

Java程序设计

(上)

2015.11.30

isszym sysu.edu.cn

概述

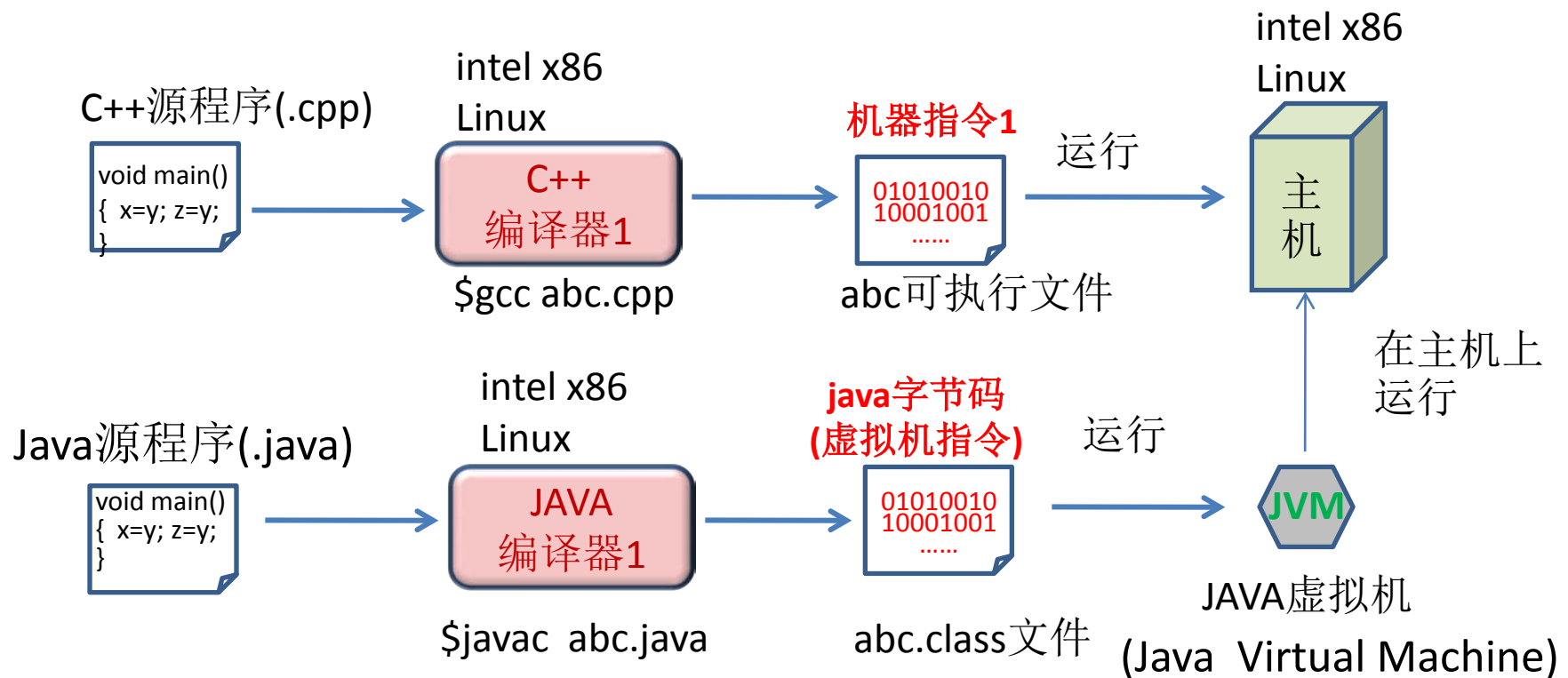
- 1991 年，Sun Microsystems 的James Gosling 、 Patrick Naughton 、 Chris Warth 、 Ed Frank和Mike Sheridan共同构想了Java语言。
- 令人惊讶的是设计Java 的最初动力是想构建一种独立于平台的语言，使该语言生成的代码可以在不同环境下的不同CPU上运行，以满足消费类电子设备(洗衣机、空调和微波炉等)的软件的需要。
- 就在快要设计出Java 的细节的时候，另一个在Java 未来中扮演关键角色的更重要的因素出现了。这就是World Wide Web。
- 1993 年Web出现后，Java 的重点立即从消费类电子产品转移到了Internet 程序设计，并最终促成了Java 的燎原之势。

<http://docs.oracle.com/javase/8/docs/>

<http://api.apkbus.com/reference/java/io/package-summary.html>

C++编译器和Java虚拟机

- C++源程序(.cpp)采用intel x86 Linux的C++编译器gcc编译成机器指令文件。该文件只能在intel x86 Linux系统上执行。
- JAVA源程序(.java)采用intel x86 Linux的Java编译器javac编译成JAVA字节码文件(.class)。该文件可以在所有安装了Java虚拟机(JVM)的系统上执行。



Java的特点

□ 简单(Simple)

设计Java的目的之一就是简化C++的功能，使其易于学习和使用。它**没有指针类型(pointer)**，避免了内存溢出等安全性问题。采用**垃圾收集器(garage collection)**自动收集存储空间的垃圾，使程序员摆脱了时常忘记释放存储空间的苦恼。

□ 易于移植(Portable)

通过使用Java字节码，Java支持交叉平台代码，Java程序可以在任何可以运行Java虚拟机的环境中执行，其执行速度被高度优化，有时甚至超过了C++的程序。

□ 面向对象(Object-oriented)

Java语言中一切都是对象，并且Java程序带有大量在运行时用于检查和解决对象访问的运行时(run-time)类型信息。

JRE和JDK

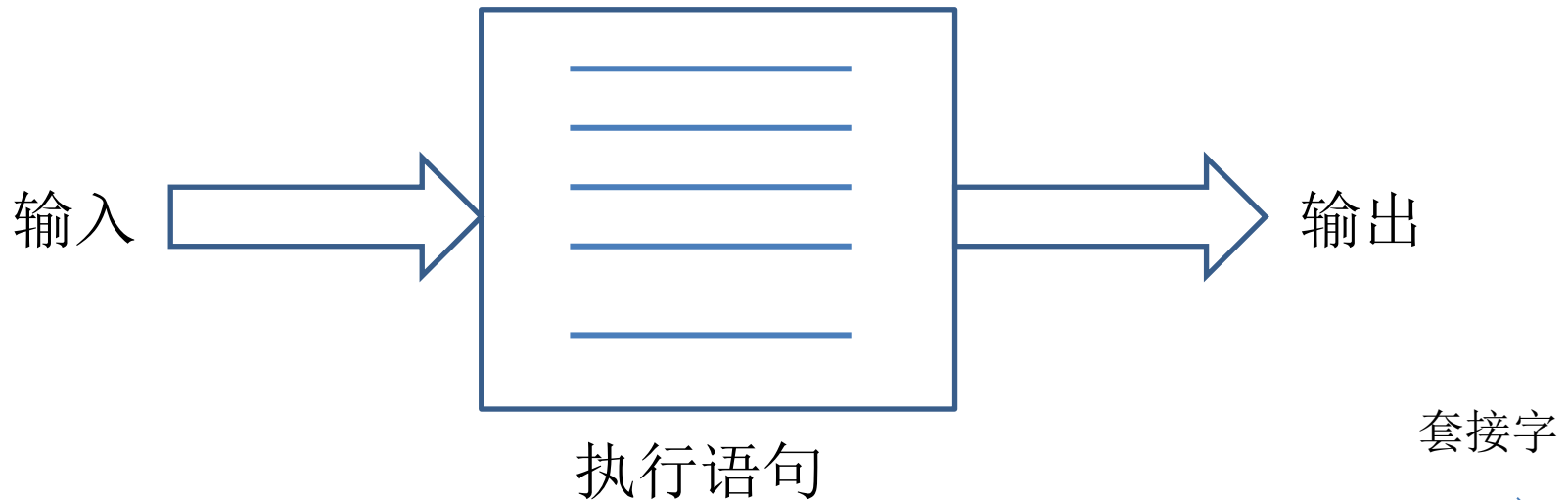
- Java平台有两个主要产品：Java Runtime Environment (JRE) 和Java Development Kit (JDK) 。
- JRE提供Java库、Java虚拟机以及运行Java应用程序所需的其它组件。字节码文件就是在JRE下运行的。
- JDK是JRE的超集，用于开发Java小程序和Java应用程序。它包括了JRE所有的内容, 并且加入了编译器和调试器等工具。JDK共有三个版本，分别用于不同的目的：

- ✓ **Standard Edition(Java SE):** 标准版，是最常用的一个版本。
- ✓ **Enterprise Edition(Java EE):** 企业版，用于开发大型Java应用程序。
- ✓ **Micro Edition(Java ME):** 微型版，用于移动设备上java应用程序。

安装JDK见附录2

程序设计语言

Java是一种面向对象的程序设计语言。用程序设计语言可以编制程序完成**计算任务**。

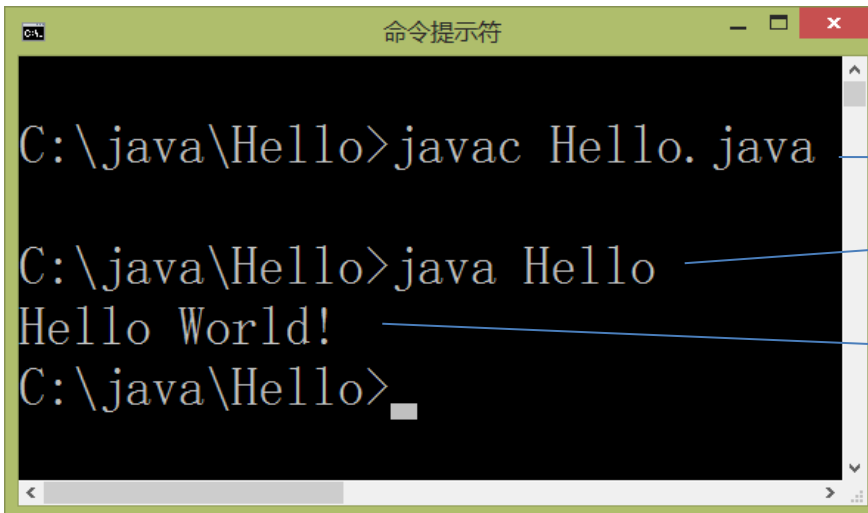


数据：常量、变量、对象
语句：赋值、循环、条件

输入输出：键盘显示器、文件、数据库、网络
执行：顺序执行或并发执行(线程)

第一个Java程序

```
//Hello.java
public class Hello           // 类名，要与文件名一致!!!
{
    public static void main(String args[]) // 主程序入口
    {
        System.out.print("Hello World!"); // 显示Hello World!
    }
} // print()为系统对象System.out的方法
```



```
C:\java\Hello>javac Hello.java
C:\java\Hello>java Hello
Hello World!
C:\java\Hello>
```

编译并生成Hello.class

运行Hello.class

显示结果

安装Java见附录

数据类型

- 程序的每个数据都需要以一定的格式存放，并可以参与运算，这就需要定义数据类型，比如，整数类型、字符类型等。
- Java中共有8种基本数据类型：

类型	字节	数据容器(类)	数的范围	默认值
byte	1	Byte	-128~127	0
short	2	Short	-32768~32767	0
int	4	Integer	$-2^{31} \sim 2^{31}-1$	0
long	8	Long	$-2^{63} \sim 2^{63}-1$	0
float	4	Float	$3.4e^{-038} \sim 3.4e^{+038}$	0.0
double	8	Double	$1.7e^{-308} \sim 1.7e^{+308}$	0.0
char	2	Character	0~65535	0
boolean	1	Boolean		false

*数值类型指short、int、long、float、double

常量

常量是指程序中不变的量。通常需要为常量命名后使用。

布尔常量:	<code>true false</code>
整型常量:	<code>100</code> (10进制整数) <code>016</code> (8进制整数) <code>0x2EF</code> (16进制整数) <code>386L</code> (长整数)
浮点常量:	<code>19.6f 3.14E3F</code> (3.14×10^3) <code>-- float</code> <code>3.14 2.536D 3.1415926E-3D</code> <code>-- double</code>
字符常量:	<code>'a'</code> 、 <code>'8'</code> 、 <code>'#'</code> 、 <code>'\n'</code> (换行)、 <code>'\r'</code> (回车) <code>'\\'</code> 、 <code>'\"'</code> 、 <code>'\''</code> 、 <code>'\u0027'</code> (单引号, unicode码)。

