# JavaSE Foundation

### 关于计算机系统中的数据表示：

* 位：bit(只有0，1两种状态)，是计算机系统中的最小数据表示单位。
* 字节：byte, 1 byte = 8 bit.(因为1个bit有两种状态，所以1 byte 可以表示 2的8次方个状态，即256个状态: -128~127)
* KB： 1KB = 1024 byte;
* MB: 1MB = 1024KB;
* ……..

### 面向对象程序设计的三大基本特征：

* 继承(Inheritance)
* 封装(Encapsulation)：类包含了数据与方法，将数据与方法放在一个类中就构成了封装。

参数传递：Java的参数传递，无论传递的是原生数据类型还是引用类型，只有值传递(pass by value)。

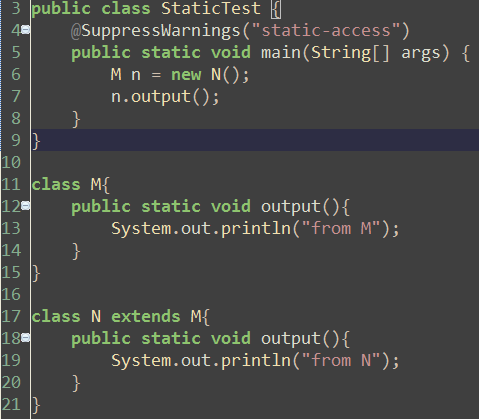
对于原生类型，传的就是值，传过去后你是你，我是我，再没有关联；

对于引用类型，即是把引用的值传过去，引用的值也即是对象的地址，传递后两个引用指向的是同一个地址，任何一个引用对对象产生了改变，都会影响另一个引用。

* 多态(Polymorphism)：父类型的引用指向了子类型的对象，从而引起引用具有的属性和方法发生了变化。

引用的能力是由声明它的类型决定的，声明的为什么类型就具有什么样的方法，至于调用方法时表现出来的是声明类型的特征还是实现类型的，就决定于实现类型有没有重写其方法，引用不能调用子类拥有但声明类型不拥有的行为，除非先进行强类型转换。

### static:



输出为”from M”

**子类没有办法重写父类的方法，可以理解为父类把自己的static方法隐藏了。**

**静态方法只能继承，不可重写。**

### **String s = “aaa”;这个语句做了什么事情？**

**有两种情况：**

* **查找Strings pool中是否存在“aaa”这个对象，如果不存在，则在Strings Pool中创建一个“aaa”对象，然后将Strings Pool中创建好的这个对象的地址返回来，赋给引用变量s,这样s就指向了Strings Pool中的这个“aaa”字符串对象。**
* **如果在Strings pool中存在“aaa”这个对象，则不创建任何对象，直接将“aaa”这个对象的地址返回给s**

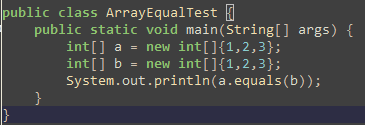
### String s = new String(“aaa”)；

* 首先在Strings Pool中查找有没有“aaa”这个对象，如果有，则不在Strings Pool中再去创建“aaa”这个对象了，直接在堆中(heap)创建一个“aaa”对象的，然后将堆中的这个“aaa”对象的地址返回来，赋给s引用，导致s指向了堆中创建的这个“aaa”字符串对象
* 如果没有，则首先在Strings Pool中创建一个“aaa”对象，然后再在堆中创建一个“aaa”对象，然后将堆中的这个“aaa”对象的地址返回来，赋给s引用，导致s指向了堆中创建的这个“aaa”字符串对象

### String的intern()方法功能：

|  |
| --- |
| **intern**  public [String](mk:@MSITStore:E:\Documents\Doc\JDK1.6.chm::/java/lang/String.html) **intern**()  Returns a canonical(典型的，规范的) representation for the string object.  A pool of strings, initially empty, is maintained privately by the class String.  When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the [equals(Object)](mk:@MSITStore:E:\Documents\Doc\JDK1.6.chm::/java/lang/String.html#equals(java.lang.Object)) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.  It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true.  All literal(文字的) strings and string-valued constant expressions are interned. String literals are defined in §3.10.5 of the [Java Language Specification](http://java.sun.com/docs/books/jls/html/)  **Returns:**  a string that has the same contents as this string, but is guaranteed to be **from a pool of unique strings**. |

