

# Java程序设计

## (基础)

2020.5.2 v2.3

isszym sysu.edu.cn

[SE8 SE10 apkbus runoob](#)  
[download](#)

# 目录

---

• 概述	...	• 枚举类型	...
• C++编译器和Java虚拟机	...	• 包	...
• Java的特点	...	• 访问权限	...
• JRE和JDK	...	• 集合框架	...
• 第一个Java程序	...	• 附录1、安装JDK	...
• 数据类型	...	• 附录2、程序设计语言	...
• 常量、变量、表达式	...	• 附录3、定义域和生存期	...
• 基本语句	...	• 附录4、命名法	...
• 数组和字符串	...	• 附录5、捕捉错误	...
• 对象和类	...	• 附录6、数值和字符串之间的转换	...
• 重载、覆盖与传址	...	• 附录7、日期和字符串之间的转换	...
• 静态成员	...	• 附录8、Math	...
• 构造器	...	• 附录9、控制台输入输出	...
• 变量初始化	...	• 附录10、参考资料	...
• final关键字	...		

---

# 概述

- 1991 年，Sun Microsystems 的工程师James Gosling、Patrick Naughton、Chris Warth、Ed Frank和Mike Sheridan共同构想了Java语言。
- 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运行，以满足消费类电子设备(洗衣机、空调和微波炉等)的需要。Java刚诞生时提出过一个著名口号：“一次编写，到处运行”(Write Once, Run Anywhere)
- 就在快要设计出Java 的细节的时候，另一个在Java未来扮演关键角色的重要因素出现了。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。
- 在2007年，安卓选用了Java语言，更使Java成为最有前途的语言。

## TIOBE 编程语言指数排行榜（2020 年1 月）

2020年 1月	2019年 1月	程式语言	等级	更改
1	1	Java	16.90%	-0.01%
2	2	C	15.77%	2.44%
3	3	Python	9.70%	1.41%
4	4	C++	5.57%	-2.58%
5	7	C#	5.35%	2.07%
6	5	Visual Basic .NET	5.29%	-1.17%
7	6	JavaScript	2.45%	-0.85%
8	8	PHP	2.41%	-0.28%
9	15	Swift	1.80%	0.61%
10	9	SQL	1.50%	-0.77%
11	18	Ruby	1.06%	-0.03%
12	17	Delphi/Object Pascal	1.00%	-0.10%
13	10	Objective-C	0.93%	-0.85%
14	16	Go	0.90%	-0.22%
15	14	Assembly language	0.88%	-0.32%
16	20	Visual Basic	0.83%	-0.20%
17	25	D	0.83%	0.25%
18	12	R	0.81%	-0.52%
19	13	Perl	0.75%	-0.48%
20	11	MATLAB	0.74%	-0.76%

TIOBE排行榜是根据互联网上有经验的程序员、课程和第三方厂商的数量，并使用搜索引擎（如Google、Bing、Yahoo!）以及Wikipedia、Amazon、YouTube统计出排名数据，只是反映某个编程语言的热门程度。

## GitHub中程序设计语言的活跃度统计（2016.1~2016.12）

JavaScript

Java

Python

Ruby

PHP

C++

CSS

C#

C

Go

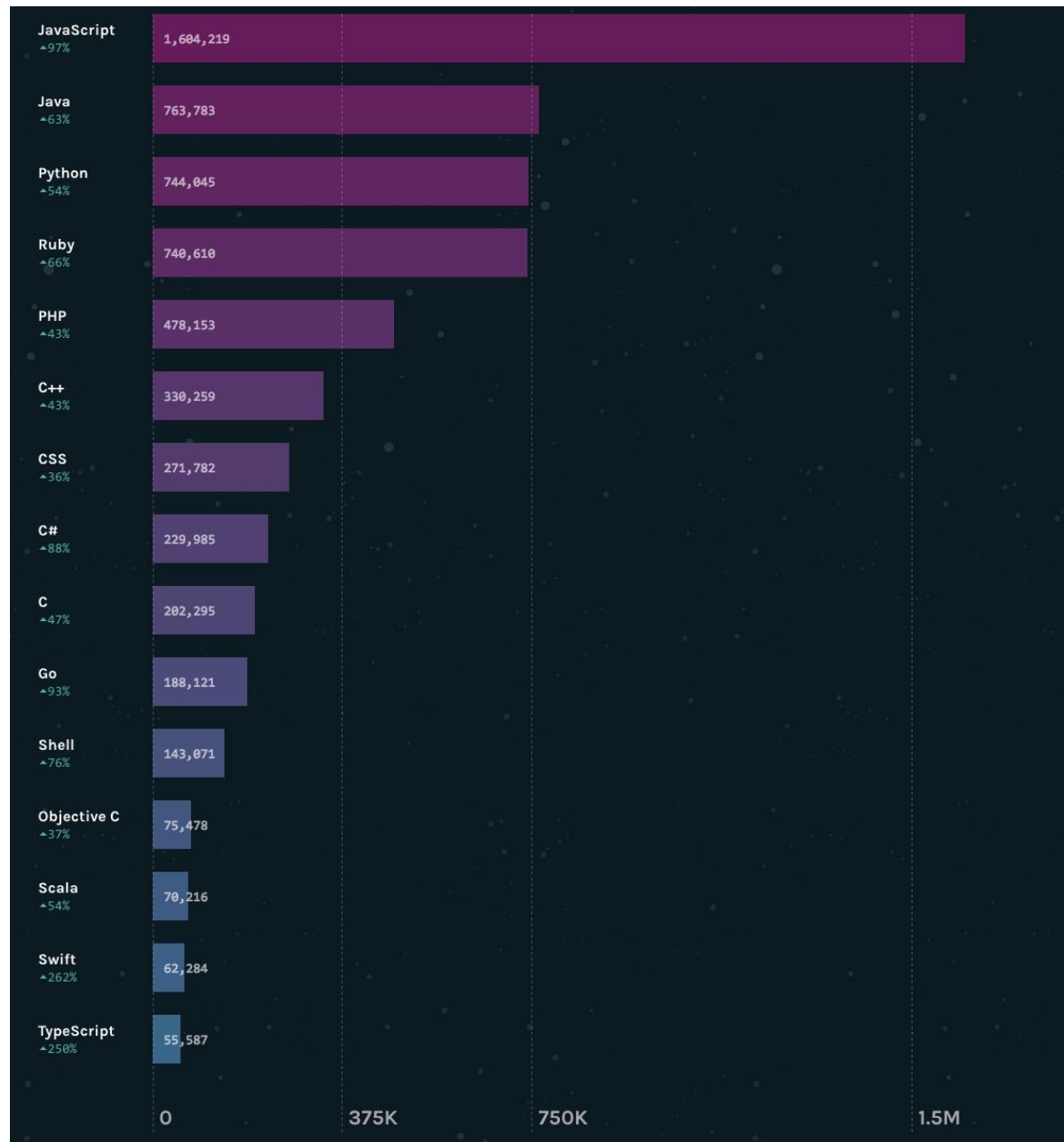
Shell

Objective C

Scala

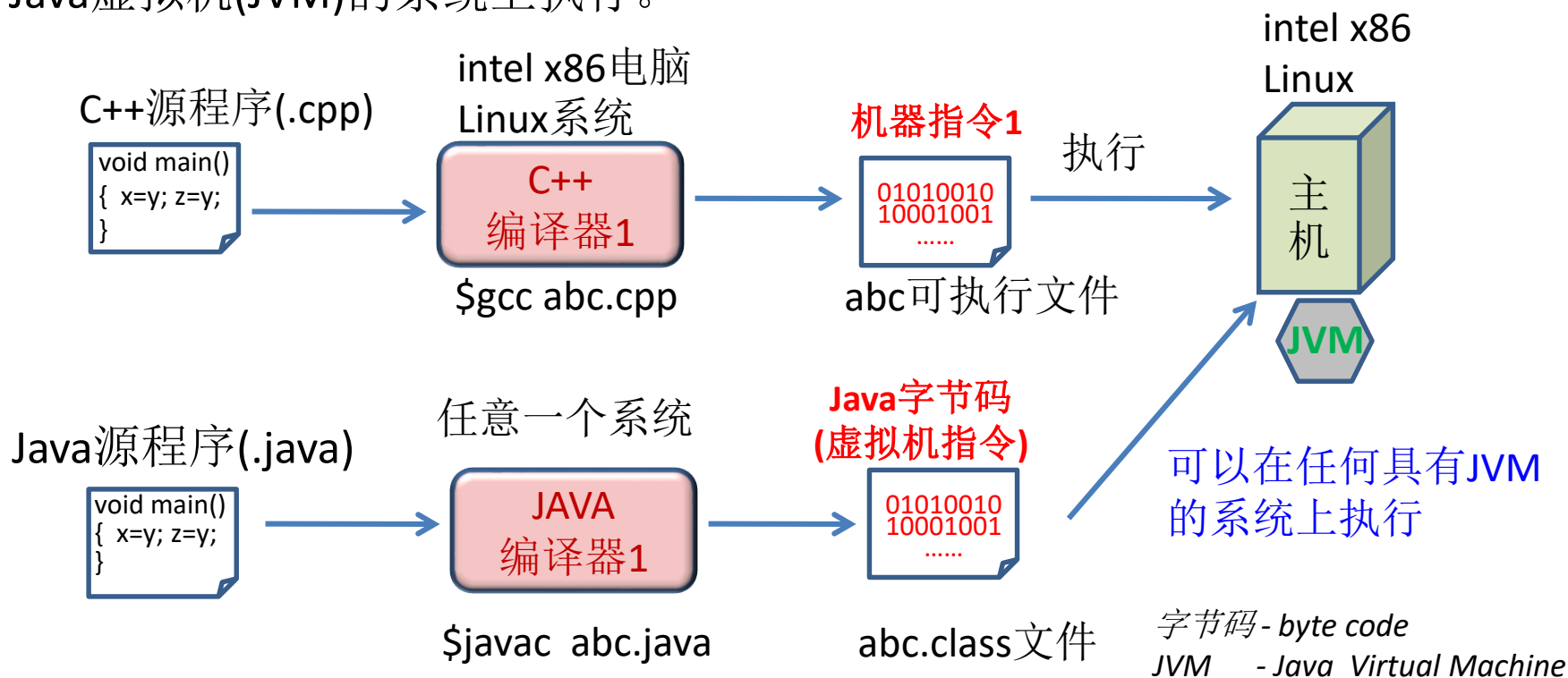
Swift

TypeScript



# C++编译器和Java虚拟机

- 如果C++源程序(.cpp)想在PC机(x86)的Linux系统上运行，则要用其gcc编译成机器指令文件，该文件只能在PC的Linux系统上执行，要在ARM的Linux系统上执行还要重新编译。
- Java源程序(.java) 在编译成Java字节码文件(.class)之后可以在任何安装了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。Java没有**指针类型(pointer)**，避免了内存溢出等安全性问题。Java的**垃圾收集器(garage collection)**自动释放不再使用的存储空间，使程序员解除了常常忘记释放存储空间的烦恼。