*其中所有字母不区分大小写

```
final int LEVEL_NUM=0x1000; //为常量命名是个好习惯
```

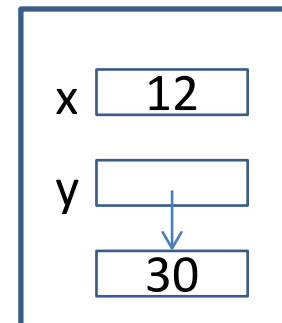
变量

- 如果有一个值在计算过程中会发生变化，比如，测量的气温值，就需要把它定义为**变量**。
- 变量是通过赋值语句改变其值。赋值语句用等于号连接变量和表达式，变量位于左边，表达式位于右边。表达式计算出来的值用于修改变量的值。
- 首先要为变量取一个名字，然后再给它赋值。**Java的变量命名规则**：
\$、字母、下划线开头的数字字母下划线串。为了分配内存和进行计算，还需要为变量定义数据类型。

```
float temperature;    // 定义一个浮点类型的变量
temperature = 26;     // 把右边的值赋给左边的变量
```

- 系统为每个变量都会分配一个存放位置，用来存放该变量的值，也可以存放一个指针，指向存放变量值的地方。

```
int x = 12;
Integer y = 30; //y为对象
```



运算符与表达式

➤ 算术表达式：用算术运算符形成的表达式，计算结果为整值

```
int x = 5, y = 6, z = 10;    // 一次定义多个变量，并赋初值
int exp = (y + 3) * z + x;    // 右边为算术表达式，exp得值95
```

➤ 关系表达式：用关系运算符形成的表达式，计算结果为真假值

```
int x = 300;
boolean r1 = x > 10;    // x是否大于10。r1得值true
boolean r2 = x <= 100;   // x是否小于等于100。r2得值false
```

➤ 逻辑表达式：用逻辑运算符形成的表达式，计算结果为真假值。与C语言不同，**在Java的逻辑表达式中，非1不会自动当成true。**

```
int y = 99;
boolean r3 = (y > 10) && (y < 100); // y是否大于10并且小于100. true
```

➤ 位表达式：用位运算符和移位运算符形成的表达式，计算结果为整数(int)。

```
int z = 16;
int r4 = z | 0x1E0F; // 按位或。0x1E1F (7711)
```

算术运算符: $a+b$ $a-b$ $a*b$ a/b (商) $a\%b$ (余数) i/j (整除,i和j为整数)
 关系运算符: $a>b$ $a<b$ $a>=b$ $a<=b$ $a==b$ (等于) $a!=b$ (不等于)
 逻辑运算符: $a\&\&b$ (短路与, a为false时不计算b) $a||b$ (短路或) $!a$ (非)
 位运算: $\sim a$ (按位非) $a\&b$ (按位与) $a|b$ (按位或) a^b (按位异或)
 移位运算: $b<<1$ (左移1位) $a>>2$ (带符号右移2位) $b>>>3$ (无符号右移3位)
 三目运算: $x<3?10:7$ (如果x小于3, 则取值10, 否则, 取值7)
 单目运算: $++x$ (x先加1, 再参与运算) $--x$ $x++$ $x--$ $-x$ (变符号)
 赋值运算: $x+=a$ ($x=x+a$) $x-=a$ $x*=a$ $x/=a$ $x\%=a$ $x\&=a$
 $x|=a$ $x\&=a$ $x|=a$ $x^a=a$ $x>>=a$ $x>>>=a$ $x<<=a$

运算优先级:

高 $[]()$ \rightarrow 单目 $\rightarrow * / \% \rightarrow + - \rightarrow << >> >>>$
 $\rightarrow < > <= >= \rightarrow == != \rightarrow \& \rightarrow ^$
 $\rightarrow | \rightarrow \&\& \rightarrow || \rightarrow$ 三目运算 \rightarrow 复杂赋值 低

类型转换

- 如果赋值语句的变量与值的数据类型不一致，系统会进行隐式类型转换。

```
int x = 100;  
long y = x;
```

- 如果要把取值范围小的数据类型转化为取值更大的数据类型，则要用强制类型转换。

```
long x = 100;  
int y = (int)x;
```

- 强制类型转换要注意出现截断错误。

```
Long i = 65536;  
short j = (short)i; //j赋值为0  
double x = 10.876;
```

- ```
int y = (int)x;
```

 //y赋值为10（取整）。`Math.round()`四舍五入
- 对于移位运算，char、byte和short类型会先变为int类型再进行移位运算，long和int类型直接进行移位。

# Java基本语句

- 赋值语句

|                              |                     |
|------------------------------|---------------------|
| <code>int x;</code>          | //变量定义              |
| <code>x = 20;</code>         | //赋值语句，左边为变量，右边为表达式 |
| <code>float y = 1000;</code> |                     |
| <code>x = (int)y;</code>     | //强制类型转换            |

- 注释语句

|                           |                                                                                    |
|---------------------------|------------------------------------------------------------------------------------|
| <code>//</code>           | 注释一行                                                                               |
| <code>/* ..... */</code>  | 注释若干行                                                                              |
| <code>/** ..... */</code> | 文档注释， 可以用javadoc提取， 产生html文件。Java会自己加入标题， 自动加入public和protected成员。每行开头的星号和空格不会包含进去。 |

## ● 分支控制语句

```
if (逻辑表达式){
 语句1;
 语句2;
 ...
}

if (逻辑表达式){
 语句1;
 语句2;
 ...
}
else{
 语句1;
 语句2;
 ...
}
```

//逻辑表达式取值true或false。

```
if (逻辑表达式1) {
 语句1;
 语句2;
 ...
}
else
 if (逻辑表达式2){
 语句1;
 语句2;
 ...
 }
else{
 语句1;
 语句2;
 ...
}
```

\*条件嵌套: else属于最靠近它的if

```
//Statements.java
String bkcolor;
String item= "table";
if(item.equals("table")){
 bkcolor="white";
}
else{
 bkcolor="black";
}
System.out.println("The back color is" + bkcolor);

int age = 20;
if(age<1)
 System.out.println("婴儿");
else if(age>=1 && age<10)
 System.out.println("儿童");
else if(age>=10 && age<18)
 System.out.println("少年");
else if(age>=18 && age<45)
 System.out.println("青年");
else
 System.out.println("中年或老年");
```



**switch**(整数表达式)

```
{
 case 数值1: 语句1;语句2;...;break;
 case 数值2: 语句1;语句2;...;break;
 ...
 case 数值n: 语句1;语句2;...;break;
 default:语句1;语句2;...
}
```

// 整数表达式的类型为byte, char, short, int。

//Statements.java

```
int cnt = 10;
double x;
switch(cnt){
 case 1: x=5.0;break;
 case 12: x=30.0;break;
 default: x=100.0;
}
System.out.println("x="+x);
```

## ● 循环控制语句

```
for(表达式1;布尔表达式2;表达式3)
```

循环体

```
for(type variable: collection) //foreach语句
```

循环体

```
while(逻辑表达式)
```

循环体

```
do
```

循环体

```
while(逻辑表达式)
```

```
break [标号];
```

```
continue [标号];
```

```
return 表达式;
```

- ✓ 循环体可以是单条语句，也可以是用括号{}括起的语句序列。
- ✓ 表达式1和表达式3为逗号隔开的多条语句，分别用于进入循环执行一次和每次循环都执行一次。
- ✓ break和continue加标号表示跳出加标号的循环语句或跳到加标号的循环语句执行。

```
//Statements.java
```

```
int sum=0;
```

```
for(int i=0;i<=100;i++){
```

```
 sum=sum+i;
```

```
}
```

```
System.out.println("sum1(1~100)="+sum);
```

```
double sum=0;
```

```
cnt = 0;
```

```
double scores[]={100.0, 90.2, 80.0, 78.0,93.5};
```

```
for(double score:scores){
```

```
 sum=sum+score;
```

```
 cnt++;
```

```
}
```

```
System.out.println("avg score="+sum/cnt);
```

```
sum=0;
```

```
int k=0;
```

```
while(k<=100){
```

```
 sum=sum+k;
```

```
 k++;
```

```
}
```

```
System.out.println("sum2(1~100)="+sum);
```

```

sum=0;
k=0;
do{
 sum=sum+k;
 k++;
}while(k<=100);
System.out.println("sum3(1~100)="+sum);

```

/\* 求距阵之和

\* 1 2 3 ... 10

\* 1 2 3 ... 10

\* .....  
 \*

\* 1 2 3 ... 10

\*/

```

sum=0;
for(int i=1;i<=10;i++){
 for(int j=1;j<=10;j++){
 sum=sum+j;
 }
}
System.out.println("triangle1="+sum);

```

//块定义域

int k=0;

...

{

int n =5;

int k =5; //错误!!

}

```

sum=0;
Label1:
for(int i=1;i<=10;i++){
 for(int j=1;j<=10;j++){
 if(j==i){
 continue; //跳到for结束处继续执行
 }
 sum=sum+j; //除去对角线的矩阵之和
 } ← continue跳到这里
}
System.out.println("triangle2="+sum);

```

```

sum=0;
Label2:
for(int i=1;i<=10;i++){
 for(int j=1;j<=10;j++){
 if(j==i){
 break; //跳出for循环继续执行
 }
 sum=sum+j; //下三角加对角线矩阵之和
 }
} ← break跳到这里
System.out.println("triangle3="+sum);

```

# 数组

- 数组也是一个对象(Arrays)，用于存储一系列相同类型的数据。数组的优点是存储和访问效率高，缺点是不能改变元素个数。初始化数组后，如果数组元素为基本数据类型，则自动取默认值，否则取值null。

```
// ArrayDef.java
import java.util.Arrays;
int sample[]; // 定义数组对象（未初始化，不能使用）
sample = new int[8]; // 初始化数组，分配8个元素的存储空间。
sample[7]=100; // 数组引用方法
System.out.println(sample[7]); // 显示第7个元素：100。下标从0开始
System.out.println(sample[0]); // 显示：0（默认值）。
int rnds[];
rnds = new int[]{1,3,4,5,6};
System.out.println(Arrays.toString(rnds)); //显示： [1,3,4,5,6]
char[] chars = {'我', '是', '中', '大', '人'}; //初始化一维字符数组
System.out.println(chars[3]); // 显示第4个的字符：和。
String[] s1= {"John", "Wade", "James"}; // 初始化一维字符串数组
System.out.println(s1[1]); // 显示从第1个字符串：Wade
```

```
int nums[] = {9, -10, 18, -978, 9, 287, 49, 7};
for(int num:nums){ // 枚举循环法
 System.out.println(num); // 显示数组nums的全部元素
}
double map[][] = new double[3][10]; // 定义二维数组: 3行10列
map[0][9] = 20;
System.out.println(map[0][9]);
```

// Java只有一维数组，二维数组为数组的数组，所以数组的每行的列数是可变的。

```
int table[][] = {{1},{2,3,4},{5,6,7,8}}; // 二维数组（可变长）
for(int i=0; i<table.length; i++){ // table.length为行数
 for(int j=0; j<table[i].length; j++){ // 处理每行的元素
 System.out.println(table[i][j]); // 显示第i行第j列的元素
 }
}
```