### Array equal test:



输出为false; 数组中的equal方法是比较地址的，如果真的要比较内容，则使用Arrays.equals(int[] a, int[]b)。

### 拷贝数组可以使用System.arrayCopy(Object src, int srcPos, Object dest, int destPos, int len);

### 常用排序法：

冒泡排序、快速排序、交换排序和二分查找法。

### 作业：

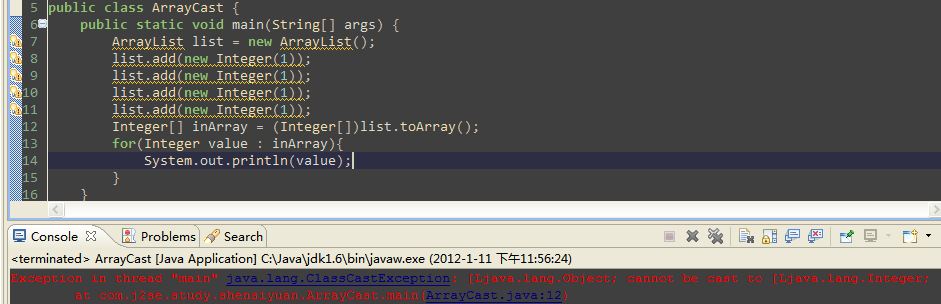
随机生成50个数字（整数），每个数字的范围是[10,50]，统计每个数字出现的次数以及出现次数最多的数字与它的个数，最后将每个数字及其出现次数打印出来，如果某个数字出现次数为0，则不要打印它。打印的时候按照数字的升序排列。

### Eclipse有一个我们没有注意到但很有用的功能。



如上图当前光标在ENTER的后面，一般我们会按箭头把光标移动行尾，但其实不需要使用箭头，只需要在这个时候按一下回车键就自动跳到行尾了。

### Object[]不能强制类型转换到Integer[]



## Collection

### 集合里放的和数组一样是对象的引用，而不是对象的本身。

### Arraylist源代码分析：

1. Arraylist维护着一个叫做elementData的**Object**（**不是泛型类型的，它在get方法里先在内部强制类型转换再返回它的值**）数组、size属性，从AbstractList中继承过来的modCount属性，这个类中大量使用了Arrays.copyOf()和System.arrayCopy方法。

Arraylist底层采用数组实现，当使用不带参数的构造方法生成Arraylist对象时，实际上会在底层生成一个长度为10的Object数组，如果增加的元素个数超过了10个，那么Arraylist底层会新生成一个长度为原数组的1.5倍+1的数组。然后将原数组的内容复制到新数组当中，并且后续增加的内容都会放到新数组当中。当新数组无法容纳增加的元素时，重复这个过程。

对于Arraylist元素的删除操作，需要将被删除元素的后续元素向前移动，代码比较高。完成元素的移动后，并不会改变数组的长度，但会将最后一个“有值的”位置置为null，以便GC进行内存回收。

|  |
| --- |
| elementData[--size] = null; // Let gc do its work |

ensureCapacity方法代码如下：

|  |
| --- |
| /\*\*  \* Increases the capacity of this <tt>ArrayList</tt> instance, if  \* necessary, to ensure that it can hold at least the number of elements  \* specified by the minimum capacity argument.  \*  \* @param minCapacity the desired minimum capacity  \*/  public void ensureCapacity(int minCapacity) {  modCount++;  int oldCapacity = elementData.length;  if (minCapacity > oldCapacity) {  Object oldData[] = elementData;  int newCapacity = (oldCapacity \* 3)/2 + 1;  if (newCapacity < minCapacity)  newCapacity = minCapacity;  // minCapacity is usually close to size, so this is a win:  elementData = Arrays.copyOf(elementData, newCapacity);  }  } |