## ❑ 易于移植(Portable)

Java支持交叉平台代码，Java字节码(byte code)程序可以在任何具有Java虚拟机的环境中执行，其执行速度被高度优化，其效率有时甚至超过了C++的程序。

## ❑ 面向对象(Object-oriented)

Java语言中一切都是**对象**，并且Java程序带有用于检查 and 解决对象访问的运行时(run-time)类型信息。

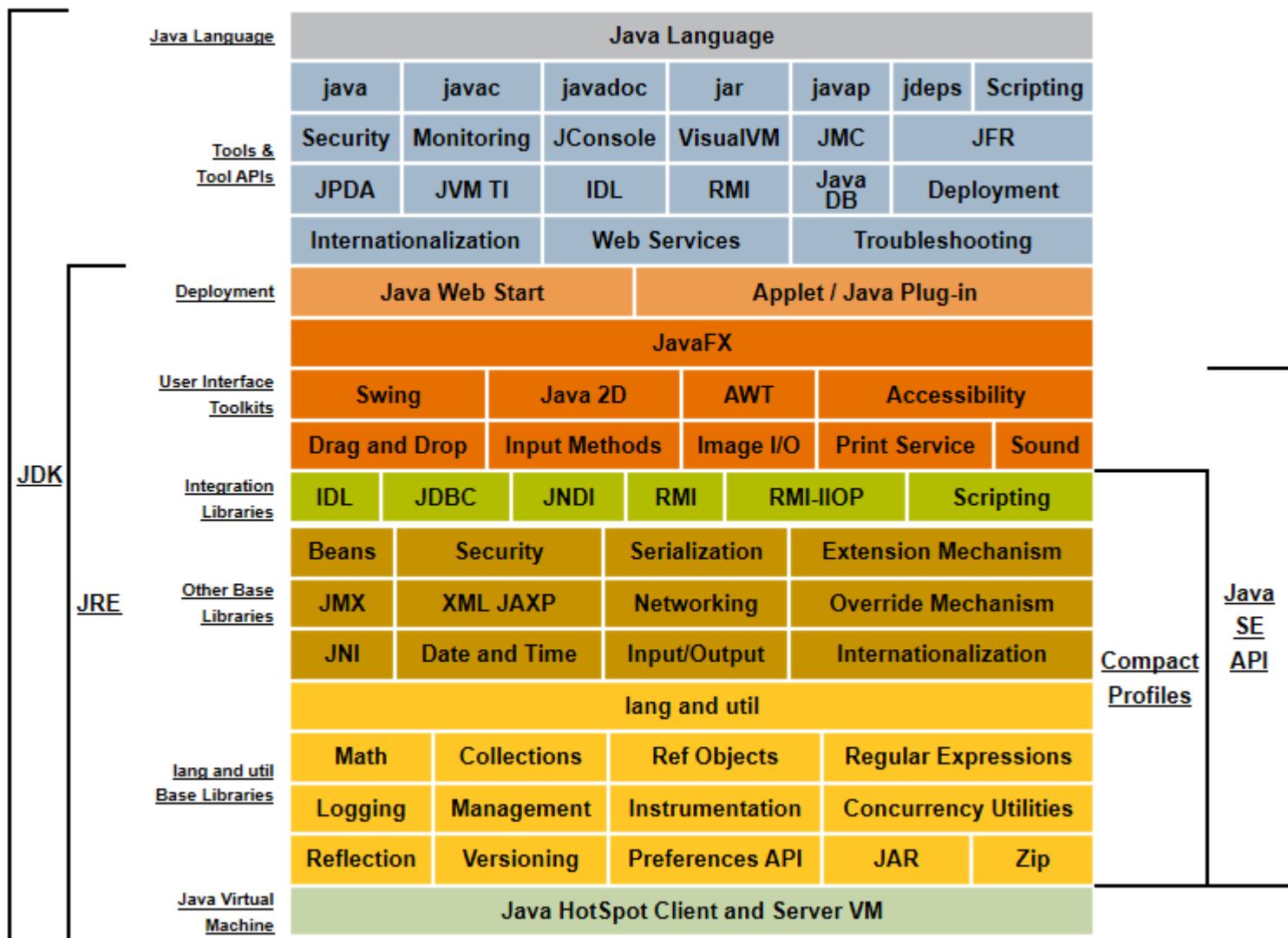
# JRE和JDK

- Java平台有两个主要产品：Java Runtime Environment (JRE) 和Java Development Kit (JDK) 。
- JRE提供Java核心库、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：

- ✓ **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。

\* 下载和安装JDK的方法见附录1 (使用第8版)

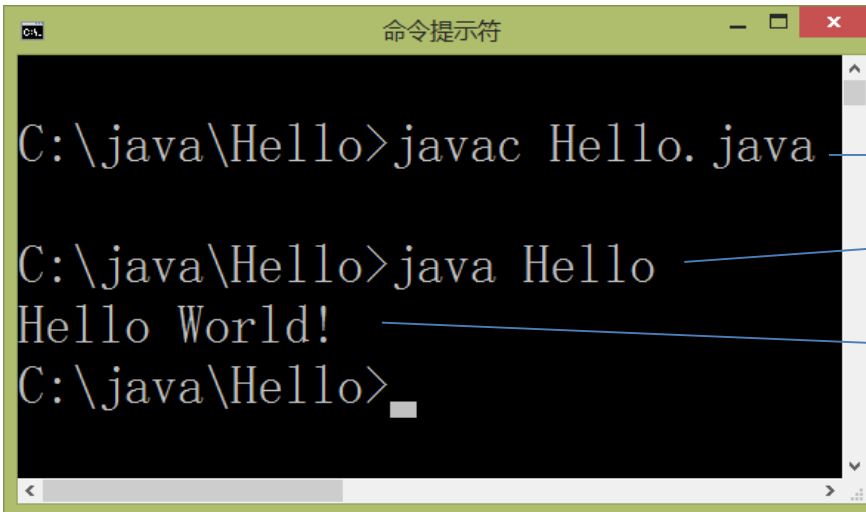




# 第一个Java程序

文件名: Hello.java

```
public class Hello // 类名，要与文件名相同!!!
{
    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

\* `Javac -encoding UTF-8 Hello.java` (UTF-8的源码，默认为ansi编码)

# 数据类型

- 程序的每个数据都要以一定的格式存放，因此需要定义数据类型，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	包装类	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

\* `int x=0; Integer y=1;` // x是基本数据，y是对象，但是它们的使用方法相同。

\* 基本数据类型作为数据域的默认值为上表最后一列。

# 常量

常量(字面量)是指程序中不变的量。我们可以直接把字面量写入语句，但是更好的编程实践是把命名常量用于语句。

布尔常量:	<code>true false</code>	<small>1~9开头</small>	<small>0开头</small>
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</code>	<code>3.14E3F</code> ( $3.14 \times 10^3$ )	-- float
	<code>3.14</code>	<code>2.536D</code>	<code>3.1415926E-3D</code> -- double
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)。		

\*其中的字母不区分大小写

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

# 变量

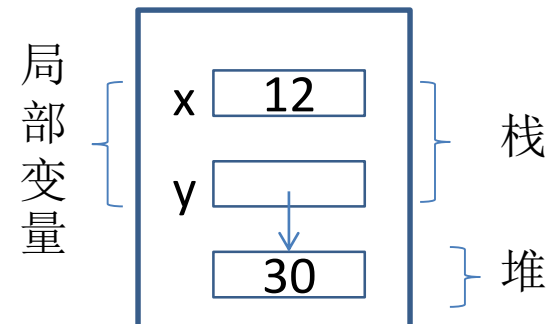
Var.java

- 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**。
- 变量的值可以通过赋值语句进行改变。赋值语句用等于号连接变量和表达式。表达式计算出来的值(右值)用于修改变量的值(左值)。
- 我们首先要为变量命名。**Java变量的命名规则**：由\$、字母、下划线和数字组成的符号串，第一个符号不能是数字。
- 为了分配内存和计算，变量还需要定义数据类型。

```
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
```

- 系统的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
long s = 40;
Long t = 50L;   //要用Long常量
```



# 运算符与表达式

- **算术表达式**: 用算术运算符形成的表达式, 计算结果为数值

```
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

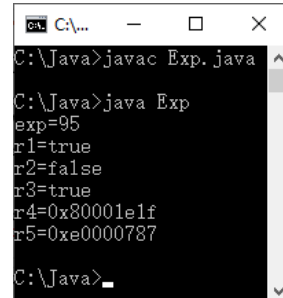
```
int x = 300;
boolean r1 = x > 10;           // x是否大于10。r1得值true
boolean r2 = x <= 100;        // x是否小于等于100。r2得值false
```

- **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0不会自动当成true, 会出错。**

```
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**: 位运算符和移位运算符, 计算结果为整数(int)。

```
int z = 16;
int r4 = z | 0x80001E0F; //按位或      结果 0x80001E1F
int r5 = r4 >> 2;       //有符号右移2位 结果 0xE0000787
```



```
C:\Java>javac Exp.java
C:\Java>java Exp
exp=95
r1=true
r2=false
r3=true
r4=0x80001e1f
r5=0xe0000787
C:\Java>
```

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

\* Java有 $==$  (类型可以不同但(转换后的)值相等) 而没有 $===$  (类型相同且值相等), JavaScript都有

# 类型转换

Cast.java

- 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。

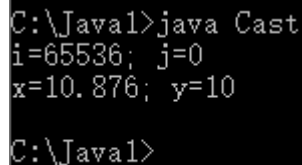
```
int x = 100;  
long y = x;
```

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

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

- 强制类型转换会出现截断错误。

```
long i = 65536;  
short j = (short)i; // j被赋值为0  
double x = 10.876;  
int y = (int)x; // y赋值为10（取整）
```



```
C:\Java1>java Cast  
i=65536; j=0  
x=10.876; y=10  
  
C:\Java1>
```

