# Collections 和 Arrays

java.util.Collection是集合的顶层接口，定义了对集合对象操作的通用方法，这些方法包括：

size()

contains(Object)

iterator()

toArry()

toArray(T[])

add(E)

remove(Object)

containsAll(Collection)

addAll(Collection)

removeAll(Collection)

clear()

retainAll(Collection)（保留）

**java 8 新增的：**

spliterator()（并行遍历元素）

stream()

removeIf(Predicate)

parallelStream()（并行流）

forEach(Consumer)（默认方法）

## toArray()和toArray(T[])方法

如下代码会报错：ClassCastException

ArrayList<Integer> list = new ArrayList<Integer>();  
for (int i = 0; i < 10; i++) {  
 list.add(i);  
}  
Integer[] array = (Integer[]) list.toArray();

Java中的类型转换是针对单个对象的，要将一个数组（**数组为引用类型**）对象强转为另一种数组类型，就必须确保这个数组在初始化时就是目标类型。

查看数组类型：

(new List[]{new ArrayList(), new LinkedList(), new ArrayList()}).getClass().getName()

类型名为：java.util.List

用上面的方法可以知道toArray方法返回的数组类型为Object，因而无法将装有Object类型对象的数组转换为指定类型的数组。

而toArray(T[])方法会借助泛型创建装有T类型的数组，返回的数组类型也为T类型，因此**应该使用toArray(T[])方法**。

结论：**toArray(T[])方法会返回指定类型的数组对象，toArray(T[])无需强转；而toArray()方法返回Object类型的数组对象，不能直接强转，只能对返回数组的具体元素进行强转。**

## Collections

Collections是一个工具类，包含很多实用的有关集合操作的方法（及其重载方法）。

如：

交换元素：swap(List,int,int)

填充集合：fill(List,T)

最大/最小值：max/min(Collection,Comparator)

打乱：shuffle(List)

排序：sort(List,Comparator)

二分查找：binarySearch(List,T)

创建一些特殊的集合（单元素集合、只读集合、同步集合etc）

…

## Array和Arrays

### Java.lang.reflect.Array

定义为final，最终类。该类的所有方法都为本地方法（native）。是通用数组的统一类实现形式。

### Java.util.Arrays

工具类，包含大量的方法来快捷的操作数组。

如：

拥有多个重载方法的排序实现：sort(int[]/double[]…)，sort(T[],comparator)

获得并行处理对象：spliterator(int[]/double[]…)

二分查找：binarySearch(byte[]/int,byte/int)

判断深度相等（多维数组）：deepEquals(Object[],Object[])

填充：fill(byte[],byte)

…

# assert

断言，一种常用的调试方式，包含一个boolean表达式，执行时假定表达式是true，若计算为false，系统会报告一个AssertionError并退出，用于调试。

1. assert expression1;

2. assert expression1: expression2;

assert 2 > 0 : "this is not true";  
assert false; // AssertionError

一个正确的程序必须保证expression为true，为false就表明程序处于一种不正确的情况下，系统会给出警告并退出。

一般情况下，assertion功能是关闭的，

* 要在编译时启用需加上 –source 1.4 参数（表示使用JDK1.4编译源码，assertion是JDK1.4添加的特性）。
* 要在运行时启用断言，可使用 -enableassertions 或者 -ea 标记。
* 要在运行时选择禁用断言，可使用 -da 或者 -disableassertions 标记。

# java GC（Garbage Collection）

1. [垃圾回收介绍](http://www.importnew.com/13504.html)
2. [垃圾回收是如何工作的？](http://www.importnew.com/13493.html)
3. [垃圾回收的类别](http://www.importnew.com/13827.htm)
4. [垃圾回收监视和分析](http://www.importnew.com/13838.html)

Java 垃圾回收是一项自动化的过程，用来管理程序所使用的运行时内存。通过这一自动化过程，JVM 解除了程序员在程序中分配和释放内存资源的开销。

每种JVM实现会采用不同的方法实现垃圾回收机制，基于HotSpot JVM分析GC。

与垃圾回收相关的主要组件为**堆内存**和**垃圾回收器**。在运行时，java实例被存放在堆内存区域，当一个对象不再被引用时，满足条件就会从堆内存移除。堆内存分为三个主要区域：新生代、老年代、永久代（javaSE 8 中被移除）。

GC会自动启动，程序员不需要在代码中显示启动垃圾回收过程。System.gc()和Runtime.getRuntime().gc()用来**请求**JVM启动垃圾回收。（启动由JVM负责，JVM可能拒绝这个请求）

**垃圾回收中实例的终结**

在释放一个实例和回收内存空间之前，Java 垃圾回收器会调用实例各自的 finalize() 方法，从而该使实例有机会释放所持有的资源。虽然可以保证 finalize() 会在回收内存空间之前被调用，但是没有确定的顺序和时间。多个实例间的调用顺序无法被预知，甚至可能会并行发生。**程序不应该预先调整实例之间的顺序并使用 finalize() 方法回收资源**。