toArrays方法（不是直接返回elementData）

|  |
| --- |
| /\*\*  \* Returns an array containing all of the elements in this list  \* in proper sequence (from first to last element).  \*  \* <p>The returned array will be "safe" in that no references to it are  \* maintained by this list. (In other words, this method must allocate  \* a new array). The caller is thus free to modify the returned array.  \*  \* <p>This method acts as bridge between array-based and collection-based  \* APIs.  \*  \* @return an array containing all of the elements in this list in  \* proper sequence  \*/  public Object[] toArray() {  return Arrays.copyOf(elementData, size);  } |

1. Arraylist和Linkedlist的比较分析：

* Arraylist底层使用数组实现，Linkedlist底层采用双向链表实现。
* 当执行插入操作时，采用Linkedlist比较好。
* 当执行搜索操作时，采用Arraylist比较好。
* 当向Arraylist添加一个对象时，实际上就是将该对象放置到了Arraylist底层所维护的数组当中；当向Linkedlist中添加一个对象时，实际上LinkedList内部会生成一个Entry对象，该Entry对象的共有三个 属性： E element; Entry<E> next; Entry<E> previous;当中的element就是我们向LinkedList中所添加的元素，然后Entry又构造好了向前与向后的引用previous、next，最后将生成的这个Entry对象加入到了链表当中。**换句话说，LinkedList中所维护的是一个个的Entry对象，而不是我们添加的对象本身。**

1. 使用LinkedList来实现自己的Queue和Stack
2. Equals方法的API描述：

|  |
| --- |
| public boolean **equals**([Object](mk:@MSITStore:E:\Documents\Doc\JDK1.6.chm::/java/lang/Object.html) obj)  Indicates whether some other object is "equal to" this one.  The equals method implements an equivalence relation on non-null object references:   * It is *reflexive（自反性）*: for any non-null reference value x, x.equals(x) should return true. * It is *symmetric（对称性）*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true. * It is *transitive（传递性）*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true. * It is *consistent（一致性）*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified. * For any non-null reference value x, x.equals(null) should return false.   The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).  Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes. |

hashCode

|  |
| --- |
| public int **hashCode**()  Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable.  The general contract of hashCode is:   * Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently(一致地) return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application. 在java应用的一次执行过程当中，对于同一个对象的hashCode方法的多次调用，它们应当返回同样的值。（前提是该对象equals方法使用到的信息并没有发生变化），在java应用的多次执行过程中并不要求每次执行过程中返回的hashCode值要一致，即可以一致也可以不一致。 * If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. 当equals方法返回true那么这两个对象的hashCode值必须是一致的。 * It is *not* required that if two objects are unequal according to the [equals(java.lang.Object)](mk:@MSITStore:E:\Documents\Doc\JDK1.6.chm::/java/lang/Object.html#equals(java.lang.Object)) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables. 两个对象如果它们的equals方法返回false,并不要求它们的hashCode一定要不一样，但对程序员而言，应当要了解到，如果不同则可以提高应用的性能。   As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)  **Returns:**  a hash code value for this object.  **See Also:**  [equals(java.lang.Object)](mk:@MSITStore:E:\Documents\Doc\JDK1.6.chm::/java/lang/Object.html#equals(java.lang.Object)), [Hashtable](mk:@MSITStore:E:\Documents\Doc\JDK1.6.chm::/java/util/Hashtable.html) |