- 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

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

- 注释语句

```
//           注释一行
/* ..... */  注释若干行
/** ..... */  文档注释，可以执行javadoc提取出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3;  
    语句4;  
    ...  
}
```

//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3;  
    语句4;  
    ...  
}  
else{  
    语句5;  
    语句6;  
    ...  
}
```

\*条件嵌套: else属于最靠近它的if

```
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<10)
    System.out.println("儿童");
else if(age<18)
    System.out.println("少年");
else if(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;...  
}
```

// 整数表达式的类型必须是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); // x=100

## • 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（语句块）。
- ✓ 表达式1和表达式3可以是逗号隔开的多条语句，进入循环执行一次表达式1和每次循环结束都执行一次表达式3。
- ✓ break和continue表示跳出循环语句执行或跳到加标号的循环语句执行。

```
int sum1=0;
for(int i=0;i<=100;i++){
    sum1=sum1+i;
}
System.out.println("sum1(1~100)=" + sum1); // sum1(1~100)=5050
```

```
double sum2=0;
cnt = 0;
double scores[]={100.0, 90.2, 80.0, 78.0, 93.5};
for(double score:scores){
    sum2=sum2+score;
    cnt++;
}
System.out.println("avg score=" + sum2/cnt); //avg score=88.34
```

← **foreach语句**

```
int sum3=0;
int k=0;
while(k<=100){
    sum3=sum3+k;
    k++;
}
System.out.println("sum3(1~100)=" + sum3); // sum3(1~100)=5050
```

```

int sum4=0;
k=0;
do{
    sum4=sum4+k;
    k++;
}while(k<=100);
System.out.println("sum4(1~100)=" +sum4); // sum4(1~100)=5050

```

/\* 求距阵之和

```

* 1 2 3 ... 10
* 1 2 3 ... 10
* .....
* 1 2 3 ... 10
*/

```

```

int sum5=0;
for(int i=1;i<=10;i++){
    for(int j=1;j<=10;j++){
        sum5=sum5+j;
    }
}
System.out.println("triangle1="+sum5); //triangle1=550

```

// Java局部变量的作用域为方法体

```

void f(){
    int k=0;

    ...
    if(s==0){
        int n =5;
        int k =5;    //错误！！
    }
}

```

// for语句可以重复定义循环变量

```

int sum6=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
        }
        sum6=sum6+j;    // 除去对角线的矩阵之和
    } ← continue跳到这里
}
System.out.println("triangle2="+sum6); // triangle2=495

```

```

int sum7=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
        }
        sum7=sum7+j;    // 下三角加对角线矩阵之和
    }
} ← break跳到这里
System.out.println("triangle3="+sum7); // triangle3=165

```



```

int sum8=0;
cont:
for(int i=1;i<=10;i++){ System.out.println("*");
    for(int j=1;j<=10;j++){
        if(j==i){
            continue cont; // 一直执行并跳出内层循环直到标号指出外层循环
        }
        System.out.print(" "+j); sum8=sum8+j; // 下三角加对角线矩阵之和
    }
}
System.out.println("*");
System.out.println("triangle4="+sum8);
int sum9=0;
prn:
for(int i=1;i<=10;i++){ System.out.println("*");
    for(int j=1;j<=10;j++){
        if(sum9>=120){
            break prn; // 跳出标号指出的外层循环并执行其后的语句
        }
        System.out.print(" "+j); sum9=sum9+j; // 部分三角矩阵之和
    }
}
System.out.println("*"); // 执行"break prn"之后会直接跳到这条语句执行
System.out.println("triangle5="+sum9);

```

```

*
*
1*
1 2*
1 2 3*
1 2 3 4*
1 2 3 4 5*
1 2 3 4 5 6*
1 2 3 4 5 6 7*
1 2 3 4 5 6 7 8*
1 2 3 4 5 6 7 8 9*
triangle4=165
*
1 2 3 4 5 6 7 8 9 10*
1 2 3 4 5 6 7 8 9 10*
1 2 3 4*
triangle5=120

```

# 数组

ArrayDef.java

- Java的数组是一个对象(Arrays类)，其元素是基本数据类型或对象。
- 数组初始化后，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。

**int stu[23]; 出错**

```
import java.util.Arrays;
int sample[];
sample = new int[8];
sample[7]=100;
System.out.println(sample[7]); // 定义数组对象（未初始化，不能使用）
System.out.println(sample[0]); // 初始化数组，分配8个元素的存储空间。
// 数组引用方法
// 显示第7个元素：100。下标从0开始
// 显示：0（默认值）。
```

```
int 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]); //显示字符：大。
String[] s1= {"John", "Wade", "James"}; //初始化一维字符串数组
System.out.println(s1[1]); // 显示字符串：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。** 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
    }
}

```

**//下面程序做了什么？**

```

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

```

```
import java.util.Arrays;
```

ArrayOp.java

```
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)
System.out.println(s1); // Hello*** (显示为空格)
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);
```

```
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);
}
int pos=Arrays.binarySearch(s1,'1'); // (二分) 查找值为1的元素
System.out.println(pos);
```

\* `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

StringDef.java

字符串类型(String)为一个用于文字操作的类, 其值为一串字符, 采用Unicode编码存储。

[C++的String](#)

```
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." };
String s6 = s1.concat(s2);
String s7 = s1 + s2;
boolean b1 = s1.equals(s2);
boolean b2=s1.equalsIgnoreCase("hello");
boolean b3 = (s1==s2);
boolean b4 = s1.isEmpty();
int len = s1.length();
String s8 = Arrays.toString(s5);

String f0 = String.valueOf(100.3f);
double f1 = Double.parseDouble("100.3");
```

// 定义一个字符数组  
// 建立字符串: Hello  
// 建立字符串: World  
// 从字符数组建立字符串: abcde  
// 从字符数组子集建立字符串: bcd  
// 字符串数组  
// 字符串并置: HelloWorld  
// 字符串并置(同concat): HelloWorld  
// s1和s2 (对象内容)是否相同: false  
// 相等比较, 忽略大小写: true  
// 对象值是否相同: false  
// 空串判断, 与equals("")相同  
// 字符串长度: 5  
// 转变为用逗号隔开的  
// 字符串: This,is,a,test.  
// 将浮点数转换为字符串  
// 将字符串转换为Double数

```

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

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

```

//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.

//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");             // 第一个匹配子串的索引:1 未找到返回-1.
int pos2 = s11.indexOf("e",3);           // 结果:4 从位置3开始查找
int pos3 = s11.lastIndexOf("e");         // 从尾部开始查找子串:4
s12=String.format("%05d,%s",501,"op");  // 格式化字符串: 00501,loop
String regex =“(;|,)” ;                  // 正则表达式 :;或, 与[,;]相同
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

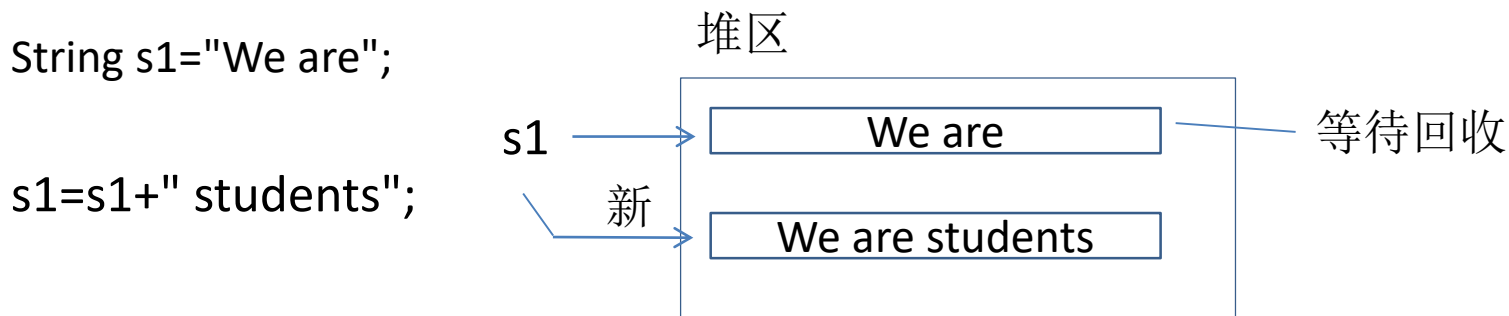
bytes[] s14 = s11.getBytes("ISO-8859-1"); // 把s11(GB2312)转换成字节数组
String s15 = new String(s14,"GB2312");    // 再把字节数组转换为字符串
int n1=s1.codePointAt(i);                 // 取到s1的第i字符的编码(UNICODE)

```

- \* ISO-8859-1编码为单字节编码，常用于把任何编码的字符转换为单字节数组。
- \* 数值和日期与字符串之间的转换方法见附录。



**String 是不可变的对象**，每次修改其内容都会生成了一个新的 String 对象，原来的String对象将不再使用，等待垃圾回收器自动清理它们，因此，频繁改变字符串变量会十分低效。



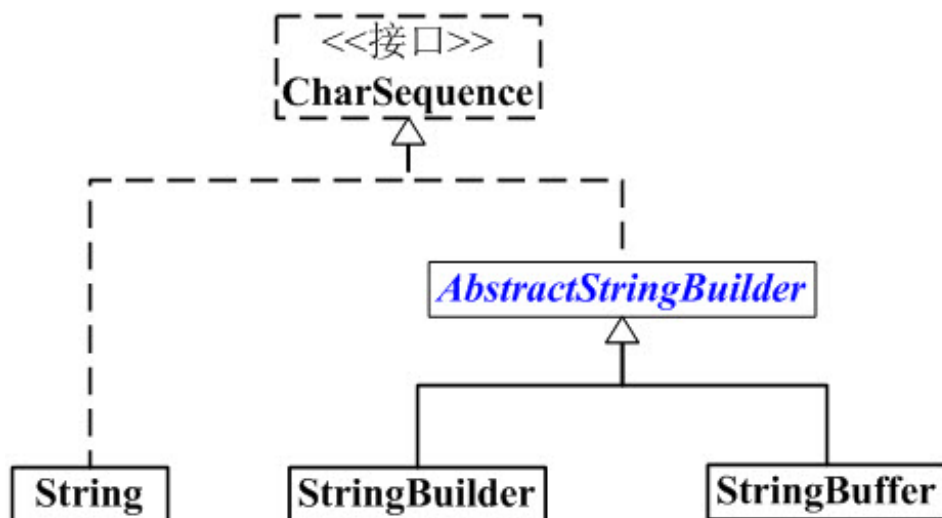
为了提高效率，可以使用**StringBuilder**和StringBuffer。类StringBuffer和StringBuilder (java.lang.\*) 的主要操作有 append 和 insert 方法：

```
StringBuilder s20=new StringBuilder("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
```