//下面一段程序表示什么意思？

```
int table1[][] = new int[10][];
for(int j=0; j<table1.length; j++){
 table1[j] = new int[j+1];
}
```

```
// ArrayOp.java
import java.util.Arrays; //导入数组类
char s1[]={ 'H', 'e', 'l', 'l', 'o' };
s1=Arrays.copyOf(s1,8); // 复制出一个8元素数组:Hello*** *为null字符
System.out.println(s1); // Hello***
char s2[];
s2=Arrays.copyOf(s1,3); // 复制: s2得到一个3元素数组:Hel
char s3[]=Arrays.copyOfRange(s1, 1, 3); // 复制: s3得到el

Arrays.fill(s2, 'a'); // 把s2的全部元素填充为a
System.out.println(s2); // 结果:aaa
Arrays.fill(s3,2,5, 'o'); // 把s1的第2~4个元素填充为o
System.out.println(s3); // 结果:Heoo
boolean r = Arrays.equals(s1,s2); //比较元素个数和值是否都相等:false
System.out.println(r);
```



```
int pos=Arrays.binarySearch(s1,'1'); // （二分）查找值为1的元素
System.out.println(pos);
Arrays.sort(s1); // 排序s1:***Hello *为null
System.out.println(s1);
int a[]={3,5,4,26,19,2,9};
Arrays.sort(a,1,5); // 排序第1~4个元素:3,4,5,19,26,2,9
for(int x:a){
 System.out.println(x);
}
```

\* `binarySearch()`:使用二分搜索算法来搜索指定的 `int` 型数组，以获得指定的值。**必须在进行此调用之前对数组进行排序**（通过上面的 `sort` 方法）。如果没有对数组进行排序，则结果是不明确的。如果数组包含多个带有指定值的元素，则无法保证找到的是哪一个。

# 字符串

字符串类型(String)为一个用于文字操作的类，内部采用Unicode编码。

```
// StringDef.java
import java.util.Arrays;
char c1[] = {'a', 'b', 'c', 'd', 'e'};
String s1 = "Hello";
String s2 = new String("World");
String s3 = new String(c1);
String s4 = new String(c1, 1, 3);
String s5[] = {"This", "is", "a", "test."}; // 字符串数组
String s6 = s1.concat(s2);
String s7 = s1 + s2;
boolean b1 = s1.equals(s2); // s1和s2 (内容)是否相同
boolean b2 = s1.equalsIgnoreCase("hello"); // 相等比较，忽略大小写.true
boolean b3 = (s1 == s2); // s1和s2是否为同一个对象
boolean b4 = s1.isEmpty(); // 是否为空串。与equals("")相同
int len = s1.length(); // 字符串长度。 5
String s8 = Arrays.toString(s5); // 把数组元素变为逗号隔开的字符串
```

```
// StringOp.java
```

```
import java.util.Arrays;
import java.util.regex.*;
```

```
String s1 = "Hello";
char c1[]={ 'W', 'o', 'r', 'l', 'd' };
String s2 = String.valueOf(c1);
```

```
String s3 = s1.toUpperCase();
String s4 = s1.toLowerCase();
String s5 = s1.substring(0,4);
String s5a= s1.substring(2);
char c2 = s1.charAt(4);
String[] s6 = s1.split("e");
String s7 = s1.replace("l","L");
String s8 = s1.replaceAll("l","L");
String s9 = s1.replaceFirst("l","L");
String s8a = s1.replaceAll("[Hl]","L");
String s9a = s1.replaceFirst("[Hl]","L");
String s10 = " We learn Java";
boolean b1=s10.endsWith("Java");
boolean b2=s10.startsWith("We");
String s11=s10.trim();
```

**s1.codePointAt(i)**取回s1的第i个字符的unicode码(int类型)。

//把字符数组变为字符串:World

//变大写字母:HELLO

//变小写字母:hello

//第0到3个字符的子串:Hell

//第2个字符开始的子串:llo

//取第4个字符:o. **codePointAt()**

//分割字符串: "H","llo"

//替换字符串(所有):heLLO。

//替换字符串(所有):heLLO

//替换第一次出现:heLlo

//替换所有字符H或l:LeLLO

//替换第一次出现字符H或l:LeLlo

// 以什么子串结尾:true

// 以什么子串开始:false

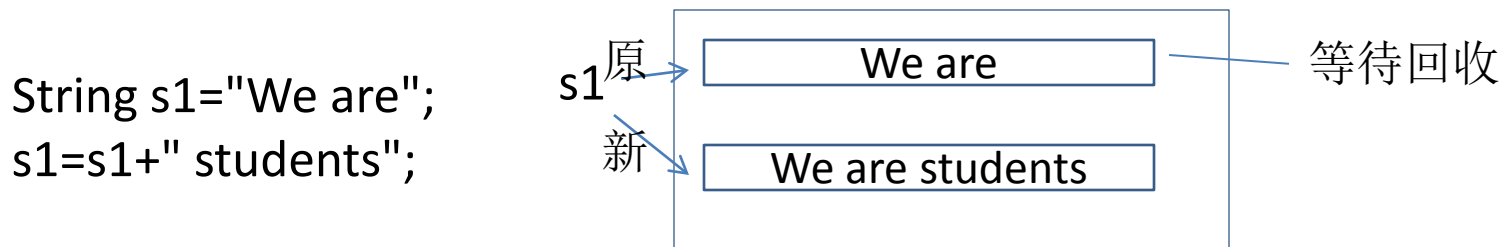
// 删除头尾空格:We learn Java

```

int i=25;
String s12=String.valueOf(i); // 把整数转换为字符串
boolean b3=s10.contains("learn"); // 是否包含子串learn: true
int pos1 = s11.indexOf("e"); // 匹配子串的索引:1.未找到返回-1.
int pos2 = s11.indexOf("e",3); // 匹配子串的索引:4。从位置3开始
int pos3 = s11.lastIndexOf("e"); // 从尾部往前一个匹配子串的索引:4
s12=String.format("%05d,%s",501,"loop");//格式化字符串: 00501,loop
String regex = "(;|,)"; //正则表达式: ;或,
String[] s14= "a;b,c;d".split(regex); //以;或,拆分字:"a","b","c","d"
boolean b4=s11.matches("^We.*") ; //匹配正则表达式(以We开头):true.
int n1 = "abcd".compareTo("abcD"); //词典序:32 0等于<0小于>0大于
int n2 = "abcd".compareToIgnoreCase("abcD"); //忽略大小写
byte[] bytes = s11.getBytes(); // 把字符串转变为字节数组
s15 = new String(s11.getBytes("ISO-8859-1"),"UTF-8");
 //把字符串转换成UTF-8编码

```

**String 是不可变的对象**，每次对 String 类型的对象内容进行改变的时候其实都等同于生成了一个新的 String 对象，然后指向新的 String 对象，原来的字符串将不再使用，并由垃圾回收器自动清理它们。所以经常改变内容的字符串最好不要用 String，而是使用 StringBuffer 和 StringBuilder。



如果需要对字符串频繁操作，可以采用 **StringBuilder** 和 **StringBuffer** 提高效率。

**StringBuffer** 和 **StringBuilder** (java.lang.\*) 都是通过缓冲区操作字符串的对象。它们都不需要重新生成字符串。主要操作有 **append** 和 **insert** 方法：

```
StringBuffer s20=new StringBuffer("uv");
s20.append("xyz"); // 并入末尾。uvxyz
s20.insert(3,"w"); // 插入到中间。uvwxyz
String s21=s20.toString(); // 取出s20的内容
```

**StringBulider** 用于单线程环境。**StringBuffer** 对方法有同步机制，可以用于多线程环境。对于单线程编程，**StringBulider** 比 **StringBuffer** 更有效率。

## StringBuffer和StringBuilder对象的常用方法:

```
append(String str) // 数值变量等类型也可以作为参数
insert(int offset, String str) // 第二个参数和append一样
delete(int start, int end) // 删除一个子串
indexOf(String str, int fromIndex) // 从fromIndex开始查找一个子串
replace(int start, int end, String str) // 从start到end替换一个子串
substring(int start, int end)
reverse() // 字符反转。"abc"=>"cba"
length()
toString() // 取出缓存的字符串
```

具体使用例子见StringBufferOp.java

# 数值与字符串之间转换

```
// DataConversion.java
```

```
int i = 123456;
```

```
println(""+i);
```

```
println(Integer.toString(i));
```

```
println(Integer.toBinaryString(i));
```

```
println(Integer.toHexString(i));
```

```
println(Integer.toString(i, 16));
```

```
println(Integer.MAX_VALUE);
```

```
int j=Integer.reverse(i);
```

```
println(j);
```

```
println(String.format("%08d", i));
```

```
String s1="567890";
```

```
i = Integer.parseInt(s1);
```

```
i = Integer.parseInt(s1,16);
```

```
i = Integer.valueOf(s1);
```

```
i = Integer.valueOf(s1,16);
```

```
float f = 1234.0f;
```

```
println(""+f);
```

```
println(Float.toString(f));
```

```
println(String.format("%010.3f",f));
```

```
f = Float.valueOf("4567.789");
```

```
// 把i转化为字符串
```

```
// 把i转化为字符串
```

```
// 转化为二进制字符串:11110001001000000
```

```
// 转化为十六进制字符串: 1e240
```

```
// 转化为十六进制: 1e240
```

```
// 最大整数: 2147483647
```

```
// i的二进制(32位)的逆转:38240256
```

```
// 逆转: 10010001111000000000000000
```

```
// 按格式转换: 00123456(不够8位前面添0)
```

```
//把字符串s1转换为整数
```

```
// 转换为16进制整数
```

```
//把字符串s1转换为整数
```

```
//转换为16进制整数
```

```
//把浮点数s1转换为字符串
```

```
//把浮点数s1转换为字符串
```

```
//按格式转换: 001234.568(共10位)
```

```
//把字符串转换为浮点数
```

# 日期和字符串之间的转换

```
// DataConversion.java
import java.util.*; //格式化时间: Date Calendar
import java.text.*; //格式化时间: DateFormat SimpleDateFormat
// 把时间转换为字符串
Date now = new Date();
System.out.println(""+now); //显示: Thu Aug 21 11:36:59 CST 2014
DateFormat sdf1 // 默认为本地语言 (省略第二个参数)
 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss aa E",Locale.ENGLISH); //M hh
System.out.println(sdf1.format(now)); //显示: 2014-08-21 11:42:37 AM Thu

// 把字符串转化为时间
Date date1 = sdf1.parse("2008-07-10 19:20:00 PM FRI");
System.out.println(sdf1.format(date1));

//取出年月日: Calendar.YEAR .MONTH .DAY_OF_MONTH .HOUR .MINUTE
// .SECOND .MILLISECOND .DATE .AM_PM .DAY_OF_YEAR
// .DAY_OF_WEEK .HOUR_OF_DAY .ZONE_OFFSET
Calendar cal=Calendar.getInstance();
cal.setTime(new Date()); //cal设置为当前时间
System.out.println("cal.DAY_OF_MONTH: "+cal.get(Calendar.DAY_OF_MONTH));
System.out.println("cal.DAY_OF_WEEK: "+cal.get(Calendar.DAY_OF_WEEK));
System.out.println("cal.HOUR: "+cal.get(Calendar.HOUR));
System.out.println("cal.HOUR_OF_DAY: "+cal.get(Calendar.HOUR_OF_DAY));
```



```
// 增加日期
Calendar cal1=Calendar.getInstance();
cal1.setTime(now); //cal1 设置为当前时间
cal1.add(Calendar.DAY_OF_YEAR, 20); //增加20天。-20就是减20天
System.out.println("cal.add: "+cal1.get(Calendar.DAY_OF_MONTH));
```

```
// 比较日期
boolean b1=cal.before(cal1);
System.out.println("cal.before: "+b1);
```

```
// 求两个日期之间的天数
DateFormat sdf2 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"); //M hh
Calendar cal2 = Calendar.getInstance();
cal2.setTime(sdf2.parse("2012-09-08 10:10:10"));
Calendar cal3 = Calendar.getInstance();
cal3.setTime(sdf2.parse("2012-09-15 00:00:00"));
System.out.println(""+(cal3.getTimeInMillis()-cal2.getTimeInMillis())/(1000*3600*24));
```

```
long time= System.currentTimeMillis()
```

```
long time= new Date().getTime(); //距离1970年1月1日0点0分0秒的毫秒数
```