1. 所谓的字典顺序是按ASCII来说的，所以如果要按字典排序来对字母进行排序的话，小写字母要排在大写字母的前面，
2. **Collections.reserveOrder();方法会返回一个comparator，所以要对列表进行反向排序有两种方法：**

|  |
| --- |
| **Arrays.sort(a, Collections.reverseOrder());** |

**和**

|  |
| --- |
| **Collections.reservse(list);** |

1. **Map的keySet()方法会返回key的集合，因为Map的键是不可能重复的，所以返回的是Set类型，而Map的值是可以重复的，所以它的values()方法返回的就是Collection类型。**
2. Main方法是可以传递参数的，如java test aa bb cc，那么这个main方法将有三个参数，默认会以空格来分割参数，但如果当要传递的参数里本来就带空格时，就可以用下来的方法来解决：java test “aa bb” cc，这时main方法就只有两个参数了。
3. 类里面可以再定义类，接口里面也可以再定义接口，如Map这个接口里就再定义了另一个接口Entry。
4. 遍历Map的常用两种方式：

|  |
| --- |
| Set<String> keySet = map.keySet();  for(String key : keySet){  System.out.println(key + "=" + map.get(key));  } |

和：

|  |
| --- |
| Set<Map.Entry<String, Integer>> entrySet = map.entrySet();  for(Map.Entry<String, Integer> entry : entrySet){  entry.getKey();  entry.getValue();  } |

1. Strategy Pattern

* 抽象策略角色
* 具体策略角色
* 环境类
* 客户端

抽象策略角色：

|  |
| --- |
| public interface Strategy {  int caculate(int a,int b);  } |

具体策略角色：

|  |
| --- |
| public class AddStrategy implements Strategy {  @Override  public int caculate(int a, int b) {  return a + b;  }  } |

具体策略角色：

|  |
| --- |
| public class MulStrategy implements Strategy {  @Override  public int caculate(int a, int b) {  return a \* b;  }  } |

环境类：

|  |
| --- |
| public class Enviroment {  private Strategy strategy;  public Enviroment(Strategy strategy) {  this.strategy = strategy;  }  public void **setStrategy**(Strategy strategy) {  this.strategy = strategy;  }  public int **caculate**(int a, int b){  return strategy.caculate(a, b);  }  } |

客户端：

|  |
| --- |
| public class Client {  public static void main(String[] args) {  Enviroment env = new Enviroment(new AddStrategy());  System.out.println(env.caculate(3, 4));    env.**setStrategy**(new MulStrategy());  System.out.println(env.caculate(3, 4));  }  } |

Strategy Pattern’s disadvantage:

* The client must know all the implemented strategy, and have to decide which strategy to use by itself.
* There will be many implemented strategy classes.

Solutions:

Use factory pattern.

1. **HashSet底层是使用HashMap实现的。当使用Add方法将对象添加到Set中时，实际上是将该对象作为底层所维护的Map对象的Key，而Value则一直是同一个Object对象PRESENT（这个对象我们用不上）**
2. **HashMap底层维护一个数组，我们向HashMap中所放置的对象实际上是存储在该数组当中；当向HashMap中put一对键值时，它会根据key的hashCode值计算出一个位置，该位置就是此对象准备往数组中存放的位置。如果该位置没有对象存在，就将此对象直接放到数组当中，如果该位置已经有对象存在了，则顺着此存在的对象的链开始寻找（Entry类有一个Entry类型的next成员变量，指向了该对象的下一个对象）。如果此链上有对象的话，再去使用equals方法进行比较，如果对此链上的某个对象的equals方法比较为false，则将该对象放到数组当中，然后将数组中该位置以前存在的那个对象链接到此对象的后面。**
3. 下面的代码是否能编译通过？

|  |
| --- |
| GenericFoo<Integer> foo1 = new GenericFoo<Integer>();  foo1.setFoo(20);  String b = (String)foo1.getFoo(); |

不能。类型转换只能在不知道被转型类型是什么类型的情况下发生，如果编译时已经知道被转型对象的具体类型，则不会通过编译。

1. 在定义泛型类别时，预设可以使用任何的类型来实例化泛型类型中的类型，但是如果想要限制使用泛型类别时，只能用某个特定类型或者其子类型才能实例化该类时，可以在定义类型时，使用extends关键字指定这个类型必须是继承某个类，或者实现某个接口。如：Class MyGeneric<T extends List> {}，当没有指定泛型继承的类型或者接口时，默认为T extends Object。
2. GenericTest< ? extends List> ge = null;这个语句声明了ge这个引用可以指向如下的类型：