StringBuffer对方法有同步机制，一般用于多线程环境，而StringBulider适用于单线程环境。对于单线程编程，StringBulider比StringBuffer更有效率。

## StringBuffer和StringBuilder对象的常用方法

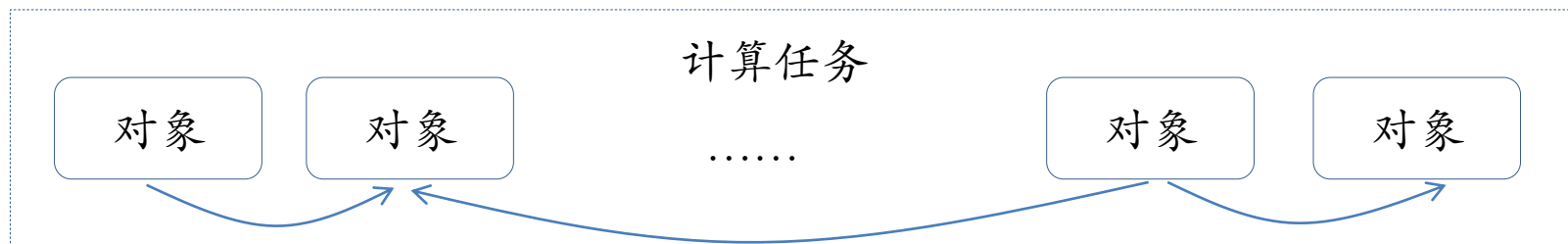
```
append(String str)           // 数值变量等类型也可以作为参数
insert(int offset, String str) // 第二个参数和append一样
delete(int start, int end)    // 删除一个子串
indexOf(String str, int fromIndex) // 从fromIndex开始查找一个子串
replace(int start, int end, String str) // 从start到end替换一个子串
substring(int start, int end)
reverse()    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



- CharSequence是一个操作字符数组的接口，它只包括length(), charAt(int index), subSequence(int start, int end)这三个方法。

# 对象和类

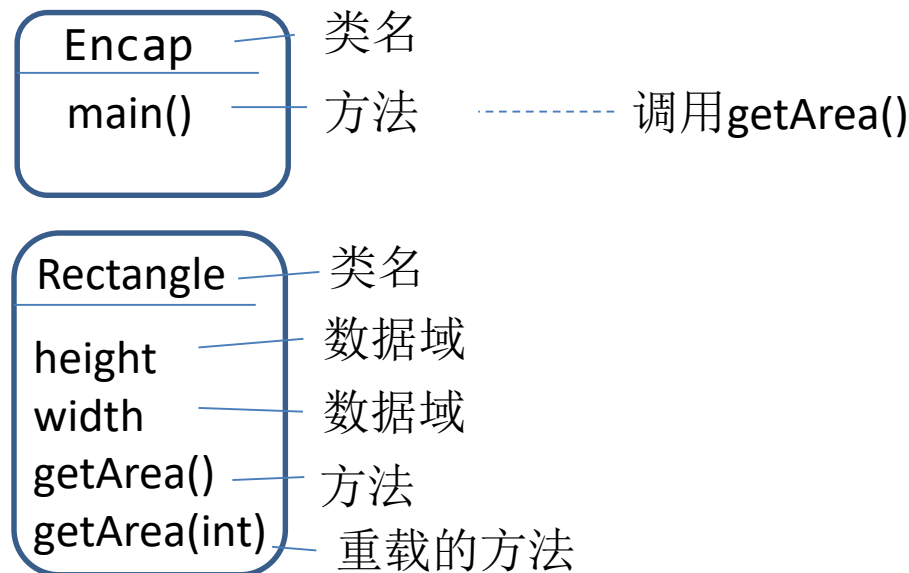
- 要完成一个计算任务必然要涉及很多事物，在面向对象程序设计中把这些事物称为**对象(Object)**，并把具有相同属性和操作的对象划分为**类(Class)**。对象也称为类的实例(**Instance**)，简称**实例**。
- 对象通过操作对外提供服务。与采用函数调用的**面向过程的程序设计**不同，**面向对象的程序设计**是通过对象之间的相互操作来完成计算任务的。



- Java对象的属性被称为**数据域(data field)**，而Java对象的操作称为**方法(method)**，在其他语言中，对象属性和方法有时也被称为**成员变量(数据成员、字段)**和**成员函数**，统称为对象的**成员**。
- 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)**是指把对象的数据域和方法封装为一个整体对外提供服务，使用者不必知道实现细节，通过调用对象方法使用对象提供的服务。
- 为了实现一个对象的方法，可以使用该对象的数据域和调用其他对象的方法。
- 同一个类中可以定义多个同名但是参数个数或类型不同的方法，称为**重载(overwrite)**
- 下面是一个计算面积的类的例子，这里定义了两个类Rectangle和Encap。 Rectangle封装了两个数据域height和width、一个方法getArea()和一个重载方法getArea(int) 。



```

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
    };
    double getArea(int times){ // 重载方法
        return height*width*times;
    };
}

```

单一职责原则 (Single responsibility principle) 就是指一个类只担负一个职责。 -- 面向对象设计原则之一

```

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

```

//运行结果: 200.0

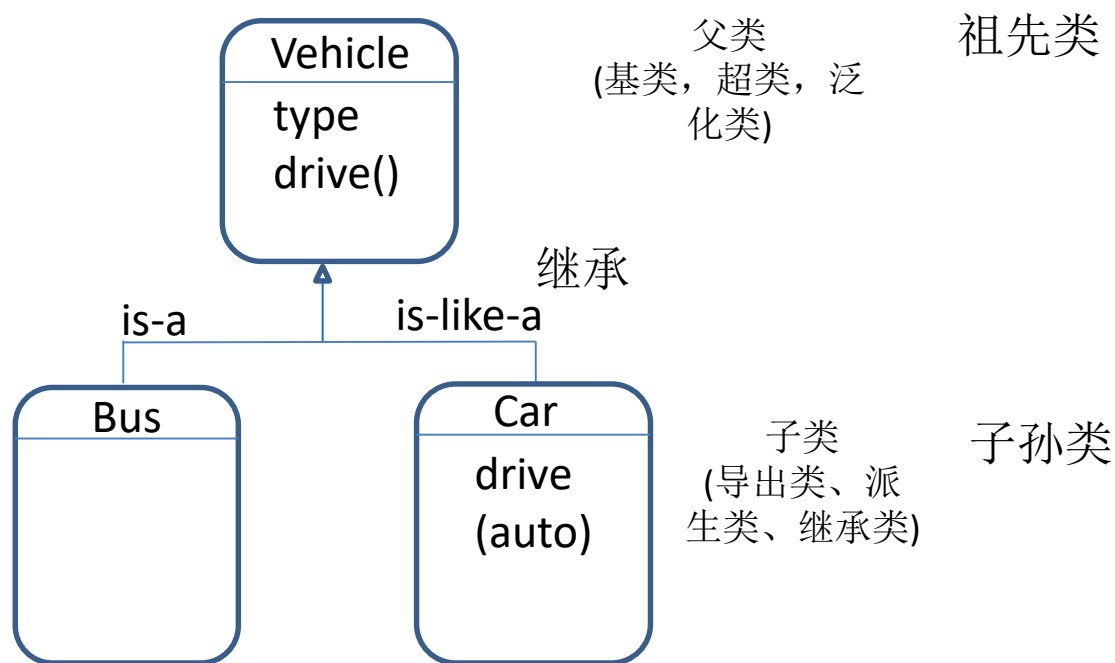
- 一个java文件可以包含很多class, 但是只能有一个public的class。编译时, 每个class都会编译成一个独立的class文件。每个类的信息会保留在其class文件的头部。
- 下边是Java对数据域(这里是Height)赋值的标准方法。

```

void setHeight(val){ height=val;}
int getHeight(){return height;}

```

- 通过**继承性(Inheritance)**，子类可以自动共享父类的非私有的属性和操作。子类也称为导出类、派生类、继承类、具体类，父类也称为基类、超类、泛化类。
- 子类与父类之间是**is-a**（完全继承）或**is-like-a**（加入了自己的属性或方法）的关系。Java的所有基类自动继承Object类(系统类)。
- 下面举一个继承性的例子。**Bus**和**Car**都是类**Vehicle**的子类，它们都继承了类**Vehicle**的数据域**type**和方法**drive()**。**Car**增加了一个方法**drive(auto)**。

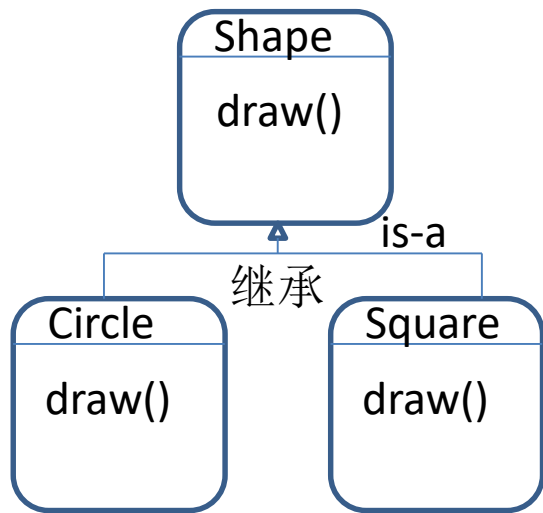


里氏替换原则(Liskov Substitution Principle LSP)中说，任何基类可以出现的地方，子类一定可以出现。  
--面向对象设计的基本原则之一

```
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
}
class Car extends Vehicle{ //子类
    void drive(boolean auto){//重载(overload)方法
        System.out.println((auto?"run(auto)!=":"run(no-auto)!")
            + this.getClass());
    };
}
public class Inherit{
    public static void main(String args[]){
        Bus bus = new Bus();
        bus.drive();
        Car car = new Car();
        car.drive(true);
    }
}
```

```
C:\Java1>java Inherit
run!class Bus
run(auto)!class Car
C:\Java1>
```

- **多态性(Polymorphism)**就是使用同一类的对象的不同操作产生了不同结果的现象。
- 实际上，多态性是通过子类对父类的方法进行覆盖或重写(**override**)实现的。下面是多态性的例子，**Circle**和**Square**都是**Shape**的子类，它们的方法**draw()**覆盖了**Shape**的方法**draw()**：