# Math

|                              |              |                                  |                           |
|------------------------------|--------------|----------------------------------|---------------------------|
| <code>Math.E</code>          | 常量 2.71828   | <code>Math.pow(n1,n2)</code>     | $n_1$ 的 $n_2$ 次幂          |
| <code>Math.PI</code>         | 常量3.14159    | <code>Math.random()</code>       | 随机数: $\geq 0.0$ 且 $< 1.0$ |
| <code>Math.abs(n)</code>     |              | <code>Math.round(n)</code>       | $n$ 的四舍五入                 |
| <code>Math.acos(n)</code>    | 反余弦          | <code>Math.sin(n)</code>         | 三角正弦                      |
| <code>Math.asin(n)</code>    |              | <code>Math.sinh(n)</code>        | 双曲线正弦                     |
| <code>Math.atan(n)</code>    |              | <code>Math.sqrt(n)</code>        | 正平方根                      |
| <code>Math.cbrt(n)</code>    | 立方根          | <code>Math.tan(n)</code>         | 三角正切                      |
| <code>Math.ceil(n)</code>    | 往上到达的第一个整数   | <code>Math.tanh(n)</code>        | 双曲线余弦                     |
| <code>Math.cos(n)</code>     |              | <code>Math.toDegrees(rad)</code> | 把弧度转换为角度                  |
| <code>Math.cosh(n)</code>    | 双曲线余弦        | <code>Math.toRadians(deg)</code> | 把角度转换为弧度                  |
| <code>Math.exp(n)</code>     | $e$ 的 $n$ 次幂 |                                  |                           |
| <code>Math.floor(n)</code>   | 往下到达的第一个整数   |                                  |                           |
| <code>Math.log(n)</code>     | 自然对数 底数是 $e$ | <code>MathFunc.java</code>       |                           |
| <code>Math.log10(n)</code>   | 底数为 10 的对数   |                                  |                           |
| <code>Math.max(n1,n2)</code> | 取较大的一个       |                                  |                           |
| <code>Math.min(n1,n2)</code> | 取较小的一个       |                                  |                           |

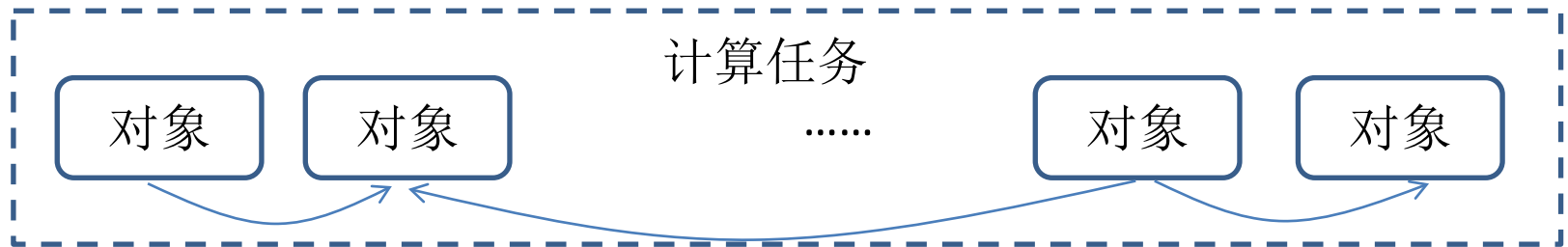
`Integer`和`Math`都是类，为什么可以不通过对象定义而直接使用其方法？

# 控制台输入输出

```
//ConsoleIO.java
import java.util.*;
Scanner in = new Scanner(System.in);
System.out.println("What is your name?(line)");
String name = in.nextLine(); // 输入一行
System.out.println("How old are you?(int)");
int age = in.nextInt(); // 输入一个整数
System.out.println("How much do you weigh?(float)");
float weight = in.nextFloat(); // 输入一个浮点数
System.out.println("name:" + name + " age:" + age + " weight:" + weight);
System.out.println("Input three words:\r\n");
int cnt = 0;
while (in.hasNext() && cnt < 3) { // 是否还有单词
 String word = in.next(); // 读取下一个单词
 System.out.println("'" + cnt + ": " + word);
 cnt++;
}
in = new Scanner("11.0 22.0 33.0 44.0 55.0"); // 直接输入字符串
while (in.hasNextDouble()) { // 是否还有双精度数
 double x = in.nextDouble(); // 读出下一个双精度数
 System.out.println(x);
}
```

# 对象和类

- 要完成一个计算任务必然要涉及很多事物，在面向对象程序设计中把这些事物称为**对象(object)**，并把具有相同属性和操作的对象划分为一类，定义为**类(class)**。
- 与**面向过程程序设计**采用函数调用完成计算功能不同，**面向对象程序设计**是通过对象之间的相互作用来完成计算任务。



- 在Java中，对象的属性和操作被称为**数据域(data field)**和**方法(method)**，也称为**成员变量(数据成员)**和**成员函数**。 如何通过需求设计类？
- 面向对象程序设计方法有哪三个主要特征？  
(1) 封装性      (2) 继承性      (3) 多态性

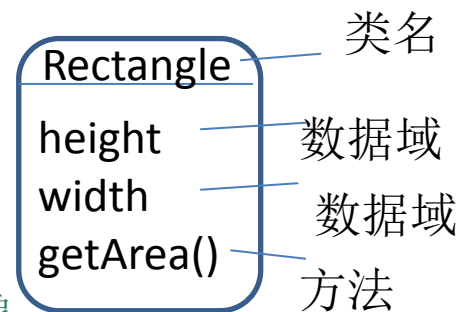
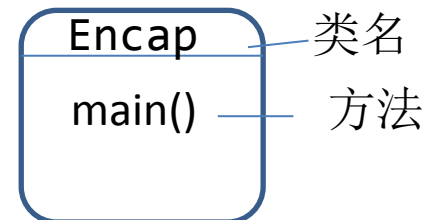
**封装性(Encapsulation)**是指把属性和操作封装为一个整体，使用者不必知道操作实现的细节，只是通过对象提供的操作来使用该对象提供的服务。

类Example1的方法main()调用了类Rectangle的方法getArea()计算面积，但是它不需要知道面积的计算方法。

```
//Encap.java
class Rectangle { //类名
 double height; //数据域(成员变量)
 double width; //数据域(成员变量)
 double getArea(){ //方法(成员函数)
 double area=height*width; //局部变量
 return area; //返回值
 };
}

public class Encap { //类名
 public static void main(String args[]){ //方法
 Rectangle rect = new Rectangle(); //建一个新对象
 rect.height=10; // 数据域赋值
 rect.width=20; // 数据域赋值
 System.out.println(rect.getArea());
 }
}

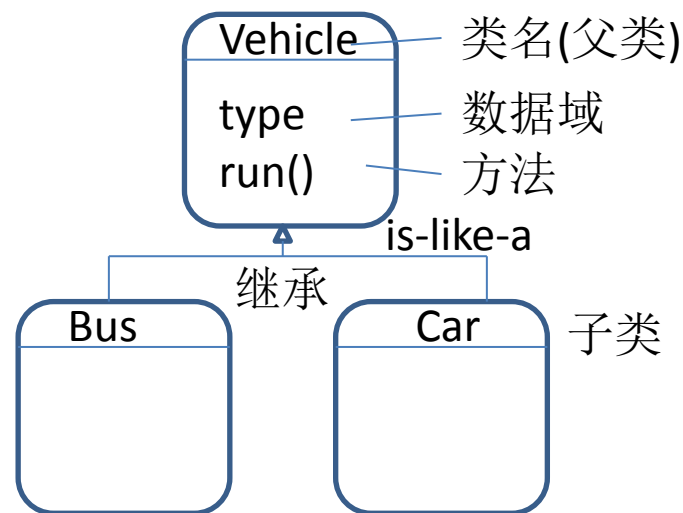
//运行结果： 200.0
```



通过**继承性(Inheritance)**，子类（导出类）可以自动共享父类（积累）的属性和操作。子类与父类之间是is-a或is-like-a的关系。Java的所有基类自动继承内部的Object类。

//Inherit.java getClass和toString是Object的两个方法，getClass会得到当前对象（this）的类名，toString取到对象内容。

```
class Vehicle{ //父类(基类)名
 int type; //数据域
 void drive(){ //方法
 System.out.println("run!" + this.getClass());
 };
}
class Bus extends Vehicle{ //子类(导出类)名 is-like-a
}
class Car extends Vehicle{ //子类名 is-like-a
}
public class Inherit{
 public static void main(String args[]){
 Bus bus = new Bus();
 bus.drive();
 Car car = new Car();
 car.drive();
 }
}
//运行结果: run! class Bus
// run! class Car
```

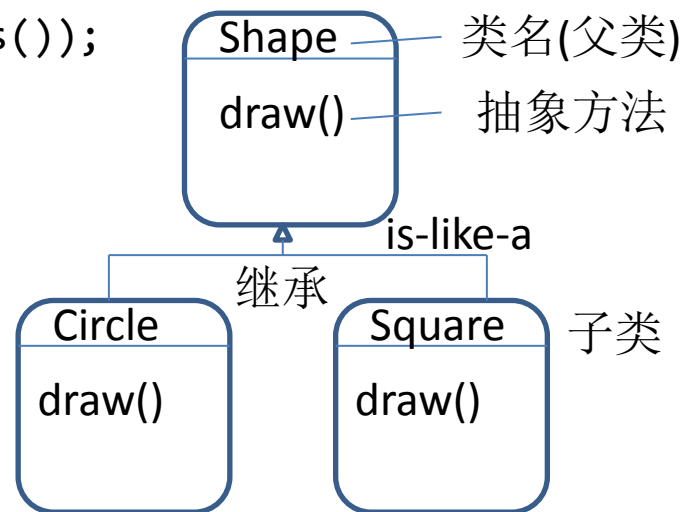


**多态性(Polymorphism)**是指对属于同一类的不同对象采用相同的操作时可以产生完全不同的行为。Shape类的三个对象调用draw()的结果不同。

```
//Poly.java
class Shape{ // 父类(基类)
 int color; // 数据域
 void draw(){ // 方法
 System.out.println("draw!" + this.getClass());
 }
}
class Circle extends Shape{ // 子类(导出类)
 void draw(){ System.out.println("draw!" + this.getClass());};
 // super.draw()可以用于访问父类中的方法
}
class Square extends Shape{
 void draw(){
 System.out.println("draw! " + this.getClass());
 };
}

public class Poly{
 public static void main(String args[]){
 Shape shape1 = new Shape(); shape1.draw();
 Shape shape2 = new Square(); shape2.draw();
 Shape shape3 = new Circle(); shape3.draw();
 }
} // 这里利用了后期绑定。覆盖(override)才是多态。
```

运行结果: draw! class Shape  
draw! class Square  
draw! class Circle



# 包

- 概念

Java中的一切都是对象(类)。类放在java文件中，每个java文件可以包含多个类，但只能包含一个**public**类，编译该文件后每个类会产生一个**.class**文件(字节码文件)。一个大型程序由很多这样的.class文件组成。Java通过子目录来有效地管理它们。同一个子目录下的所有.class文件被认为在同一个包(package)中，还可以把它们形成一个.jar文件(Java文档文件)。jar文件就是Java的类库。

- 定义包

除了注释语句，Java的所有文件可以在第一行用package语句指明所属包的名称，即所在子目录的名称。在子目录com\group\food下的文件的包语句为"**package com.group.food**", 其中，com.group.food 就是包的名称，这里用网站(group.com)及子目录(food)唯一确定一个包。没有定义包的文件属于默认包。



- Cookie.class和Bread.class放在目录c:\java\com\group\food下

```
// Cookie.java
```

```
→ package com.group.food;
public class Cookie {
 public void eat(){
 System.out.println("Eat cookie");
 }
}
```

```
// Bread.java
```

```
→ package com.group.food;
public class Bread {
 public void eat(){
 System.out.print("Eat bread");
 }
}
```

