星期日为第一天）
11. System.out.print("星期" + week);
12. int hour = calendar.get(Calendar.HOUR\_OF\_DAY); // 获取当前小时数（24 小时制）
13. System.out.print(hour + "时");
14. int minute = calendar.get(Calendar.MINUTE); // 获取当前分钟
15. System.out.print(minute + "分");
16. int second = calendar.get(Calendar.SECOND); // 获取当前秒数
17. System.out.print(second + "秒");
18. int millisecond = calendar.get(Calendar.MILLISECOND); // 获取毫秒数
19. System.out.print(millisecond + "毫秒");
20. int dayOfMonth = calendar.get(Calendar.DAY\_OF\_MONTH); // 获取今天是本月第几天
21. System.out.println("今天是本月的第 " + dayOfMonth + " 天");
22. int dayOfWeekInMonth = calendar.get(Calendar.DAY\_OF\_WEEK\_IN\_MONTH); // 获取今天是本月第几周
23. System.out.println("今天是本月第 " + dayOfWeekInMonth + " 周");
24. int many = calendar.get(Calendar.DAY\_OF\_YEAR); // 获取今天是今年第几天
25. System.out.println("今天是今年第 " + many + " 天");
26. Calendar c = Calendar.getInstance();
27. c.set(2012, 8, 8); // 设置年月日，时分秒将默认采用当前值
28. System.out.println("设置日期为 2012-8-8 后的时间：" + c.getTime()); // 输出时间

上面的示例代码演示了 Calendar 类中的方法与常量的结合使用，从而完成处理日期的操作。

#### 例 2

下面使用 Calendar 类来实现日历的打印功能，代码实现如下：

1. import java.util.Calendar;
2. **public** **class** CalendarDemo {
3. **public** **static** void main(String[] args) {
4. Calendar calendar = Calendar.getInstance();
5. calendar.set(2016, 5, 1); // 实际的calendar对象所表示的日期为2016年6月1日
6. // 判断2016年6月1日为一周中的第几天
7. int index = calendar.get(Calendar.DAY\_OF\_WEEK) - 1;
8. char[] title = { '日', '一', '二', '三', '四', '五', '六' }; // 存放曰历的头部
9. int daysArray[][] = **new** int[6][7];// 存放日历的数据
10. int daysInMonth = 31; // 该月的天数
11. int day = 1; // 自动增长
12. **for** (int i = index; i < 7; i++) {
13. // 填充第一周的日期数据，即日历中的第一行
14. daysArray[0][i] = day++;
15. }
16. **for** (int i = 1; i < 6; i++) {
17. // 填充其他周的日历数据，控制行
18. **for** (int j = 0; j < 7; j++) {
19. // 如果当前day表示的是本月最后一天，则停止向数组中继续赋值
20. **if** (day > daysInMonth) {
21. i = 6;
22. **break**;
23. }
24. daysArray[i][j] = day++;
25. }
26. }
27. System.out.println("------------------2016 年 6 月--------------------\n");
28. **for** (int i = 0; i < title.length; i++) {
29. System.out.print(title[i] + "\t");
30. }
31. System.out.print("\n");
32. // 输出二元数组daysArray中的元素
33. **for** (int i = 0; i < 6; i++) {
34. **for** (int j = 0; j < 7; j++) {
35. **if** (daysArray[i][j] == 0) {
36. **if** (i != 0) {
37. // 如果到月末，则完成显示日历的任务，停止该方法的执行
38. **return**;
39. }
40. System.out.print("\t");
41. **continue**;
42. }
43. System.out.print(daysArray[i][j] + "\t");
44. }
45. System.out.print("\n");
46. }
47. }
48. }

该程序看似复杂其实很简单。因为 Calendar 类所表示的时间月份是 set() 方法中表示月份的参数值 +1，因此 Calendar 类的实际时间为 2016 年 6 月 1 日。在下面的代码中分别获取 6 月 1 日为本周中的第几天，以便在相应的星期下开始输出 6 月份的日历。程序中的 daysArray 是一个二元数组，该二元数组控制了日历的格式输出，第一个子数组控制日历的行，第二个子数组控制曰历的列，即可输出二元数组中的每一个元素。  
  
运行程序，执行结果如下所示。

------------------2016 年 6 月--------------------

日 一 二 三 四 五 六

1 2 3 4

5 6 7 8 9 10 11

12 13 14 15 16 17 18

19 20 21 22 23 24 25

26 27 28 29 30 31