* 任何在 finalize过程中未被捕获的异常会自动被忽略，然后该实例的 finalize 过程被取消。
* JVM 规范中并没有讨论关于弱引用的垃圾回收机制，也没有很明确的要求。具体的实现都由实现方决定。
* 垃圾回收是由一个守护线程完成的。

**对象什么时候符合垃圾回收的条件？**

所有实例都没有活动线程访问。

没有被其他任何实例访问的引用实例

[Java 中有不同的引用类型](http://javapapers.com/core-java/java-weak-reference/)。判断实例是否符合垃圾收集的条件都依赖于它的**引用类型**。

| **引用类型** | **垃圾收集** |
| --- | --- |
| 强引用（Strong Reference） | 不符合垃圾收集 |
| 软引用（Soft Reference） | 垃圾收集可能会执行，但会作为最后的选择 |
| 弱引用（Weak Reference） | 符合垃圾收集 |
| 虚引用（Phantom Reference） | 符合垃圾收集 |

Java编译器会将实例后从未被使用的对象赋值为null（java编译器的一种优化策略），标记实例可回收。

# String str = new String(“str”)

<http://www.cnblogs.com/OnlyCT/p/5433410.html>

执行语句String str = new String(“str”);时，不管字符串池中是否存在字符串”str”，会直接新建一个字符串（不是在字符串池中），然后赋给str。

## 总之，创建字符串有两种方式：两种内存区域(pool,heap)

1. ""创建的字符串在**字符串池**中；
2. new 创建字符串时，首先查看池中是否有相同的字符串，如果有则**拷贝**一份放到堆中，然后返回堆中的地址；如果池中没有则在堆中创建一分，然后返回堆中的地址；
3. 在对字符串赋值时，如果右操作数含有一个或一个以上的字符串引用时，则在堆中再建立一个字符串对象，返回引用如：String s= str1+"blog";（StringBuilder的toString方法会new新的String对象）

如：

String a = “a”;

String b = “b”;

String ab = a + b;

ab == “ab” **// false，ab的结果是运行时确定的，且通过StringBuilder拼接的。**

## 创建字符串的两种方式

[**http://www.cnblogs.com/fangfuhai/p/5500065.html**](http://www.cnblogs.com/fangfuhai/p/5500065.html)

1. 字面值方式赋值：String str = “aaa”

 采用字面值的方式创建一个字符串时，JVM首先会去字符串池中查找是否存在"aaa"这个对象，如果不存在，则**在字符串池中创建**"aaa"这个对象，然后将池中"aaa"这个对象的引用地址返回给字符串常量str，这样str会指向池中"aaa"这个字符串对象；如果存在，则不创建任何对象，直接将池中"aaa"这个对象的地址返回，赋给字符串常量。

1. new 关键字创建 String str3 = new String(“aaa”)

采用new关键字新建一个字符串对象时，JVM首先在字符串池中查找有没有"aaa"这个字符串对象，如果有，则不在池中创建"aaa"这个对象了，直接在堆中创建一个"aaa"（从字符串池中拷贝）字符串对象，然后将堆中的这个"aaa"对象的地址返回给引用str3，这样，str3就指向了堆中创建的这个"aaa"字符串对象；

**如果没有，则首先在字符串池中创建一个"aaa"字符串对象，**然后再在堆中创建一个"aaa"字符串对象，然后将堆中这个"aaa"字符串对象的地址返回赋给str3引用，这样，str3指向了堆中创建的这个"aaa"字符串对象。

## 字符串池的优缺点

* 优点就是避免了相同内容的字符串的创建，节省了内存，省去了创建相同字符串的时间，同时提升了性能；
* 字符串池的缺点就是牺牲了JVM在常量池中遍历对象所需要的时间，不过其时间成本相比而言比较低。

## intern方法

java有一个初始为空的字符串池，它由**类String独自维护**。当调用 intern方法时，**如果池已经包含一个等于此String对象的字符串（用equals(oject)方法确定），则返回池中的字符串。否则，将此String对象添加到池中，并返回此String对象的引用。** 对于任意两个字符串s和t，当且仅当s.equals(t)为true时，s.instern() == t.instern才为true。所有字面值字符串和字符串赋值常量表达式都使用 intern方法进行操作。

## **GC回收**

字符串池中维护了共享的字符串对象，这些字符串不会被垃圾收集器回收。

# equals 和hashcode

两个方法都在Object定义，所以任何对象都有这两个方法。

**equals()相等的两个对象，hashcode()一定相等，equals()方法不相等的两个对象，hashCode()有可能相等。**反过来：hashcode()不等，一定能推出equals()也不等；hashcode()相等，equals()可能相等，也可能不等。

在object类中，hashcode()方法是本地方法，返回的是**对象的地址值**，而object类中的equals()方法**比较的也是两个对象的地址值（==比较）**，如果equals()相等，说明两个对象地址值也相等，当然hashcode() 也就相等了；