Ge = new GenericTest<ArrayList>();也可以指向: ge = new GenericTest<LinkedList>();(这个时候类的声明为：class GenericTest<T>{})

1. 28使用的方式是在类声明的时候就指定实例化的类型必须是List的实现类，而29这种方式是在使用该类的时候才指定该引用的类型。
2. **使用<?>或是<? Extends SomeClass>的声明方式，意味着您只能通过该名称来取得所参考实例的信息，或者是移除某些信息，但不能增加它的信息，因为只知道当中放置的是SomeClass的子类，但不确定是什么类的实例，编译器不让您加入信息，理由是，如果可以加入信息的话，那么您就得刻取回的实例是什么类型，然后转换为原来的类型方或进行操作，这就失去了使用泛型的意义。**

|  |
| --- |
| **GenericTest<String> ge1 = new GenericTest<String>();**  **ge1.setFoo("hello");**  **GenericTest<? extends Object> ge2 = ge1;**  **System.*out*.println(ge2);**  **ge2.setFoo(null);**  **System.*out*.println(ge2);**  **//ge2.setFoo("welcome");** |

**上面注释的代码会引起编译错误。**

## Reflection

1. 一般而言，开发者社区说到动态语言，大致认同的一个定义是：“程序运行时，允许改变程序结构或变量类型，这种语言称为动态语言”。从这个观点上看，Perl，Python，Ruby，JavaScript是动态语言，C++，Java，C#不是动态语言。
2. 获取某个类或某个对象所对应的Classic对象的常用的3种方式：

* 使用Class类的静态方法forName：Class.forName(“java.lang.String”);
* 使用类的.class方法：String.class
* 使用对象的getClass()方法：String s = “aa”; Class<?> clazz = s.getClass();

1. 通过类的不带参数的构造方法来生成对象，有两种方式：

* 先获得Class对象，然后通过该Class对象的newInstance()方法直接生成即可：Class<?> classType = String.class; Object obj = classType.newInstance();
* 先获得Class对象，然后通过该对象获得对应的Constructor对象，再通过该Contructor对象的newInstance()方法生成
* Class<?> classType = Customer.class;
* Constructor cons = classType.getConstructor(new Class[]{});
* Object obj = cons.newInstance(new Object[]{});

1. 若想通过类的带参数的构造方法生成对象，只能使用下面的这一种方式：

* Class<?> classType = Customer.class;
* Constructor cons = classType.getConstructor(new Class[]{String.class,int.class});//指定参数类型
* Object obj = cons.newInstance(new Object[]{“hello”,3});

## IO

1. FileImportStream fin = new FileInputStream(“employee.dat”); Java IO中的类都将相对路径名解释为以**用户工作目录开始**，但在不同的环境会有不同的输出，可以用System.getProperty(“user.dir”)查看：

|  |
| --- |
| package com.j2se.study.io.corejava;  public class FileDirectory {  public static void main(String[] args) {  System.out.println(System.getProperty("user.dir"));  }  } |

在eclipse中运行输出：E:\WorkSpace\SwingProjects\J2SE（即是当前工程的根目录）

把文件放于C盘中，并注释掉包声明，即不生成包目录，输出为：C:\

把文件放于C盘中，不注释掉包声明，即生成包目录，输出依然为：C:\

1. Using the buffer mechanism for DataInputStream

If you want buffering and the data input methods for a file, you need to use the following rather monstrous sequence of constructors:

|  |
| --- |
| DataInputStream din = **new** DataInputStream(**new** BufferedInputStream(**new** FileInputStream("employee.dat"))); |

1. For some IO classes, we can use the checkError method to check that it occur errors or not for current IO object. Like PrintWriter
2. You obtain a Charset by calling the static forName method with either the official name or one of its aliases:

|  |
| --- |
| Charset cset = Charset.forName(“UTF-8”); |

1. Character set names are case insensitive.
2. For compatibility with other naming conventions, each character set can have a number of aliases. For example, ISO-8859-1 has aliases.