## ● 引用包

- 把语句“**import** com.group.food.\*”加到 Java 文件的前面就可以引用包 com.group.food 中的所有类，也可以指定引入一个类，例如，用语句“import com.group.food.Cookie”引入该包的类 Cookie。引入类 Cookie 之后，就可以采用语句“Cookie cookie=new Cookie();”使用 Cookie 类。
- 如果使用的类在多个引用包中有同名的类，要区分它们就要指明包名：“com.group.food.Cookie cookie=new com.group.food.Cookie();”。

```
// PackEx.java
import com.group.food.*;
public class PackEx{
 public static void main(String[] args){
 Cookie cookie = new Cookie();
 cookie.eat();
 Bread bread = new Bread();
 bread.eat();
 }
}
```

```
// 运行结果: Eat cookie
// Eat bread
```

未指明包的类只能放在根目录下

## ■ Java编译器和虚拟机如何找到类？

- 要找到所引用的类也就是要找到import语句中的子目录。到哪里去找这些子目录呢？这需要用环境变量classpath设置要查找的根目录。
- 例如，`classpath = .;c:\path;c:\pack`就定义了三个根目录，即当前目录（.）、`c:\path`和`c:\pack`。Java编译器和虚拟机只从classpath定义的根目录下或者这些根目录的子目录（由import语句指明）下查找所需要的类。
- 我们也可以把一个目录结构和其中的文件一起打包为一个jar文件，例如：用命令"`jar cvf food.jar *`"可以得到文件`food.jar`（包含子目录信息）。然后在classpath中加入该文件的路径，就可以访问jar文件中的包，例如：`classpath = .;c:\path;c:\pack;c:\java\food.jar`。
- 如果不去设置环境变量classpath，也可以在编译和运行时加入classpath(或cp)参数指出需要的包和根目录：

```
C:>javac -classpath c:\java\PackJar\food.jar;c:\java;. PackJar.java
```

```
C:>java -classpath c:\java\PackJar\food.jar; c:\java;. PackJar
```

# 访问权限

- 类的访问权限

没有修饰词 只能被同一个包的类所访问

**public:** 能被任何类在任何地点所访问

\* 类没有private和protected权限。

- 成员变量和方法的访问权限

**public** 能被任何类的方法在任何地点所访问

**private** 只能被同一个类中的方法所访问。

**protected** 只能被子类 and 同一个包的类中的方法所访问

没有修饰词 只能被同一个包的类中所访问(类似friendly)

\* 为了更好地控制对成员变量的访问，经常把成员变量(例如，**Variable**)设置为**private**，采用**getVariable()**和**setVariable()**的方法取值和设置。

## 数据域和方法的访问权限的另一种表述：

一个类的子类可以直接访问该类的数据域和方法，一个类的非子类可以通过建立该类的对象来访问这个类的数据域和方法，具体的访问权限如下：

对于一个类，和它在同一个包中的子类和非子类可以访问该类所有非**private**的数据域和方法，和它不在同一个包的子类可以访问其**public**和**protected**的数据域和方法，和它不在同一个包的非子类只能访问其**public**的数据域和方法。

|                  | 同一个类 | 同一个包的子类<br>和非子类 | 不同包的子类 | 不同包的非子类 |
|------------------|------|-----------------|--------|---------|
| <b>private</b>   | √    |                 |        |         |
| 无修饰词             | √    | √               |        |         |
| <b>protected</b> | √    | √               | √      |         |
| <b>public</b>    | √    | √               | √      | √       |

# 变量初始化

- 变量的初始化是指给定义的变量赋予初值，可以直接用常量、方法、对象对变量赋值进行初始化。没有初始化的**成员变量**自动获得默认值或null(对象)。在方法中定义的变量称为**局部变量**，Java的局部变量只有在初始化后才有值。

```
// InitDirect.java
import java.util.Random;
class RandNum {
 int x = 10;
 int y = getRand();
 int getRand(){
 Random rnd= new Random(50);
 return rnd.nextInt(100);
 }
}

public class Example6 {
 public static void main(String args[]){
 int i=20;
 RandNum rnd = new RandNum();
 int j = rnd.getRand();
 System.out.format("i=%d j=%d\r\n",i,j);
 System.out.format("x=%d y=%d",rnd.x,rnd.y);
 }
}
```

// 执行结果: i=20 j=17  
// x=10 y=17

// 调用方法对成员变量进行初始化  
// 方法  
// 局部变量rnd初始化50为种子  
// 返回0~99之间的随机数

- 采用构造器(Constructor)进行初始化

```
// InitConst.java
import java.util.Random;
class RandNum {
 int x;
 int y;
 RandNum(int x){
 this.x = x;
 y = getRand();
 }
 int getRand(){
 Random rnd= new Random();// 取值[0,1), 默认为系统时间
 return rnd.nextInt(100); // 返回0~99之间的随机数
 }
}

public class InitConst {
 public static void main(String args[]){
 int i=20;
 RandNum rnd = new RandNum(10);
 int j = rnd.getRand();
 System.out.format("i=%d j=%d\r\n",i,j);
 System.out.format("x=%d y=%d",rnd.x,rnd.y);
 }
}
```

//执行结果: i=20 j=56(随机数)  
// x=10 y=98(随机数)

// this.x指本对象的成员变量x

- 采用块(block)进行初始化

```
// InitBlock.java
import java.util.Random;
class RandNum {
 int x;
 int y;
 {
 //块内可以放置多个语句,在构造器之前执行
 x = 20;
 y = getRand();
 }
 int getRand(){
 Random rnd= new Random();// 取值[0,1), 默认为系统时间
 return rnd.nextInt(100); // 返回0~99之间的随机数
 }
}

public class InitBlock {
 public static void main(String args[]){
 int i=20;
 RandNum rnd = new RandNum();
 int j = rnd.getRand();
 System.out.format("i=%d j=%d\r\n",i,j);
 System.out.format("x=%d y=%d",rnd.x,rnd.y);
 }
}
```

//执行结果: i=20 j=9(随机数)  
// x=10 y=92(随机数)



# enum

enum是类，有自己的方法toString()，其具名值为常量。

```
enum Signal { //定义了enum的导出类Signal（一种整数类型）
 GREEN, YELLOW, RED
}
public class TrafficLight {
 Signal color = Signal.RED;
 public void change() {
 switch (color) {
 case RED:
 color = Signal.GREEN;
 break;
 case YELLOW:
 color = Signal.RED;
 break;
 case GREEN:
 color = Signal.YELLOW;
 break;
 }
 }
}
```

```
public class Test {
 public enum Color {
 RED("红色", 1), GREEN("绿色", 2), BLANK("白色", 3), YELLO("黄色", 4);
 // 成员变量
 private String name;
 private int index;
 // 构造方法
 private Color(String name, int index) {
 this.name = name;
 this.index = index;
 }
 // 覆盖方法
 @Override
 public String toString() {
 return this.index + "_" + this.name;
 }
 }
 public static void main(String[] args) {
 System.out.println(Color.RED.toString());
 }
}
```

<http://www.cnblogs.com/happyPawpaw/archive/2013/04/09/3009553.html>

# 容器类和映射类

- 我们可以利用数组存放相同类型的很多元素，但是有时不知道最终的元素个数，也可能要不断插入和删除元素，或者我们希望每个元素可以用键值进行快速查找，就要用到容器类和映射类。
- 容器类Collection可以存放需要经常增加元素或删除元素的一组元素，其主要子类为Set(无重复无序)和List(有序可重复):
  - (1) HashSet、TreeSet、LinkedSet为三种Set子类。它们分别用Hash表、红黑树和链表作为集合的数据结构，性能不同但是操作类似。
  - (2) LinkedList和ArrayList是两类List子类，分别采用链表和数组实现。
- 映射类Map可以将键映射到值。一个映射所包含的键值不能重复，每个键值只映射到一个值。HashMap和TreeMap是其子类。TreeMap为有序类。
- LinkedList可以作为队列使用。Vector类可以用于多线程环境，Stack类是Vector类的子类。还有其他一些集合类，见下下页。
- Dictionary也是一种将键映射到值的类(已过时)， Hashtable为其子类(已过时)。

## • ArrayList和LinkedList

ArrayList是一个用顺序存储结构实现线性表的类，随机访问速度快，插入删除操作比较慢，而LinkedList用链表实现，插入删除快，随机访问慢。

```
import java.util.*;
public class ArrayListTest {
 public static void main(String[] args) {
 ArrayList<Student> hs = new ArrayList<Student>(); // 泛型
 Student stu1 = new Student(101, "Wang");
 Student stu2 = new Student(102, "Li");
 Student stu3 = new Student(103, "He");
 Student stu4 = new Student(112, "李四");
 hs.add(stu1); // 尾部追加
 hs.add(stu2);
 hs.add(stu3);
 hs.add(1, stu4); // 中间插入
 Iterator<Student> it = hs.iterator(); // 顺序取出
 while (it.hasNext()) {
 System.out.println("***" + it.next());
 }
 System.out.println("+++" + hs.get(2)); // 随机取出
 }
} // <Student>为泛型，指明列表中元素的类型。泛型要求使用类，不能使用基本数据类型。编写的代码可以被不同类型的对象所重用称为泛型程序设计。
```

```

class Student {
 int num; // 学号
 String name; // 姓名
 Student(int num, String name) {
 this.num = num;
 this.name = name;
 }
 public String toString() {
 return num + " " + name;
 }
}

```

其它泛型类:

|                 |                        |
|-----------------|------------------------|
| ArrayDeque      | 一种用循环队列实现的双端队列         |
| EnumSet         | 一种包含枚举类型值的集合类          |
| LinkedHashSet   | 一种可以记住元素插入次序的集合类       |
| PriorityQueue   | 一种允许高效删除最小元素的容器类       |
| EnumMap         | 一种键值属于枚举类型的映射类         |
| LinkedHashMap   | 一种可以记住键值对插入次序的映射类      |
| WeakHashMap     | 一种其值无用武之地时可以被垃圾回收的映射类  |
| IdentityHashMap | 一种用==而不是equals比较键值的映射类 |

## ArrayList和LinkedList的主要方法:

|                                                                              |                                                       |
|------------------------------------------------------------------------------|-------------------------------------------------------|
| <b>boolean</b> add(E e)                                                      | // 将指定的元素添加到列表尾部。E为泛型                                 |
| <b>void</b> add( <b>int</b> index, E e)                                      | // 将指定的元素插入列表指定位置。                                    |
| <b>void</b> clear()                                                          | // 移除列表的所有元素。                                         |
| <b>boolean</b> contains(Object o)                                            | // 如果列表包含指定的元素，则返回 <b>true</b> 。                      |
| E get( <b>int</b> index)                                                     | // 返回列表指定位置上的元素。                                      |
| <b>int</b> indexOf(Object o)                                                 | // 返回列表首次出现的指定元素的索引或 <b>-1</b> 。                      |
| <b>boolean</b> isEmpty()                                                     | // 如果列表为空，则返回 <b>true</b>                             |
| <b>int</b> lastIndexOf(Object o)                                             | // 返回列表最后一次出现指定元素的索引或 <b>-1</b> 。                     |
| E remove( <b>int</b> index)                                                  | // 移除此列表中指定位置上的元素。                                    |
| <b>boolean</b> remove(Object o)                                              | // 移除列表中首次出现的指定元素（如果存在）。                              |
| E set( <b>int</b> index, E e)                                                | // 用指定的元素替代此列表中指定位置上的元素。                              |
| <b>int</b> size()                                                            | // 返回此列表中的元素数。                                        |
| <b>void</b> trimToSize()                                                     | // 将此 <b>ArrayList</b> 实例的容量调整为列表的当前大小                |
| <b>protected void</b> removeRange( <b>int</b> fromIndex, <b>int</b> toIndex) | // 移除 <b>fromIndex</b> 和 <b>toIndex</b> (不包括)之间的所有元素。 |

\* 泛型(**generic type**)E可以使用用户自定义类或标准类(**Integer**, **String**等)