```
Shape shape1 = new Shape(); shape1.draw();
Shape shape2 = new Square(); shape2.draw();
Shape shape3 = new Circle(); shape3.draw();
```

- 虽然Square和Circle对象都**向上转型**为Shape类，但是draw()调用的是子类的方法，因而呈现不同结果。

多态的最根本好处在于，你不必再向对象询问“你是什么类型”而后根据得到的答案调用对象的某个行为——你只管调用该行为就是了，其他的一切多态性都会为你安排妥当。

---Martin Fowler 《重构：改善既有代码的设计》



```
class Shape{           // 父类(基类)
    int color;         // 数据域
    void draw(){       // 方法
        System.out.println("draw! "+this.getClass());
    }
}
class Square extends Shape{
    @Override         // 覆盖(override)才是多态
    void draw(){ 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 */
    };
}

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();
    }
}
```

运行结果:

```
draw! class Shape
draw!! class Square
draw!!! class Circle
```

\*Java调用覆盖方法采用后期(运行时)绑定, 而调用重载方法则采用前期(编译时)绑定。


# 重载、覆盖与传址

Overload.rar

- 一个类中使用参数个数或类型不同的同名方法的做法叫**重载(Overload)**。

```
class A {  
    int x1;  
    void hi(){ x1=5;}  
    void hi(int y){ x1=y;}  
}
```

重载 (Overload)



- 子类可以重载父类的方法，如果参数完全相同，则会覆盖(改写)(Override)父类的方法，此时，要通过super对象才能调用到父类的同名方法。

```
class B extends A {  
    @Override  
    void hi(int y){  
        super.hi(y);  
        System.out.println(x1);  
    }  
}
```

- 如果在方法前加上**@Override**，则会让编译器检查父类是否有这个方法，如果父类没有则会报错，否则，不会检查。

- 不能根据返回值不同类型来定义重载。

```
class A {
    int x1;
    void hi(){ x1=5;}
    void hi(int y){ x1=y;}
    int hi(int y){ x1=2*y;return y;} // 出错!!!
}
```

- Java需要使用自定义类作为引用参数才可以带回参数值。

```
class C {
    int y1=9;
    int y2=18;
}

class D {
    void hi(int s, C c){
        s = 15;
        c.y1 = 5;
        c.y2 = 10;
    }
}

int x = 6;
C c = new C();
D d = new D();
d.hi(x,c);
System.out.println(x+
    " "+c.y1+" "+c.y2);
//结果: 6 5 10
```

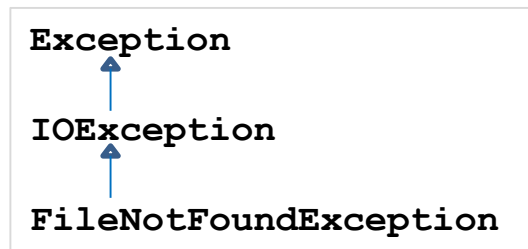
- 定义为final的方法不能被覆盖。

```
class A {
    int x1;
    void hi(){ x1=5;}
    final void hi(int y){ x1=y;}
}
```

```
class B extends A {
    @Override
    void hi(int y){ // 出错!!!
        super.hi(y);
        System.out.println(x1);
    }
}
```

- 继承并且覆盖超类方法的时候，子类中的方法的可见性必须等于或者大于超类中的方法的可见性，子类中的方法所抛出的受检异常只能是超类中对应方法所抛出的受检异常的子类。

```
public class SuperClass{                                     -- 百度百科的例子
    public void methodA() throws IOException{}
}
public class SubClassA extends SuperClass{
    //this overriding is illegal.
    private void methodA() throws Exception{}
}
public class SubClassB extends SuperClass{
    //this overriding is OK.
    public void methodA() throws FileNotFoundException{}
}
```



从Java5开始，子类中的方法的返回值也可以是对应的超类方法的返回值的子类。这叫做“协变”(Covariant)。

```
public class SuperClass {
    public Number caculate(){
        return null;
    }
}
public class SubClass extends SuperClass{
    //only compiles in Java 5 or later.
    public Integer caculate(){
        return null;
    }
}
```

# 静态成员

- 静态数据域和静态方法不需要建立实例就可以直接访问。

```
class A {  
    static String s = "Hello!";  
    static void f(){  
        System.out.println(s);  
    }  
}
```

Test1.java

执行结果:

Hello!

Hello!

```
class Test1 {  
    public static void main(String[] args){  
        System.out.println(A.s);    // 访问静态属性  
        A.f();                      // 访问静态方法  
    }  
}
```

- Java的静态数据域被所有实例所共享。

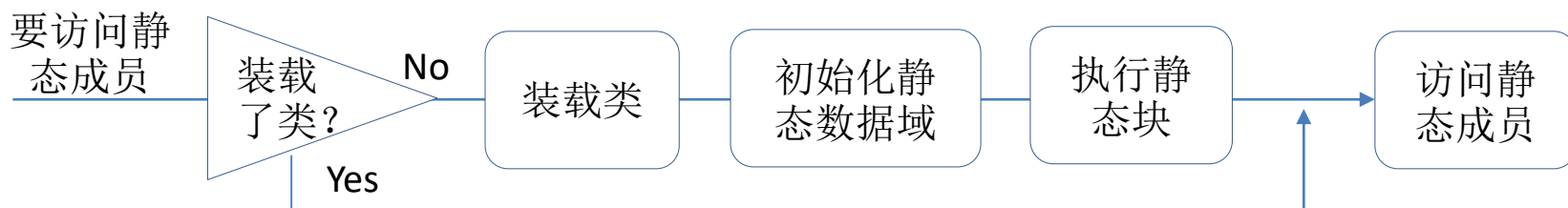
```
class Test2 {  
    public static void main(String[] args){  
        A a1 = new A();  
        A a2 = new A();  
        a1.s = "Hi!";  
        System.out.println(a2.s);  
    }  
}
```

Test2.java

执行结果:

Hi!

- 在第一次访问静态数据域和静态方法时系统会装载它们的类，如果该类没有被装载的话。系统同时会初始化其所有静态数据域并执行其静态块。这些静态数据域要到程序结束时才会被释放。



- Java没有全局变量和函数。由于Java的静态数据域和方法存在于整个程序执行期，它们可以用作类似C++的全局变量和函数。

```
public class Global {  
    public static int x = 20;  
    public static void f(){ }  
}
```

Global.java

```
class B {  
    void fB(){  
        int y = Global.x;  
        Global.f();  
    }  
}
```

```
class C {  
    int z;  
    void fC(){  
        Global.x = z + 2;  
        Global.f();  
    }  
}
```

- 非静态方法可以直接访问静态数据域或方法。静态方法不能直接访问非静态数据域和方法，静态方法只有通过实例才能访问非静态数据域和方法。

```
class D {  
    String s = "Hello!";           // 非静态数据域  
    static String t = "Hi!";       // 静态数据域  
  
    void fD1(){                    // 非静态方法  
        System.out.println("D, "+s);  
    }  
  
    static void fD2(){             // 静态方法  
        System.out.println("D, "+t);  
    }  
  
    public static void main(String args[]){ // 静态方法  
        System.out.println(s); // 访问非静态数据域，出错!!  
        System.out.println(t); // 访问静态数据域  
        fD1();                 // 访问非静态方法，出错!!  
        fD2();                 // 访问静态方法  
    }  
}
```

D.java

运行结果（删除错误语句后）：

```
Hi!  
D,Hi!
```

D2.java

```
class D2 {
    String t = "Hello!";    // 非静态数据域
    E s = new E();          // 非静态数据域
    void fD(){              // 非静态方法
        System.out.println(t);
    }
    public static void main(String args[]){ // 静态方法
        D2 d = new D2();
        d.s.x = "Hi!";      // 直接用s.x会出错
        d.fD();              // 用fD()代替d.fD()会出错
        System.out.println(d.s.x);
        System.out.println(d.t); // 用t代替d.t会出错
    }
}
```

运行结果:

```
Hello!
Hi!
Hello!
```

- 与C++不同，Java没有静态局部变量。

```
class F {
    void fF(){
        static String t = "Hello!"; //出错!!
    }
}
```

F.java



- 静态方法不能被非静态方法覆盖，但是可以被静态方法覆盖。

G.java

```
class G {
    static void fG(){ // 静态方法
    }
}
class H extends G {
    static void fG(){ // 覆盖静态方法，不能加@Override,
    }                // 也不能用super.fG()访问父类的方法
}
    去掉static将出错
```

- 因为**this**、**super**都是与实例相关的，在静态方法中不能使用它们。

```
class I {
    void fI(){ // 静态方法
    }
}
class J extends I {
    J(){}
    int x;
    static void fJ(){
        this.x = 2;    // 出错!!
        super.fI();    // 出错!!
    }
}
```

```
I.java:9: 错误: 无法从静态上下文中引用非静态 变量 this
    this.x = 2;    // 出错!!
    ^
I.java:10: 错误: 无法从静态上下文中引用非静态 变量 super
    super.fI();    // 出错!!
    ^
```

# 构造器

- 构造器(constructor)是名字与类名相同的方法，一般用于对数据域进行初始化，没有返回值。在创建实例时系统会首先调用构造器。
- 构造器可以重载，对于多个构造器，系统要根据创建实例时使用的参数情况来选择不同的构造器。
- 构造器默认为public的，不可以为static的。构造器之间可以采用this进行相互调用。

```
class A { A.java
    int x;
    A(){
        this(10);    // 调用重载的构造器，必须是第一个语句
    }
    A(int y){        // 重载
        x = 2 * y;
    }
}
```

- 如果一个类没定义构造器，系统会为它创建一个无参数的默认构造器。

- 子类生成实例时会先调用父类的构造器，再调用自己的构造器。
- 如果父类没有定义构造器，则会调用父类的默认构造器。
- 如果父类定义了构造器，则子类只能调用其中之一，而不能调用默认构造器。如果子类构造器显式地调用了父类的构造器，则它必须是第一条语句，否则系统会默认调用其无参数构造器，如果父类没有无参数构造器，则会出错。

```
class B {
    String x;
    B(String y){
        x = "" + y + "";
    }
}
```

D.java

注释super之后的出错信息：