## 装饰模式

## 序列化

1. 一个类若想被序列化，则需要实现java.io.Serializable接口，该接口中没有定义任何方法，是一个标识性接口(Marker Interface)，当一个类实现了该接口，就表示这个类的对象是可以序列化的。

2. 在序列化时，static变量是无法序列化的；如果A包含了对B的引用，那么在序列化A的时候也会将B一并地序列化；如果此时A可以序列人，B无法序列化，那么当序列化A的时候就会发生异常，这时就需要将对B的引用设为transient，该关键字表示变量不会被序列化。

3．将对象从内存中保存到文件称为序列化 ，将保存在文件中的对象读取到内存中称为反序列化。

4. 当我们在一个待序列化/反序列化的类中实现了以下两个private方法(方法声明要与上面的保持完全的一致),那么就允许我们以更加底层，更细粒度的方式控制序列化/反序列化的过程。

|  |
| --- |
| private void writeObject(java.io.ObjectOutputStream out)  throws IOException  private void readObject(java.io.ObjectInputStream in)  throws IOException, ClassNotFoundException;  private void readObjectNoData()  throws ObjectStreamException; |

5. 关于实现Serializable接口的类中 的serialVersionUID问题

如果你的对象序列化后存到硬盘上面后，可是后来你却更改了类的field（增加或减少或改名），当你反序列化时，就会出现Exception的，这样就会造成不兼容性的问题

但当serialVersionUID相同时，它就会将不一样的field以type的缺省值Deserialize，这个可以避开不兼容性的问题。即如果Deserialize时发现field多了，则给增加的field赋以默认值，如果Field少了就把它忽略掉，即以当前编译出来的class文件为基准。

## 线程

进程和线程的区别：

* 多个进程的内部数据和状态都是完全独立的，而多线程是共享一块内存空间和一组系统资源，有可能互相影响
* 线程本身的数据通常只有寄存器数据，以及一个程序执行时使用的堆栈，所以线程的切换比进程切换的负担要小。

多线程编程的目的：

* 多线程编程的目的，就是“最大限度的利用CPU资源”，当某一线程的处理不需要占用CPU而只和I/O等资源打交道时，让需要占用CPU资源的其它线程有 机会获得CPU资源。从根本上说，这就是多线程编程的最终目的。

Process VS Thread

* 多线程程序比多进程程序需要更少的管理费用。进程是重量级的任务，需要分配给它们独立的地址空间。进程间通信是昂贵和受限的。进程间的转换也是很需要花费的。另一方面，线程是轻量级的选手。它们共享相同的地址空间并且共同分享同一个进程。线程间通信是便宜的，线程间的转换也是低成本的。

关于创建的线程优先级的API说明：

|  |
| --- |
| Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon. |

Start()方法：

Start()首先为线程的执行准备好系统资源，然后再去调用run方法。如果直接调用线程的run方法，那么这个线程就相当于普通的类，而run方法就相当于这个普通类里的一个普通的方法，不会启动新的线程，要启动线程只有通过调用start方法。

Thread源代码分析：

|  |
| --- |
| **public** **synchronized** **void** start() {  /\*\*  \* This method is not invoked for the main method thread or "system"  \* group threads created/set up by the VM. Any new functionality added  \* to this method in the future may have to also be added to the VM.  \*  \* A zero status value corresponds to state "NEW".  \*/  **if** (threadStatus != 0)  **throw** **new** IllegalThreadStateException();  group.add(**this**);  start0();  **if** (stopBeforeStart) {  stop0(throwableFromStop);  }  }  **private** **native** **void** start0() |

在start()，我们无法看到start()显式的调用run方法，但它调用了一个本地的start0()方法，可以相像start0()方法就是分配系统资源和调用run方法的。