当我们重写一个对象的equals方法，**就必须重写他的hashCode方法**，如果不重写他的hashCode方法的话，Object对象中的hashCode方法始终返回的是一个对象的hash内存**地址**，而这个**地址是永远不相等的**。所以这时候即使是重写了equals方法，也不会有特定的效果的，因为hashCode方法如果都不想等的话，就不会调用equals方法进行比较了，所以没有意义了。

当一个类覆写了equals却没有覆写hashCode方法，那么当这个类的两个实例对象用equals()方法比较的结果相等时，他们本来应该无法被同时存储进set集合中，但是，如果将他们**存储进HashSet集合**中时，由于他们的hashCode()方法的返回值不同(Object中的hashCode方法返回值是永远不同的)，第二个对象首先按照哈希码计算可能被放进与第一个对象不同的区域中，这样，它就不可能与第一个对象进行equals方法比较了，也就可能被存储进HashSet集合中了，（ Object类中的hashCode()方法不能满足对象被存入到HashSet中的要求，因为它的返回值是通过对象的内存地址推算出来的，同一个对象在程序运行期间的任何时候返回的哈希值都是始终不变的，所以，只要是两个不同的实例对象，即使他们的equals方法比较结果相等，他们默认的hashCode方法的返回值是不同的。）

HashMap的put方法：首先是判断hashCode是否相等，不相等的话，直接跳过，相等的话，再来比较这两个对象是否为同一对象（==）或者这两个对象的equals方法是否为true，因为是进行的**或**操作，所以只要有一个成立即可。

# 重写equals方法时需要注意什么

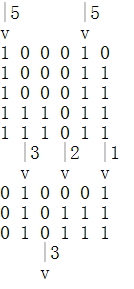
Object的equals实现为直接比较对象的内存地址是否相等，即比较是指向相同的引用。这是一种绝对相同的状态，因为两个对象本质上为同一个对象，只占据了一份内存空间。

另一种情况为部分内容相同就认为相等，我们可以重写equals方法以指定对象的哪些属性相同时，两个对象逻辑上相等。

## 重写equals的原则

1. **自反性**：对于任何**非空**引用x，x.equals(x)应该返回true。
2. **对称性**：对于任何引用x和y，如果x.equals(y)返回true，那么y.equals(x)也应该返回true。
3. **传递性**：对于任何引用x、y和z，如果x.equals(y)返回true，y.equals(z)返回true，那么**x.equals(z)也应该返回true**。
4. **一致性**：如果x和y**引用的对象没有发生变化**，那么反复调用x.equals(y)应该返回同样的结果。
5. **非空性**：对于任意**非空**引用x，x.equals(null)应该返回false。

你有无穷的水 和一个5公升的水桶和3公升的水桶 2个水桶形状大小都不同 你怎么才能准确称出4公升的水 写出详细流程？



不用任何函数库实现：

int=321 数值反转

String="abcde" 字符串反转

public static void main(String[] args) {  
 int b, a = 4321;  
 b = a;  
 int ar = 0;  
 int i = 0;  
  
 while ((b /= 10) > 0) {  
 i++;  
 }  
  
 while (a % 10 > 0) {  
 int c = 1;  
 for (int j = 0; j < i; j++) {  
 c \*= 10;  
 }  
  
 ar += a % 10 \* c;  
 a /= 10;  
  
 i--;  
 }  
 // ar == 1234  
  
}

public static void main(String[] args) {  
 String str = "abcd";  
 int len = str.length();  
 char[] result = new char[len];  
 for (int i = 0; i < len; i++) {  
 result[i] = str.charAt(len - i - 1);  
 }  
 // result -> "dcba"  
}

# hash表（散列表）

**Hash表：**是根据关键字而直接进行访问的数据结构，也就是说，散列表建立了**关键字和存储地址之间的一种直接映射**关系。

**散列函数：**一个把查找表中的关键字映射成该关键字对应的地址的函数，记为Hash（key）=Address。

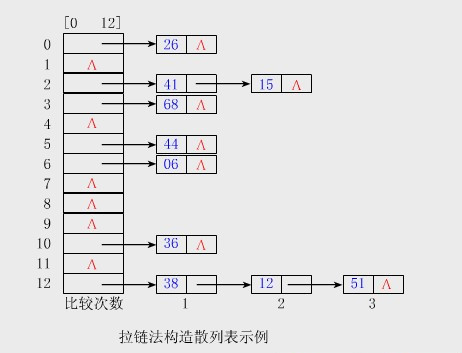
**冲突：**散列函数可能会把两个或两个以上的不同关键字映射到同一地址，称这种情况为“冲突”。

**常用的散列函数：**

1. 直接定址法——H（key）=a\*key+b
2. 除留余数法——H（key）=key%p
3. 数字分析法
4. 平方去中法

处理冲突的方法：

1. 拉链法：指把所有具有散列函数值的数据存储在一个线性链表中，这个链表由散列地址唯一标识。



1. 开放地址法