```
D. java:9: 错误: 无法将类 B中的构造器 B应用到给定类型:
    C(String z){
        ^
需要: String
找到: 没有参数
原因: 实际参数列表和形式参数列表长度不同
1 个错误
```

```
class C extends B{
    String x;
    C(String z){
        super(z);    // 显式调用父类的构造器
        this.x = z + "--" + super.x;
    }
}
```

c1的出错信息：

```
D. java:17: 错误: 无法将类 C中的构造器 C应用到给定类型:
    C c1 = new C();    // 出错!!
              ^
需要: String
找到: 没有参数
原因: 实际参数列表和形式参数列表长度不同
1 个错误
```

```
class D {
    public static void main(String args[]){
        C c1 = new C();    // 出错!!
        C c2 = new C("Hello!");
        System.out.println(c2.x);
    }
}
```

删除出错语句后的运行结果：

Hello!--'Hello!'

G.java

```
class E {
    String x;
    E(){
        x = "'Hi'";
    }
    E(String y){
        x = "'" + y + "'";
    }
}

class F extends E{
    String x;
    F(String z){ // 自动调用父类无参数构造器
        this.x = z + "--" + super.x;
    }
}

class G {
    public static void main(String args[]){
        F f = new F("Hello!");
        System.out.println(f.x);
    }
}
```

运行结果:

Hello!--'Hi!'

- 如果构造器定义为**private**，则不能在外部直接建立实例，而需要在其静态方法中创建实例。

```
class H {  
    String x;  
    private H(){  
        x = "'Hi'";  
    }  
    static H make(){  
        return new H();  
    }  
}
```

I.java

```
class I {  
    public static void main(String args[]){  
        H h = new H();    // 出错!!  
        H h = H.make();  
        System.out.println(h.x);  
    }  
}
```

删除出错语句后的运行结果:

**'Hi!'**

# 变量初始化

- 变量初始化是指给定义的数据域或局部变量赋予初值。
- 建立一个对象时，数据域是按照如下顺序进行初始化的：
  - (1) 赋予默认值，基本数据类型为0、0.0或false，对象为null。
  - (2) 在声明语句中初始化

InitStep.java

```
int x = 10;  
double y = Math.random(); // 调用方法进行初始化
```

- (3) 在实例块中初始化实例域

```
int z;  
{  
    z = 20; // 每次创建实例时执行一次  
}
```

- (4) 在静态块中初始化静态数据域

```
static int s;  
static {  
    s = 20; // 只在装载类时执行一次  
}
```

(5) 在构造器中初始化数据域

```
class RandNum {  
    int x;  
    double y;  
    RandNum(int x){  
        this.x = x;           // this.x指本对象的成员变量x  
        y = Math.random();    // 也可以用this.y  
    }  
}
```

- 局部变量不会被自动赋予默认值，因此必须对它赋值来进行初始化。

```
class ShowInit {                                     ShowInit.java  
    int x;  
    void show(int x){  
        int y;  
        int z = x+y;  // 出错！！y没有初值  
    }  
}
```

# final关键字

- 定义为**final**的数据域在初始化后不能被再次赋值。

```
class A {  
    final int COUNT = 20; // 常量  
    final double x;       // 变量(只能赋值一次)  
    A(){                  // 构造器  
        x = Math.random(); // 每个实例得到不同的x值  
    }  
    void hi(){  
        x = 10.0; // 出错!!!  
    }  
}
```

A.java

B.java

- 如果x定义为静态(static)，则类装载时赋值一次。

```
final static double x = Math.random(); // 所有实例的值相同
```

- **final**参数在方法内不能被再次赋值。

C.java

```
class C {  
    void hi(int x, final double y){  
        x = 2;  
        y = 5; // 出错!!!  
    }  
}
```



- 如果定义为final的数据域是引用型的，其内容(对象)是可以被修改的。

```
class D {
    final static E s = new E();
    final static String t = "Hello!";
    public static void main(String args[]){
        s.x = "Hi";
        System.out.println(s.x);
        t = "Hi";           // 出错!!
    }
}
```

```
class E {
    String x;
}
```

D.java

赋值String变量要修改其引用

- 定义为final的方法不能在子类中被覆盖(Override)。

```
class F {
    final int x = 10;
    int y;
    final void hi(){
        y = x * 2;
    }
}
```

```
class G extends F{
    int x;           // 新数据域
    void hi(){       // 出错!!
        x = super.x * 2;
    }
}
```

F.java

- 因为所有private方法都不能并子类继承，因此，自动指定为final的。
- 除了防止覆盖，代码少的final方法还可以在编译时代入调用位置提高效率。

- 定义为**final**的类不能被继承，因此，其所有的方法都是final的。

```
final class H {                class I extends H { // 出错!!
    int x = 10;                }
}
```

H.java

- final的局部变量只能赋值一次。

J.java

```
class J {
    void hi(){
        final double z = Math.random();
        z = 10.0; // 出错
    }
}
```

- 如果采用字面量直接初始化，final的数据域和局部变量被Java编译器当成常量。

```
class K {
    final int x = 20;
    void hi(){
        final String y = "Hello!";
    }
}
```

# 枚举类型

- 枚举类型用于定义若干有关联的命名常量，这些常量的实际取值是整数。
- 下面定义枚举类enum的导出类Signal，并给出了三个常量GREEN, YELLOW和 RED。
- enum是一种整数类，有自己的方法toString()，它会把整数的枚举值转化为常量名。

```
enum Signal {    // enum的导出类Signal
    GREEN, YELLOW, RED
}
```

- 下面定义了Signal的实例color，并用于if语句。在switch使用时，case 要直接使用color的枚举常量。

```
Signal color = Signal.RED;
if (color==Signal.GREEN)
    color = Signal.YELLOW;
```

```
switch (color) {
    case RED:
        color = Signal.GREEN; break;
    case YELLOW:
        color = Signal.RED; break;
}
```

- 下面是使用枚举类的完整例子。

```
enum Signal {    //enum的导出类  
    GREEN, YELLOW, RED
```

enum.java

```
}  
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;  
        }  
    }  
}
```

运行结果:



```
RED  
GREEN  
YELLOW
```

```
public class Enum {  
    public static void main(String args[]) {  
        TrafficLight tLight = new TrafficLight();  
        System.out.println(tLight.color); tLight.change();  
        System.out.println(tLight.color); tLight.change();  
        System.out.println(tLight.color);  
    }  
}
```

# Java的包(package)

[参考](#)

- 概念

- 一个大的工程会包含很多class文件，Java是通过子目录对这些class文件进行分类管理的。
- 一个子目录下的所有class文件（类）形成一个包（**package**）。也可以把它们连同子目录结构放到一个jar文件中。jar文件就是Java的库文件。
- 一般地，使用网站名(group.com)来确保包名(com.group.food)的唯一性。

包com.group.food

当前目录

子目录com

子目录group

子目录food

class文件

jar文件

- 定义包

- 在源文件的第一行使用语句“`package` 包名;”指出该文件所属的包。
- 没有定义包的class文件属于**默认包**。默认包的class文件放在当前目录而不是子目录中。

Java源文件--定义包

```
package com.group.food; 第一行  
.....
```

- 引用包

- 通过import语句来引用包以便使用其中的类。只给类名时，通过引入包或默认包查找该类。

Java源文件--引用包中所有class文件

```
import com.group.food.*;  
Cookie cookie = new Cookie();
```

Java源文件—只引用包中Cookie类

```
import com.group.food.Cookie;  
Cookie cookie = new Cookie();
```

- 也可以直接指明包来使用类：  
`com.group.food.Cookie cookie = new com.group.food.Cookie();`

- 定义包和使用包的例子

- 我们定义一个包com.group.food。实际上，就是把带有package语句的Cookie.class和Bread.class放在子目录com\group\food下。该子目录可以放在任何任何目录下。这里假设放在c:\java\lib下。

```
package com.group.food;
public class Cookie {
    ...
}
```

```
package com.group.food;
public class Bread {
    ...
}
```

- PackEx.class在引用(import)包com.group.food之后才可以使用其中的类。

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

- 如何找到包所在的位置呢？环境变量CLASSPATH给出要查找的目录。

CLASSPATH=c:\java\lib;c:\classes;. （包含三个目录，其中.为当前目录）

\* 如果包和PackEx.class都在相同的目录下，则可以不设置CLASSPATH。但是设置了CLASSPATH之后，系统只会按照CLASSPATH中的目录进行查找。

- 完整的程序

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

Cookie.java

package.rar

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

Bread.java

```
package com.group.show;
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();
    }
}
```

PackEx.java



- 使用jar文件的例子

第一步、把包`com.group.food`的所有`.class`文件放在当前目录(`c:\java\lib`)的子目录(`com\group\food`)中。

第二步、在当前目录(`c:\java\lib`)中执行命令“`jar cvf food.jar *`”得到文件`food.jar` (包含了子目录信息)。

第三步、把`food.jar`放在一个任意目录(`c:\java\PackJar\`)下，并设置`CLASSPATH`。  
`set CLASSPATH = c:\java\PackJar\food.jar;c:\java\test;. (一个文件两个目录)`

第四步、运行任何引用了包`com.group.food`的`class`文件。

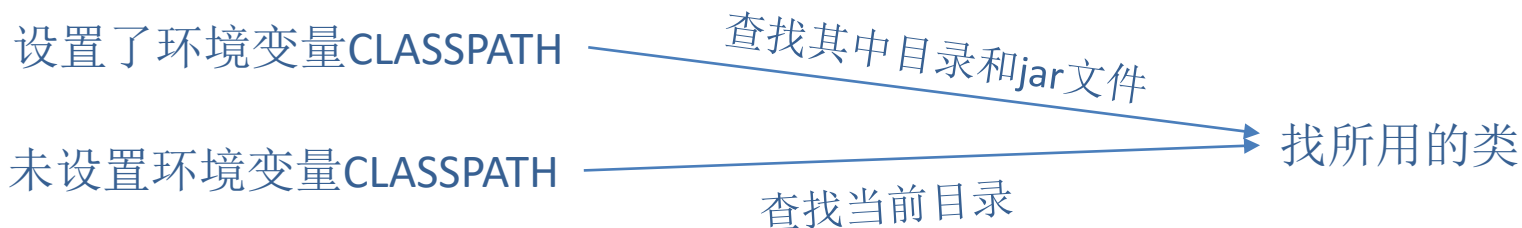
\*Java系统的lib目录中的jar文件会被自动用于查找目标类。

- 默认包和同名类

- 没定义包的`class`文件被看成使用默认包。默认包只能放在`CLASSPATH`设置的目录下，没有设置`CLASSPATH`，则放在当前目录下。
- 如果多个包都出现一个类名，只会使用第一个出现该类名的包中的类。使用其他同名类需要加上包名。

- 使用一个类时如何找到其所在的包？

- 定义了CLASSPATH后Java只会从其中目录查找包。



- 编译和运行时可以用参数-CLASSPATH或-cp直接指出CLASSPATH:  
`javac -cp c:\java\PackJar\food.jar;c:\java;. PackJar.java`  
`java -cp c:\java\PackJar\food.jar;c:\java;. PackJar`
- CLASSPATH或cp的路径中不要使用环境变量~(表示当前目录), 但是可以用其它环境变量。
- Linux下的CLASSPATH使用冒号“:”间隔目录而不是用分号“;”。

\* Java找到所用的类如果是Java文件, 每次都会自动进行编译为class文件 (如果Java文件未编译过或编译后被修改过)。

- 用 “import static 包名.类名.\*” 引入一个类的所有静态成员（静态对象、静态常量和静态方法），以便直接用*成员名*进行引用。
- 完整程序：