## • HashMap和TreeMap

HashMap和TreeMap是分别用哈希和树结构的方法实现键值映射的类，而TreeMap采用红黑树结构实现，它们的随机访问和插入删除操作速度都很快，HashMap要更快一些，但是内存占用量更大，主要差别是TreeMap的键值可以有序遍历，而HashMap 则不行。

```
// HashMapTest.java
import java.util.*;
public class HashMapTest {
 public static void main(String[] args) {
 HashMap<Integer, Student> map = new HashMap<Integer, Student>();
 Student stu1 = new Student(103, "He");
 Student stu2 = new Student(112, "李四");
 Student stu3 = new Student(101, "Wang");
 map.put(103, stu1); // 加入新的键值对
 map.put(112, stu2);
 map.put(101, stu3);
 Iterator<Integer> it
 = map.keySet().iterator(); // 无序遍历，有序遍历采用TreeMap
 while (it.hasNext()) {
 Integer key = (Integer) it.next();
 Student value = map.get(key); // 根据键值取出值
 System.out.println("***" + value.toString());
 }
 System.out.println("+++" + map.get(112));
 }
}
```

## HashMap的主要方法:

|                                            |                                      |
|--------------------------------------------|--------------------------------------|
| <b>void</b> clear()                        | // 从此映射中移除所有映射关系。                    |
| Object clone()                             | // 返回此 <b>HashMap</b> 实例的副本(不复制键和值)  |
| <b>boolean</b> containsKey(Object key)     | // 是否包含指定键                           |
| <b>boolean</b> containsValue(Object value) | // 是否包含指定值                           |
| V get(Object key)                          | // 返回指定键所映射的值或null。                  |
| <b>boolean</b> isEmpty()                   | // 映射集是否为空。                          |
| Set<K> keySet()                            | // 返回此映射中所包含的键的 <b>Set</b> 视图。       |
| V put(K key, V value)                      | // 在此映射中关联指定值与指定键。                   |
| V remove(Object key)                       | // 从此映射中移除指定键的映射关系(如果存在)             |
| <b>int</b> size()                          | // 返回此映射中的键-值映射关系数。                  |
| Collection<V> values()                     | // 返回此映射所包含的值的 <b>Collection</b> 视图。 |

\* 泛型(generic type)E可以使用用户自定义类或标准类(Integer, String等)



## • Stack类

Stack类继承Vector类。Vector的方法都采用synchronized修饰的，可以用于多线程环境，但是效率比ArrayList低。

```
import java.util.Stack;
public class Hello {
 public static void main(String[] args) {
 Stack<String> stack = new Stack<String>();
 System.out.println("now the stack is empty? " + stack.empty());
 stack.push("a"); // “a”进栈
 stack.push("b");
 stack.push("c");
 stack.push("d");
 stack.push("e");
 System.out.println("now the stack is empty? " + stack.empty());
 System.out.println(stack.peek()); // 得到栈顶元素 e
 System.out.println(stack.pop()); // 退栈，并返回该元素 e
 System.out.println(stack.pop()); // 退栈 d
 System.out.println(stack.search("b")); //返回位置或-1(未找到) 。2
 }
}
```

## • Queue类

java.util.Queue接口扩展了java.util.Collection接口，支持队列的常见操作。Queue使用offer()来加入元素，使用poll()来获取并移出元素。不要采用add()和remove()。因为它们失败的时候会抛出异常。值得注意的是LinkedList类实现了Queue接口，因此我们可以直接把LinkedList当成Queue来用。

```
import java.util.LinkedList;
import java.util.Queue;
public class Hello {
 public static void main(String[] args) {
 //add()和remove()方法在失败的时候会抛出异常(不推荐)
 Queue<String> queue = new LinkedList<String>();
 //添加元素
 queue.offer("a");
 queue.offer("b");
 queue.offer("c");
 for(String q : queue){
 System.out.println(q);
 }
 System.out.println("poll="+queue.poll()); //返回第一个元素，并在队列中删除
 System.out.println("element="+queue.element()); //返回第一个元素
 System.out.println("peek="+queue.peek()); //返回第一个元素
 }
}
```

# Enumeration类

Enumeration类可以用于枚举Hashtable(已过时)的键值。其使用方法如下所示:

```
// EnumTest.java
import java.util.Enumeration;
import java.util.Hashtable;
public class EnumTest {
 public static void main(String[] args) {
 Hashtable<String, String> hs = new Hashtable<String, String>();
 hs.put("123", "Wang");
 hs.put("124", "Li");
 hs.put("125", "Xu");
 Enumeration<String> en = hs.keys();
 while (en.hasMoreElements()) {
 String s1 = (String) en.nextElement();
 System.out.println((String) hs.get(s1));
 }
 }
}
```

在Java中Iterator类为一个接口，用于提供对 collection进行迭代的迭代器，取代了Enumeration类。迭代器与Enumeration类有两点不同：迭代器增加了移除操作，使用了新的方法名。

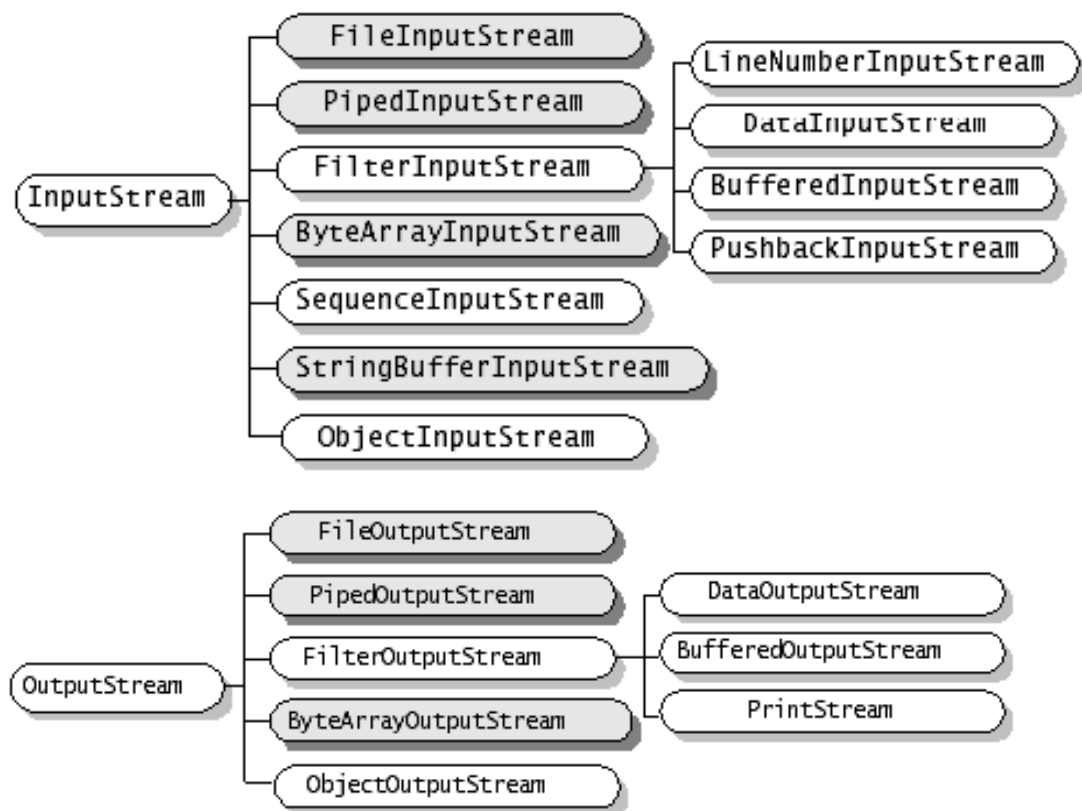
其接口定义如下：

```
public interface Iterator {
 boolean hasNext();
 Object next();
 void remove();
}
```

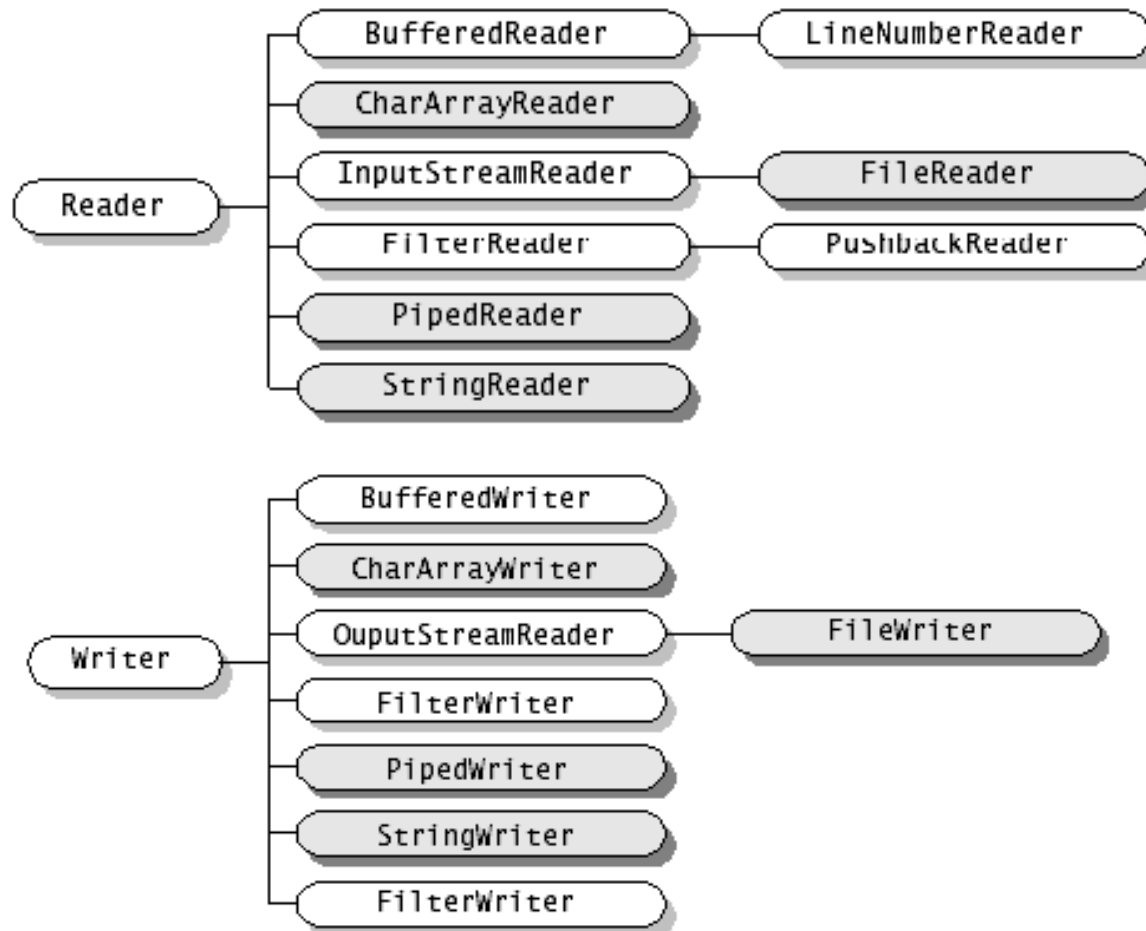
- **Object next():** 返回迭代器刚越过的元素的引用，返回值是Object，需要强制转换成自己需要的类型
- **boolean hasNext():** 判断容器内是否还有可供访问的元素
- **void remove():** 删除迭代器刚越过的元素

# 文件操作

在Java API中，可以从其中读入一个字节序列的对象称为输入流，而可以向其中写入一个字节流的对象称为输出流。这些对象可以是文件、网络、内存块、IO设备等。抽象类InputStream和OutputStream是Java基于字节流输入输出类的基类。