线程的停止

线程的停止无法通过调用stop方法来达到，只能通过在run方法里写相应的逻辑让它结束run方法，从而达到停止线程的目的。

线程的生命周期：

* 创建状态

当用new操作符创建一个新的线程对象时，该线程处于创建状态。处于创建状态的线程只是一个空的线程对象，系统不为它分配线程资源

* 可运行状态

执行线程的start方法将为线程分配必须的系统资源，安排其运行，并调用线程体run方法，这样就使得该线程处于可运行状态(runnable)状态。这一状态并不是运行中状态(Running)因为线程也许实际上并未真正运行。

* 不可运行状态

当发生下列事件时，处于运行状态的线程会转入不可运行状态。1. 调用了sleep()方法。2. 线程调用wait方法等待特定条件的满足 3. 线程I/O阻塞。如果上述三个情况得到解决1. 指定的睡眠时间过去 2. 如果线程等待某一条件，另一个对象必须通过notify()或notifyAll方法通知等待线程条件的改变。 3. 如果I/O阻塞的情况得到解决。这时线程会返回可运行状态。

* 消亡状态

当线程的run方法执行结束后，该线程自然消亡

线程的状态转换图：

线程的优先级：

线程的优先级是一个动态的变量，即使你设置某个线程的优先级为3，OS也有可能动态改变它的值，这是因为OS为避免某些优先级低的线程永远得不到运行的机会的策略。所以绝对不能依赖于优先级来调度各个线程间的调用顺序。

线程的调度策略

线程调度器选择优先级最高的线程运行。但是，如果发生以下情况，就会终止线程的运行。

* 线程体中调用了yield()方法，让出了对CPU的占用权
* 线程体中调用了sleep()方法，使线程进入睡眠状态
* 线程由于I/O操作而受到阻塞
* 另一个更高优先级的线程出现
* 在支持时间片的系统中，该线程的时间片用完。

Java中的每一个对象都有一个锁(Lock)或者叫做监视器(Monitor)，当访问某个对象的synchronized方法时，表示**将该对象上锁**，此时其它任何线程都无法再去访问**该对象所有的synchronized方法**了，直到之前的那个线程执行方法完毕后(或者抛出了异常)，那么将该对象的锁释放掉，其它的线程才有可能再去访问该synchronized方法。

如果一个对象有多个synchronized方法，某一时刻某个线程已经进入了某个synchronized方法，那么在该方法没有执行完毕前，其它线程是无法访问**同一个对象**的任何synchronized方法的，但不同对象间不会有任何影响

如果某个synchronized方法是static的，那么当线程访问方法时，它锁的并不是synchronized方法所在的对象，而是synchronized方法所在的对象所对应的class对象，因为Java中无论一个类有多少个对象，这些对象只会对应唯一一个Class对象，因此当线程分别访问同一个类的两个对象的两个static，synchronized方法时，他们执行的顺序也是顺序的，也就是说一个线程先去执行方法，执行完毕后另一个线程才开始执行。如下面的程序

|  |
| --- |
| **public** **class** SynchronizedTest {  **public** **static** **void** main(String[] args) {  Example example = **new** Example();    Thread thread1 = **new** MyThread1(example);    example = **new** Example();    Thread thread2 = **new** MyThread2(example);    thread1.start();  thread2.start();  }  }  **class** Example{  **public** **synchronized** **static** **void** execute1(){  **for**(**int** i=0;i<20;i++){  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println("hello:" + i);  }  }    //如果execute2方法没有static，只有execute1有，  //那么执行的顺序还是乱序的  **public** **synchronized** **static** **void** execute2(){  **for**(**int** i=0;i<20;i++){  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println("world:" + i);  }  }  }  **class** MyThread1 **extends** Thread{  **private** Example example;    **public** MyThread1(Example example){  **this**.example = example;  }  @Override  **public** **void** run() {  example.*execute1*();  }  }  **class** MyThread2 **extends** Thread{  **private** Example example;    **public** MyThread2(Example example){  **this**.example = example;  }  @Override  **public** **void** run() {  example.*execute2*();  }  } |

Wait与notify方法都是定义在Object类中，而且是Final的，因此会被所有的Java类所继承且无法重写。这两个方法要求在调用时线程应该已经获得了对象的锁，因此对这两个方法的调用需要放在synchronized方法或块当中。**当线程执行了wait方法时，它会释放掉对象的。Notify则会随机的唤醒任意一个等待中的线程**。