```
package com.group.food;
public class Rice {
    public static void eat(){
        System.out.println("Eat rice");
    }
}
```

ImportNormal.java

static.rar  
(PackStatic)

```
import com.group.food.Rice;
public class ImportNormal{
    public static void main(String[] args){
        Rice.eat();
    }
}
```

```
import static com.group.food.Rice.*;
public class ImportStatic{
    public static void main(String[] args){
        eat();
    }
}
```

ImportStatic.java

# 访问权限

- 在一个类B的方法M中是否可以访问到另一个类的数据域D和方法S，要看这两个类的关系(同属一个包，父类和子类，都不是)以及被访问类A及其数据域和方法的访问权限或可见性。Java有四种访问权限：public、private、protected、无修饰词。无修饰词类似C++的friendly。
- 下面的程序段定义了四种访问权限的数据域和两种访问权限的方法：

```
package com.vehicle;                                perm.rar
class Vehicle{                                       vehiacle.java
    public int type;
    protected int miles;
    int msg; // 无修饰词
    private int state;
    public void drive(){
        System.out.println("run!");
    }
    protected void repair(){
        System.out.println("repair!");
    }
}
```

在类B的M中 是否可以访问 → 类A的D或S

- 取决于B和A的关系：
  - 类B {
    - A的导出类
    - 与A同一个包
    - 其他类
- 取决于A、D、S的访问权限

C:\Java1\perm\com\vehicle\Vehicle.class

其他程序:

```
package com.vehicle;
```

Car.java

C:\Java1\perm\com\vehicle\Car.class

```
public class Car extends Vehicle{
    public void repair(){
        super.repair();
        System.out.println("repair, o!");
    }
}
```

```
package com.group;
```

GzCar.java

C:\Java1\perm\com\group\GzCar.class

```
public class GzCar{
    public static void main(String args[]){
        com.vehicle.Car car = new com.vehicle.Car();
        car.drive();
        car.repair();
    }
}
```

运行结果:

```
C:\Java1\perm>set CLASSPATH=.
```

```
C:\Java1\perm>java com.group.GzCar
run!
repair!
repair, o!
```

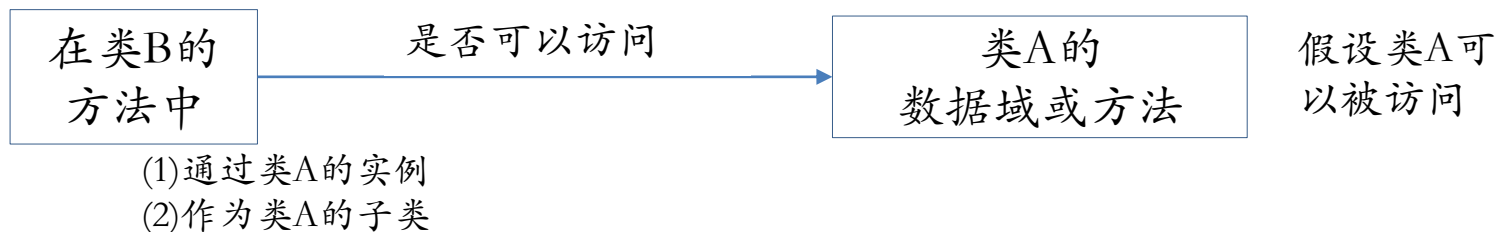
注意GzCar.java和Car.java属于不同的包。

- 类的访问权限

没有修饰词	只能被同一个包的类所访问 (default)
public	能被任何类在任何地点所访问

\* 只有内部类才可以定义private和protected权限，内部类还可以是static的。

- 数据域和方法的访问权限

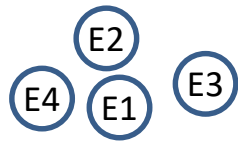


访问权限	{	public	能被任何类在任何地点所访问
		protected	只能被 <b>导出类</b> 和 <b>同一个包的类</b> 所访问
		private	只能被同一个类的方法所访问 (A=B)
		无修饰词	只能被同一个包的类所访问 (类似C++的friendly)

总结：对于不同包之间，只能访问public的或者父类protected的数据域和方法。对于相同包之间，可以访问除了private的所有其他权限的数据域和方法。

# 集合框架

- 数组类变量在初始化后不能增加或减少元素个数。使用容器类可以解决这些问题。通过映射类还可以实现按键值快速查找。
- 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)。

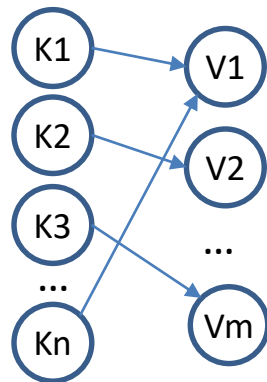


Set(无重复无序)



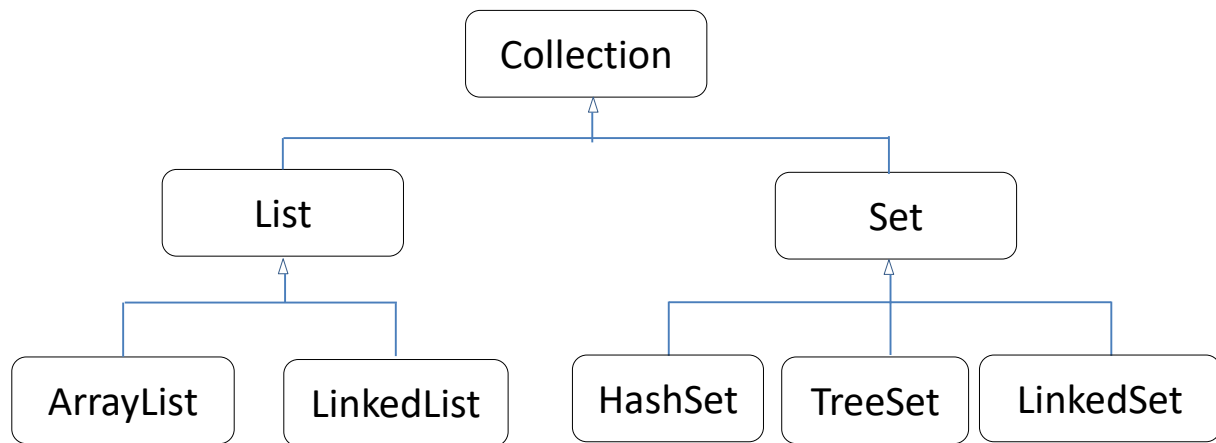
List(有序可重复)

- 映射类**Map**可以将键(Key)映射到值(Value)。其中的键值不能重复，每个键值只映射到一个值。



映射类Map

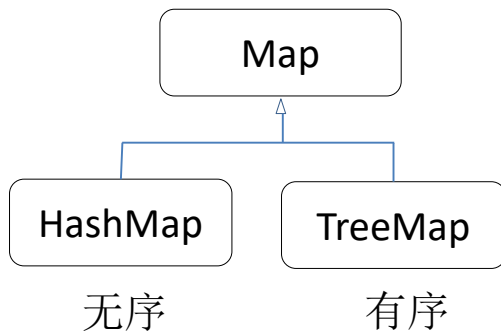
- 容器类簇



所用数据结构：      数组                      链表                      Hash表              红黑树              链表

\* 容器类还有Vector类（用于多线程环境）、Stack类(Vector的子类)、Queue类。

- 映射类簇



无序

有序

\* 映射类还有Dictionary类(已过时)， Hashtable类(已过时， Dictionary的子类， 多线程-方法sync)。



## • 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，随机访问速度慢。
- 下面的例子是采用ArrayList存储Student类实例。

```
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;  
    }  
}
```

```
ArrayList<Student> stus = new ArrayList<Student>();  
Student stu1 = new Student(101, "Wang");  
..... // stu2, stu3  
Student stu4 = new Student(112, "李四");  
stus.add(stu1); // 尾部追加  
stus.add(stu2);  
stus.add(stu3);  
stus.add(1, stu4); // 在第一个元素后面插入stu4
```

类ArrayListTest的  
部分程序

## 完整的程序

```
import java.util.*;
public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<Student> stus = new ArrayList<Student>(); // 泛型(参见附录)
        Student stu1 = new Student(101, "Wang");
        Student stu2 = new Student(102, "Li");
        Student stu3 = new Student(103, "He");
        stus.add(stu1); // 尾部追加
        stus.add(stu2);
        stus.add(stu3);
        stus.add(1, new Student(112, "李四"));
        Iterator<Student> it = stus.iterator();
        while (it.hasNext()) { // 顺序取出
            System.out.println("***" + it.next());
        }
        System.out.println("+++" + stus.get(2)); // 随机取出
        stus.forEach(stu->{stu.name=stu.name+"*"; System.out.println(stu);});
        List<Student> stus2=Arrays.asList(stu1,stu2,stu3); // 用数组转, 不能add
        System.out.println("###" + stus2);
    }
}
```

ArrayListTest.java

运行结果

```
***101 Wang
***112 李四
***102 Li
***103 He
+++102 Li
101 Wang*
112 李四*
102 Li*
103 He*
###[101 Wang*, 102 Li*, 103 He*]
```