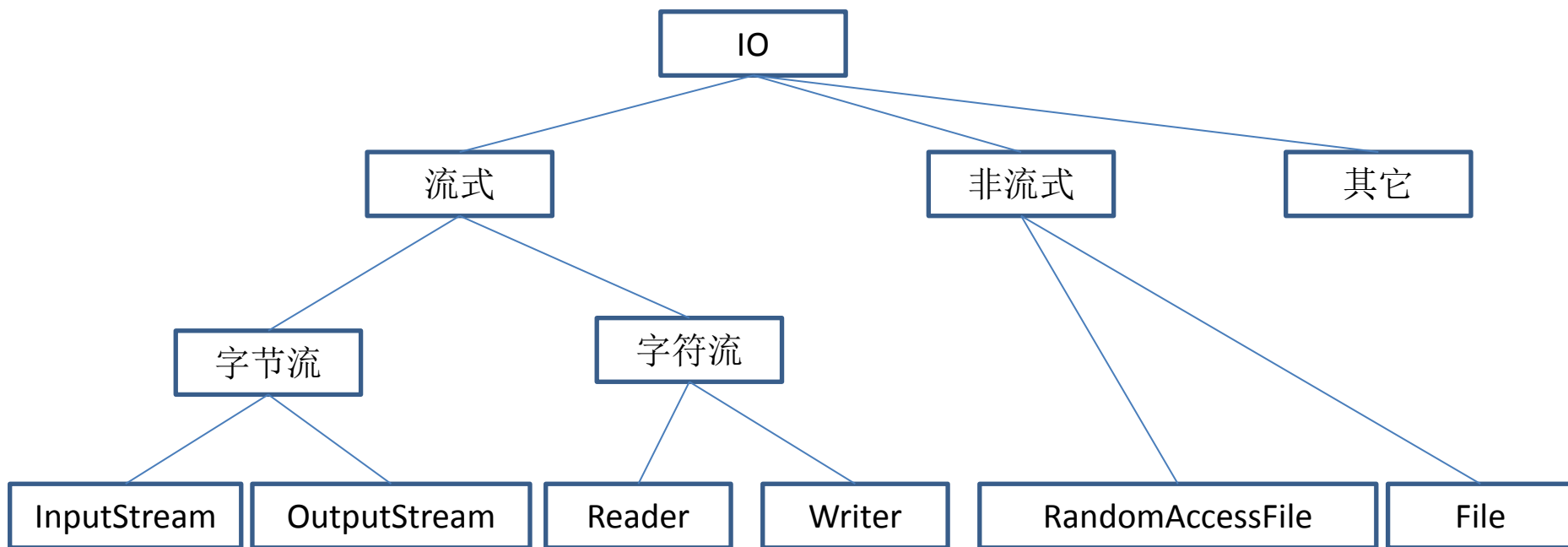
因为面向字节的流不便处理两字节的Unicode码元，抽象类Reader和Writer是Java基于字符流输入输出类的基类。



BufferedReader和BufferedWriter可以对读入字符流进行缓存，提高了读写效率。

Java还包括非流式文件操作，如File类和RandomAccessFile类。File类用于文件或者目录的描述信息，例如生成新的目录，修改文件名，删除文件，判断文件所在路径等。RandomAccessFile类主要用于随机文件操作，可以从文件的任意位置进行存取操作。

还有一些与安全相关的文件操作类，如：SerializablePermission类，以及与本地操作系统相关的文件系统类。



[http://blog.sina.com.cn/s/blog\\_b37338430101e9as.html](http://blog.sina.com.cn/s/blog_b37338430101e9as.html)

<http://api.apkbus.com/reference/java/io/package-summary.html>

## • 二进制文件读写

读写二进制文件是不考虑文件内容，所有文件都是按字节流读写。

`FileInputStream`和`FileOutputStream`是二进制文件读写类，它们的方法列在后面。

### • 一次读入整个文件

```
//CopyFileA.java
// 调用例子：copyBinaryFile("c:\\temp\\home.jpg","c:\\temp\\homeNew1.jpg");
import java.io.*;
// 用in.available()得到文件的大小
public void copyBinaryFile(String srcName,String destName)
 throws Exception {
 FileInputStream in = new FileInputStream(srcName);
 FileOutputStream out = new FileOutputStream(destName);
 byte[] tempbytes = new byte[in.available()];
 in.read(tempbytes);
 out.write(tempbytes);
 in.close();
 out.close();
}
// 一次读入整个文件，效率高，但是占内存大。
```



## (1) FileInputStream类的主要方法:

- int** available() 返回下一次对此输入流调用的方法可以不受阻塞地从此输入流读取（或跳过）的估计剩余字节数。
- void** close() 关闭此文件输入流并释放与此流有关的所有系统资源。
- int** read() 从此输入流中读取一个数据字节。
- int** read(**byte**[] b) 从此输入流中将最多 **b.length** 个字节的数据读入一个 **byte** 数组中。
- int** read(**byte**[] b, **int** off, **int** len) 从此输入流中将最多 **len** 个字节的数据读入一个 **byte** 数组中。
- long** skip(**long** n) 从输入流中跳过并丢弃 **n** 个字节的数据。

## (2) FileOutputStream类的主要方法:

- void** close() 关闭此文件输出流并释放与此流有关的所有系统资源。
- void** write(**byte**[] b) 将 **b.length** 个字节从指定 **byte** 数组写入此文件输出流中。
- void** write(**byte**[] b, **int** off, **int** len) 将指定 **byte** 数组中从偏移量 **off** 开始的 **len** 个字节写入此文件输出流。
- void** write(**int** b) 将指定字节写入此文件输出流。

- 分块拷贝文件

```
// CopyFileB.java
// 调用例子:copyBinaryFile("c:\\temp\\home.jpg","c:\\temp\\homeNew2.jpg");
// 一个文件分多次复制可以减少一次占用太多内存,但是会降低效率。
import java.io.*;

public static void copyBinaryFile(String srcName,String destName)
 throws Exception {
 InputStream in = new FileInputStream(srcName);
 OutputStream out = new FileOutputStream(destName);
 byte[] tempbytes = new byte[100]; int byteread = 0;

 // in.read()返回实际读入的字节数
 while ((byteread = in.read(tempbytes)) != -1) {
 out.write(tempbytes, 0, byteread); // 写入缓冲区、其实位置(偏移量)、字节数
 }
 in.close();
 out.close();
}
```

## • 文本文件

可以指定编码读写文本文件，并可以对读出内容直接进行字符串的查找和匹配。如果采用二进制文件读写，则只能读出字节而不会转化为字符串。

```
// CopyTextFileA.java 用InputStreamReader和OutputStreamWriter进行文件读写
import java.io.*; // 可以控制读写文件的编码
public class CopyTextFileA {
 public static void main(String args[])throws IOException{
 String srcFile="c:\\temp\\config.xml";
 String destFile="c:\\temp\\config1.xml";
 CopyTextFileA cf = new CopyTextFileA();
 cf.copyTextFile(srcFile,destFile);
 System.out.println("Copy finished!");
 }
 public void copyTextFile(String srcName,String destName)
 throws IOException {
 String s1=inputFileStream(srcName);
 outputStream(destName,s1);
 }
}
```

// 读出文本文件（可以指出编码方式），InputStreamReader创建一个使用某种字符  
// 编码的字符输入流对象。这里使用UTF-8编码，如果没有指明，则采用默认编码。

```
public String inputFileStream(String fileName) throws IOException {
 StringBuilder sb = new StringBuilder("");
 int i;

 FileInputStream f1= new FileInputStream(fileName);
 InputStreamReader in = new InputStreamReader(f1, "UTF-8");
 while ((i = in.read()) != -1) {
 sb.append((char) i);
 }
 return sb.toString();
}
```

// 写入文本文件（可以指出编码方式）。 OutputStreamWriter创建一个使用某种字符  
// 编码的字符输出流对象。这里使用UTF-8编码，如果没有指明，则采用默认编码。

```
private boolean outputFileStream(String fileName, String content)
 throws IOException{
 FileOutputStream fos = new FileOutputStream(fileName);
 OutputStreamWriter osw = new OutputStreamWriter(fos, "UTF-8");
 osw.write(content);
 osw.flush();
 osw.close();
 return true;
}
}
```

InputStreamReader类是输入字符流Reader类的子类，其主要方法说明如下：

|                                                                               |                                    |
|-------------------------------------------------------------------------------|------------------------------------|
| <b>void</b> close()                                                           | 关闭本对象(reader).                     |
| <b>String</b> getEncoding()                                                   | 返回输入流编码（文件编码）                      |
| <b>int</b> read()                                                             | 读入一个字符，返回整数（转换为2字节Unicode码，高16位为0） |
| <b>int</b> read( <b>char</b> [] buffer, <b>int</b> offset, <b>int</b> length) | 读入最多length个字符并从offset开始的位置保存在buf中。 |
| <b>boolean</b> ready()                                                        | 指出是否可读(read时不会被阻塞)                 |

OutputStreamReader类是写入字符流Writer类的子类，其主要方法说明如下：

|                                                                                |                                      |
|--------------------------------------------------------------------------------|--------------------------------------|
| <b>void</b> close()                                                            | 关闭本writer.                           |
| <b>void</b> flush()                                                            | 刷新本writer，即保存缓冲区的内容.                 |
| <b>String</b> getEncoding()                                                    | 返回输出流编码（文件编码）                        |
| <b>void</b> write( <b>char</b> [] buffer, <b>int</b> offset, <b>int</b> count) | 把buf中从offset开始的count个字符写入本writer.    |
| <b>void</b> write( <b>String</b> str, <b>int</b> offset, <b>int</b> count)     | 把字符串str中从offset开始的count个字符写入本writer. |
| <b>void</b> write( <b>int</b> oneChar)                                         | 把字符oneChar写入本writer.                 |

```

// CopyTextFileB.java
import java.io.*;
// BufferedReader和BufferWriter按字符流输入和输出。PrintWriter可以采用格式化输出
// (类似String.format())。
public class CopyTextFileB {
 public static void main(String args[])throws Exception {
 String srcFile="c:\\temp\\config.xml";
 String destFile="c:\\temp\\config2.xml";
 copyTextFile(srcFile,destFile);
 System.out.println("Copy finished!");
 }
 public static void copyTextFile(String srcName,String destName)
 throws Exception {
 //采用BufferedWriter写入缓存，避免每次都写入磁盘文件。
 BufferedReader reader = new BufferedReader(new FileReader(srcName));
 // 采用BufferedReader读入缓存，避免每次readLine都去读磁盘文件。
 PrintWriter out = new PrintWriter(
 new BufferedWriter(new FileWriter(destName)));
 String tempString = null;
 while ((tempString = reader.readLine()) != null) {
 out.println(tempString);
 }
 reader.close();
 out.flush();
 out.close();
 }
}

```

<http://375940084.blog.51cto.com/2581965/751040>

## • 文件查找

读取目录中的内容、获取文件和目录的属性。

```
// FileList.java
import java.io.File;
// 显示某个文件夹下两级文件夹的所有文件和文件夹的绝对路径
File root = new File("C:\\temp\\test"); // 取得根文件夹
for (File file: root.listFiles()) { // 取得根文件夹中的所有文件和文件夹
 if (file.isDirectory()) {
 System.out.println("*"+file.getAbsolutePath());
 for (File f: file.listFiles()) {
 System.out.println(f.getAbsolutePath());
 }
 }
 else{
 System.out.println(file.getAbsolutePath());
 }
}

//File类的其它方法:
// String getName() 返回文件或文件夹的名称
// String getPath() 返回文件或文件夹的路径(不含文件名)
// int length() 返回文件大小(字节)
// int lastModified() 返回最后修改时间
// String getParent() 获得双亲文件夹名
// boolean mkdir() 建立文件夹
// boolean delete() 删除文件或空文件夹
```

# 附录1、安装JDK

- 文件：jdk-8u60-windows-i586  
下载地址：<http://www.oracle.com/technetwork/java/javase/downloads/>
- 安装后验证安装成功：