另一个会导致线程暂停的方法就是sleep方法，它会导致线程睡眠指定的毫秒数，**但线程在睡眠过程中是不会释放掉对象的锁的。**

## Deep clone and Shallow clone

Java中对象的克隆：

* 为了获取对象的一份拷贝，我们可以利用Object类的Clone方法
* 在派生类中覆盖基类的Clone方法，**并声明为Public**
* 在派生类的Clone方法中，**调用super.clone()**
* 在派生类中**实现Cloneable接口**

说明：

* 为什么我们在派生类中覆盖Object的Clone方法时，一定要调用super.clone呢？
* 在**运行时刻**，Object中的clone识别出你要复制的是那一个对象，然后为此对象分配空间，并进行对象的复制，将原始对象的内容一一复制到新对象的存储空间中
* 继承自Object类的clone方法是**浅复制**

## 网络编程

端口号：通过一个IP地址可以确定一个机器的位置，但跟该机器上的那一个程序连接就是依靠端口号来确定的。

TCP：Transfer Control Protocol的简称，是一种面向连接的保证可靠传输的协议。通过TCP协议传输，得到的是一个顺序的无法差错的数据流。发送方和接收方的成对的两个socket之间必须建立连接以便在TCP协议的基础上进行通信，当一个Socket（通常都是server socket）等待建立连接时，另一个socket可以要求进行连接，一旦这两个socket连接起来，它们就可以进行双向数据传输，双方都可以进行发送或接收操作。

TCP是一个基于连接的协议，它能够提供两台计算机之间的可靠的数据流。HTTP, FTP, TELNET等应用都需要这种可靠的通信通道。

UDP:是User Datagram Protocol（用户数据报协议）是一种无法连接的协议，每个数据都是一个独立的信息，包括完整的源地址或目的地址，它在网络上以任何可能的路径传往目的地，因此能否到达目的地，到达目的地的时间以及内容的正确性都是不能被保证的。常用于在网络中传输视频，音频等数据量大的传输。

UDP是从一台计算机向另一台计算机发送称 为数据报的独立数据包的协议，该协议并不保证数据报是否能正确地到达目的地。它是一个非面向连接的协议。

下面我们对这两种协议做简单的比较

* 使用UDP时，每个数据报中都给出了完整的地址信息，因此无需要建立发送方和接收方的连接
* 对于TCP协议，由于它是一个面向连接的协议，在socket之间进行数据传输之前必然要建立连接，所以在TCP中多了一个连接建立的时间
* 使用UDP输入数据是有大小限制的，每个被传输的数据报必须限定在64KB之内
* TCP没有这方面的限制，一旦连接建立起来，双方的socket就可以按统一的格式传输大量的数据
* UDP是一个不可靠的协议，发送方所发送的数据报并不一定以相同的次序到达接收方
* TCP是一个可靠的协议，它确保接收方完全正确地获取发送方所发送的全部数据

## JVM

1. 号称是每一位java程序员都会做错的题目

|  |
| --- |
| class Singleton {  private static Singleton singleton = new Singleton();  public static int counter1;  public static int counter2 = 0;  private Singleton() {  counter1++;  counter2++;  }  public static Singleton getInstance() {  return singleton;  }  }  public class MyTest {  public static void main(String[] args) {  Singleton singleton = Singleton.getInstance();  System.out.println("counter1=" + singleton.counter1);  System.out.println("counter2=" + singleton.counter2);  }  } |

输出1和0

1. 类的加载、连接与初始化

加载：查找并加载类的二进制数据

连接：

验证：确保被加载的类的正确性

准备：为类的静态变量分配内存，并将其初始化为默认值

解析：把类中的符号引用转换为直接引用

初始化：为类的静态变量赋予正确的初始值

Note：

静态变量在类实例化之前就已经分配内存了

如public static final int a = 3;对于a这个变量会进行两次赋值，先赋默认值，再赋3