控制台下运行

C:>javac

C:>java

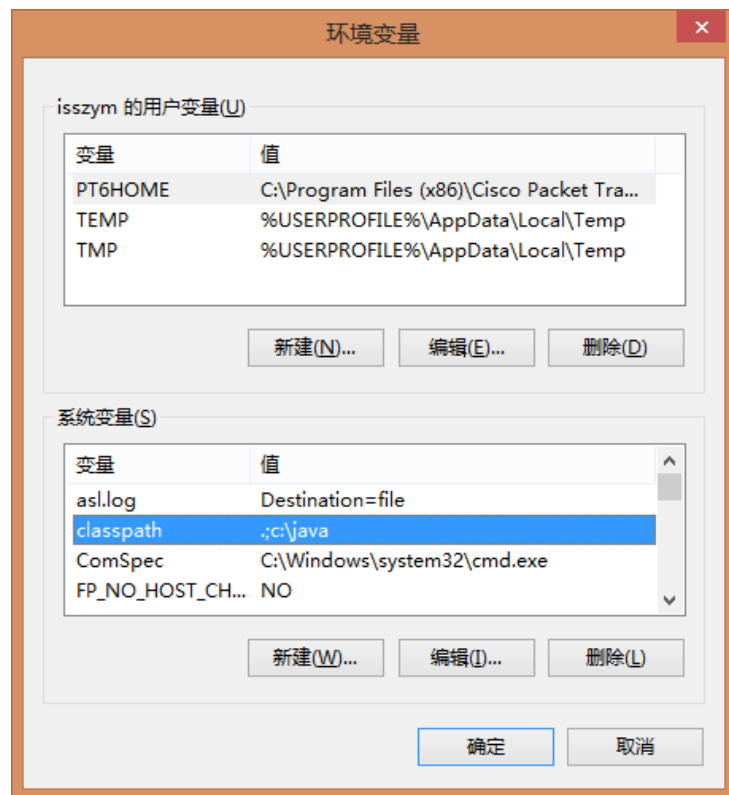
如果显示出帮助信息，则表示安装成功。如果显示错误信息，应该是Java程序目录(例如：windows---C:\Program Files\java\jdk1.7.0\_21\bin)没有加入环境变量PATH中。

编译和执行Java程序还要通过环境变量CLASSPATH设置包查找目录，否则执行字节码文件时可能出现“找不到或无法加载类”的错误。

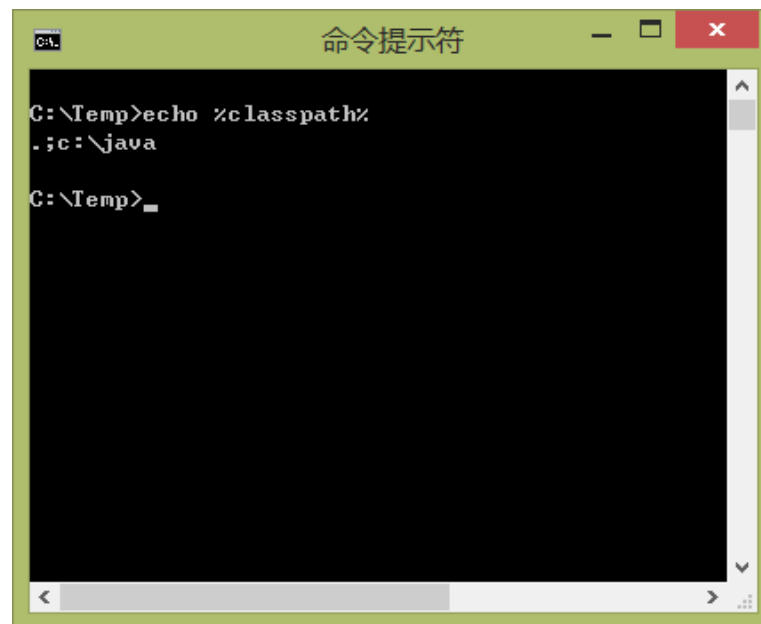


- Window环境变量设置方法:

计算机(我的电脑)/属性/高级系统设置/环境变量/系统变量

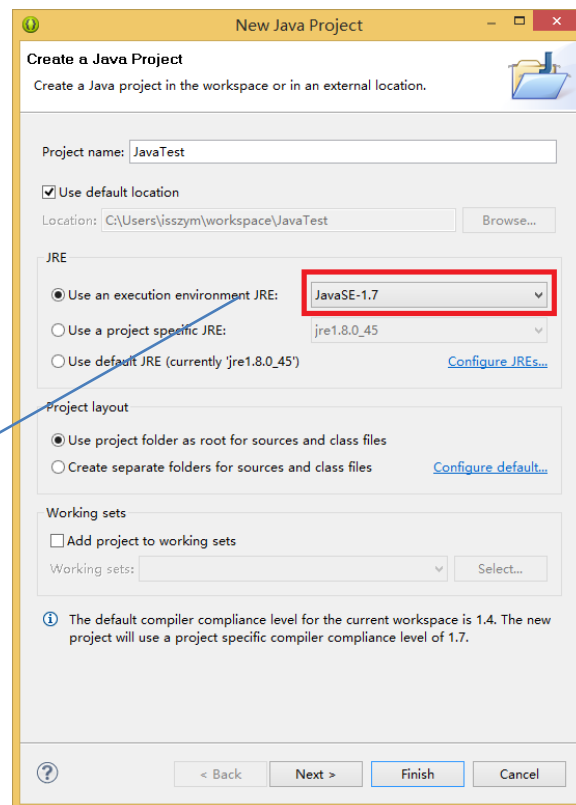
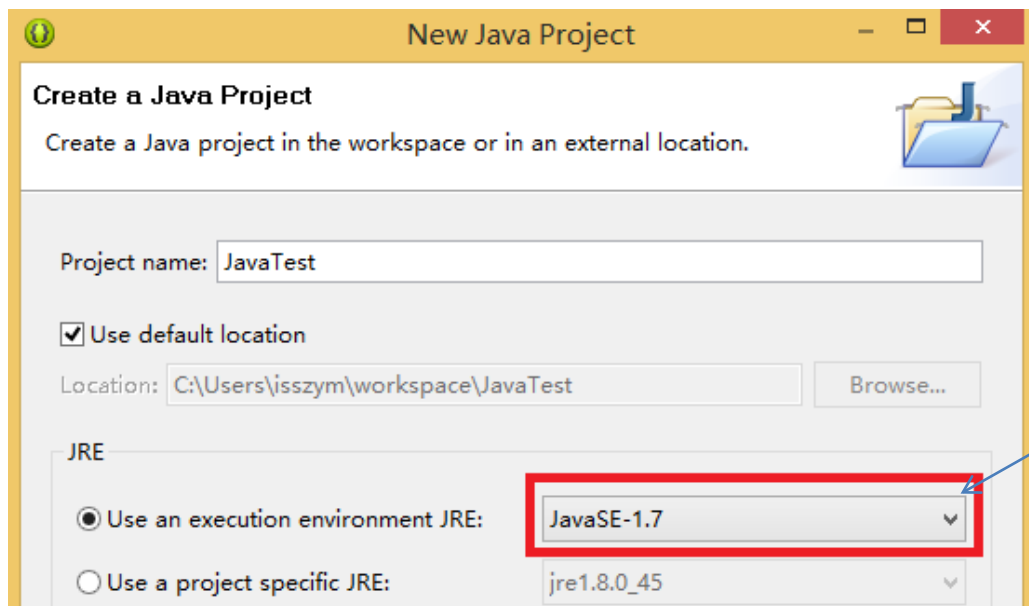


用echo验证是否环境变量设置正确



# 附录2、安装使用Eclipse

- 安装完JRE后，直接解压包“adt-bundle-windows-x86\_64-20130522.zip”，然后运行Eclipse.exe。
- 建立Java Project (菜单：file/new/Java Project)，取好项目名，选择JRE执行环境。



也可以在项目属性中设置：Java BuildPath/Libraries 删除旧版的JRE，增加最新版的JRE。这也是增加其它jar文件到项目的方法。

## ● 运行和调试

- ❑ 然后可以通过项目菜单new/class增加java源程序。
- ❑ 菜单run/run可以直接编译运行，run/debug可以调试运行，run/toggle breakpoint可以用来在源代码中设置和取消断点。
- ❑ 运行到断点会进入debug状态，此时，run/step over（F6）可以逐语句执行，并且不会进入方法中执行，run/step into（F5）会进入方法执行。在调试状态，通过右键点击变量得到菜单watch，可以实时监看变量的值。这些菜单操作都可以在工具栏中找到。

# • Eclipse开发界面

调试运行 直接运行

代码区

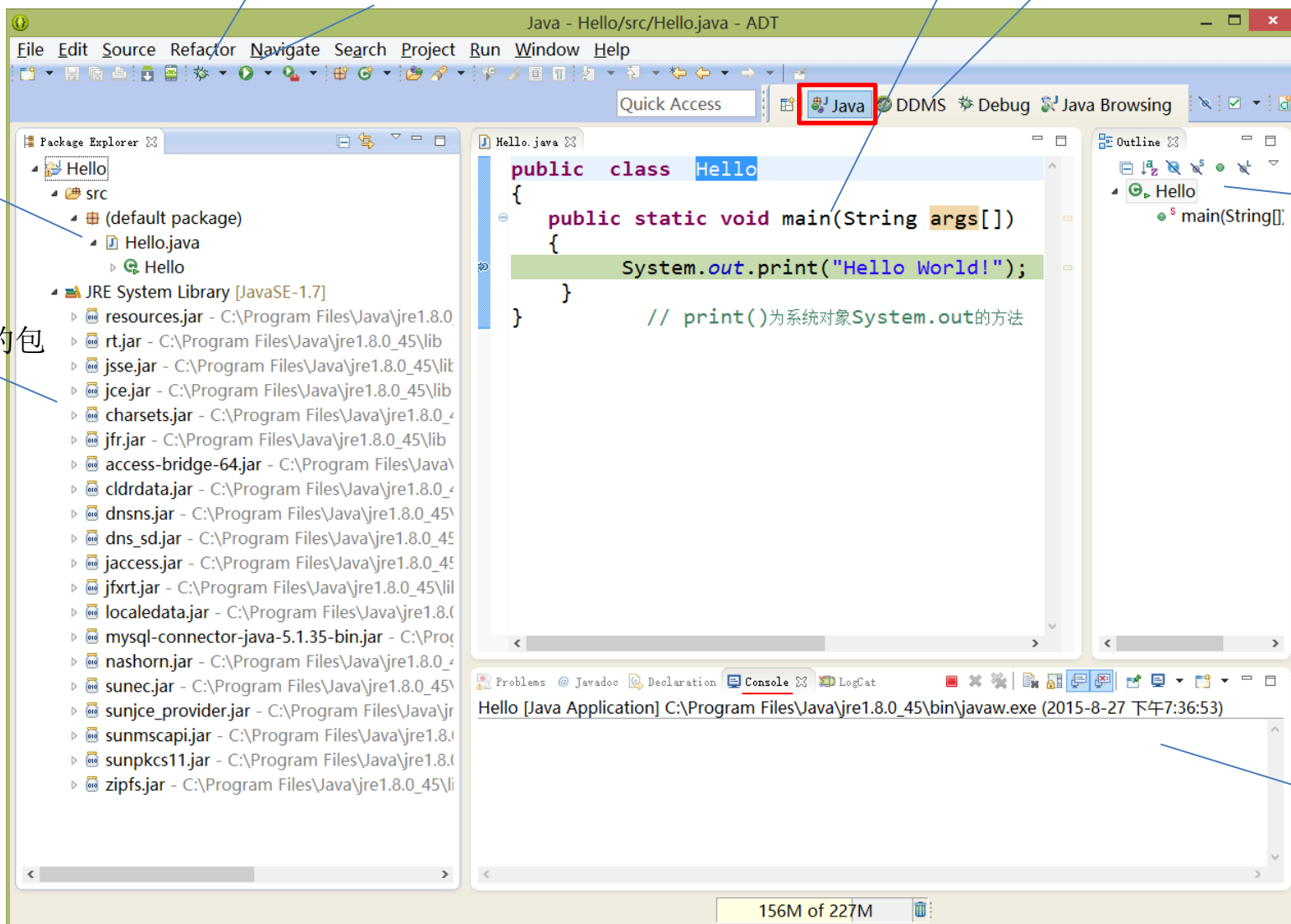
Dalvik Debug Monitor Service  
(安卓调试)

源代码

所引用的包

当前类

控制台  
输入输出



# • 调试界面

step in step over

Watch的变量和值

源码  
当前执行位置  
断点标志