```
* ->为lamda表达式
* stus2向上转型为List。
```

ArrayList和LinkedList的主要方法：泛型E可以使用用户自定义类或标准类(Integer, String等)

<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>boolean</b> addAll(List list)	// 将列表所有元素插入到当前列表中，返回正确或错误。
<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 fromIndex和 toIndex(不包括)之间的所有元素。

- \* 在多线程环境下使用CopyOnWriteArrayList代替ArrayList，链式使用ConcurrentLinkedQueue。
- \* ConcurrentLinkedQueue为多线程队列，也能使用上述方法(除了带index的)和前面的Iterator循环。
- \* 多线程队列还可以使用BlockingQueue(接口)，它的实现类有ArrayBlockingQueue、DelayQueue、LinkedBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue、SynchronousQueue。

## • HashMap和TreeMap

随机访问和插入删除操作很快，HashMap要更快一些，但是内存占用量更大。TreeMap的键值可以有序遍历，而HashMap不行。

HashMapTest.java

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

运行结果

```
***112 李四
***101 Wang
***103 He
+++112 李四
```

\* stus向上转型为Map类，也可以继续使用HashMap。

## 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K使用标准类(Integer, String等)
- \* 泛型V还可以使用用户自定义类
- \* 在多线程环境下使用**ConcurrentHashMap** (采用lock和obj. syn, 比HashTable效率高)

# 附录1、安装JDK

下载最新版JDK: [JDK](#)

\* 共有五个版本，根据使用的操作系统进行下载



Java SE Development Kit 10.0.2		
You must accept the <a href="#">Oracle Binary Code License Agreement for Java SE</a> to download this software.		
<input type="radio"/> Accept License Agreement <input checked="" type="radio"/> Decline License Agreement		
Product / File Description	File Size	Download
Linux	306 MB	<a href="#">jdk-10.0.2_linux-x64_bin.rpm</a>
Linux	338.43 MB	<a href="#">jdk-10.0.2_linux-x64_bin.tar.gz</a>
macOS	395.46 MB	<a href="#">jdk-10.0.2_osx-x64_bin.dmg</a>
Solaris SPARC	207.07 MB	<a href="#">jdk-10.0.2_solaris-sparcv9_bin.tar.gz</a>
Windows	390.25 MB	<a href="#">jdk-10.0.2_windows-x64_bin.exe</a>

安装过程: [网址](#)

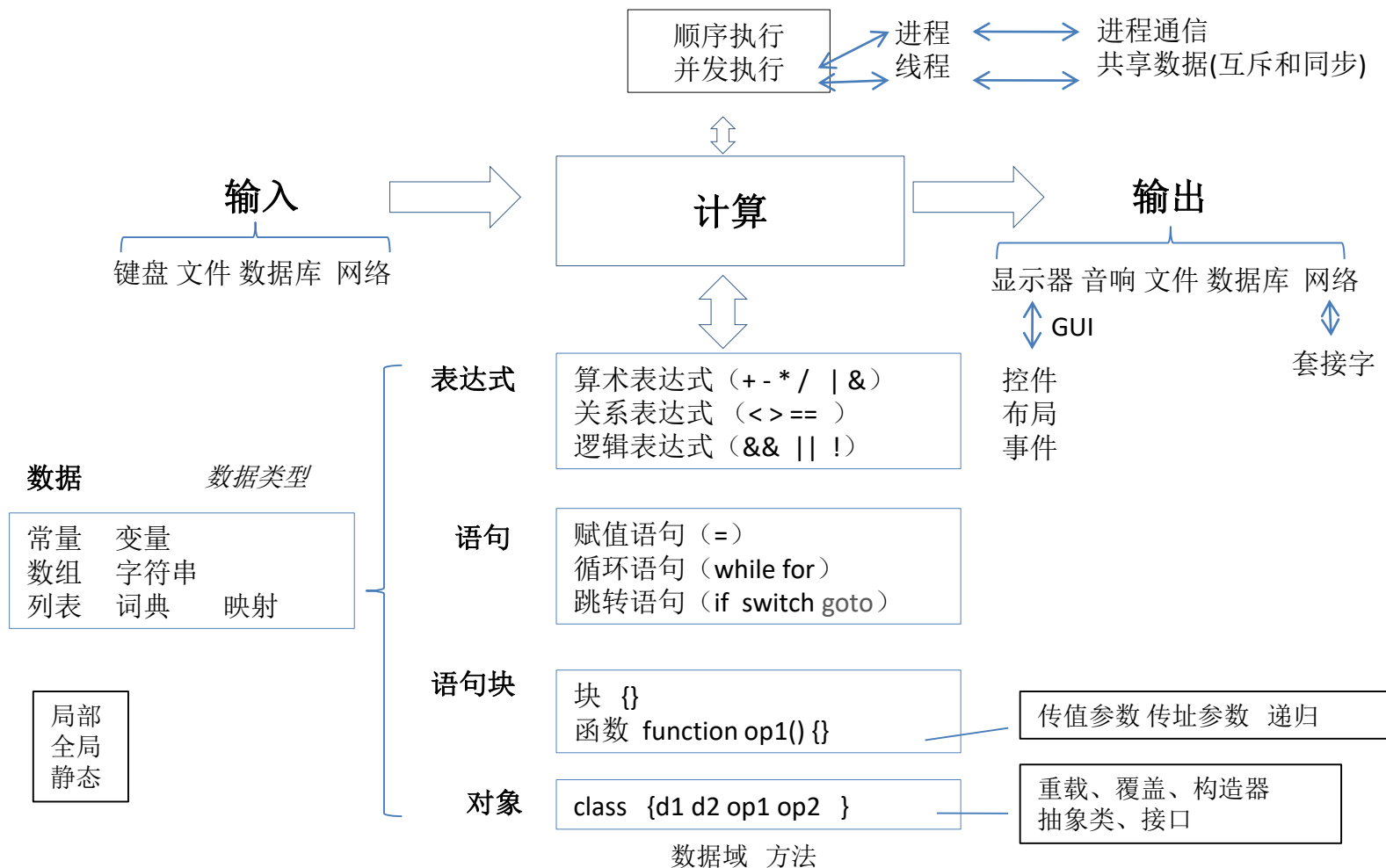
我安装的是JDK 1.8.0\_66（2014年发布，比较稳定，全世界80%使用）

集成环境可以使用Eclipse和IntelliJ IDEA（Eclipse可以在教学网站上下载）

为了可以查看Java类的源码，需要在Eclipse中安装源码: [链接](#)

# 附录2、程序设计语言

程序设计语言主要用于完成计算任务：输入数据，进行计算，输出结果。下面是程序设计语言的主要组成部分。

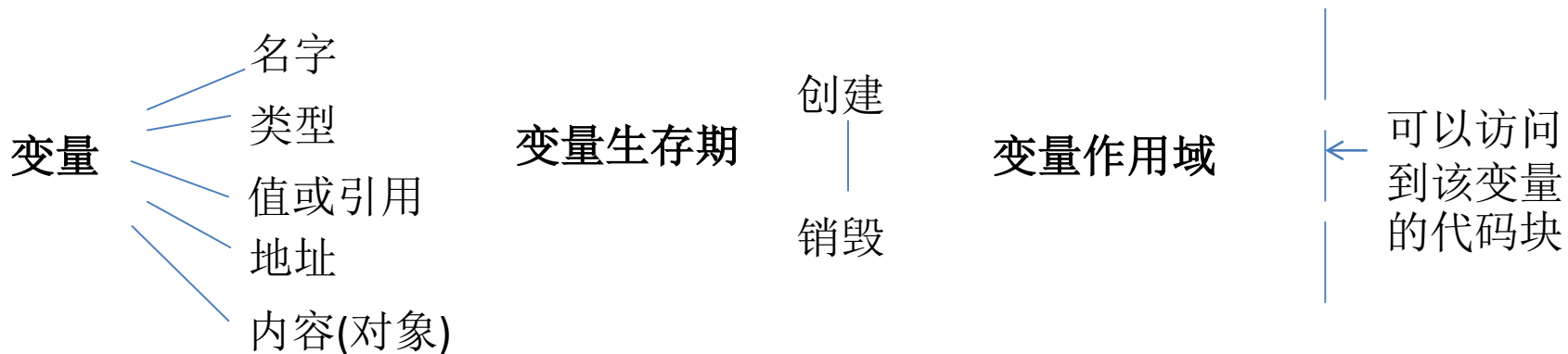


# 附录3、作用域和生存期

- 是否可以访问到变量取决于变量的生存期和作用域。
- **变量的作用域**是指可以访问到该变量的范围，例如：局部变量的作用域是变量定义的函数或块，私有数据域为所在对象的内部，静态数据域是整个程序。
- **变量的生存期**是指变量占用内存的时间，如果超出变量的生存期，即使在作用域内也不能访问该变量。局部变量的生存期为从其所在方法，非静态数据域的生存期与其所在对象的生存期相同。静态数据域的生存期都是从其所在类装入内存到应用程序结束。
- Java的**同名局部变量不能在同一个方法中定义两次，尽管它们位于不同语句块。**（C++可以，只使用最近定义的）。
- 在Java中，没有初始化的基本数据类型的数据域取值为0、0.0或者false，引用类型的数据域为null。未初始化的局部变量均为未定义(undefined)，不能直接使用。

[参考](#)





```
class MyMath {  
    String msg;  
    int sum(int n){  
        int res=0;  
        for(int i=1;i<n;i++){  
            n+=i;  
        }  
        return res;  
    }  
}
```

数据域 { 生存期: 对象存在期  
作用域: 由类和数据域的访问权限决定

局部变量 { 作用域: 定义它的块或方法  
生存期: 它所在的方法被调用的时候

Java没有全局变量!

- 变量的类型有基本数据类型和引用类型。变量的地址指出值存放的位置。引用类型的变量的值指向对象的存放位置(堆)。

# 附录4、命名法

- Java的变量名和方法名一般采用**驼峰命名法**，即第一个单词首字母小写，其余单词首字母大写，例如，`userAge`。
- 变量名和参数名也可以采用**匈牙利命名法**，即开头字母用变量类型的缩写，其余单词首字母大写，如，开始`pUserName`用于参数，`sUserName`用于局部变量(String类型)，`mUserName`用于成员变量。
- 类名采用**帕斯卡命名法**，所有单词的首字母都大写，例如，`UserRecord`。
- 常量采用**下划线命名法**，即单词之间采用下划线分割，如，`MAX_USER_CODE`。
- 项目名全部小写，单词之间用中划线连接，例如，`xingang-sale`。
- 包名全部小写，例如，`com.group.food`。
- 布尔变量取名加前缀`is`、`has`等，例如，`isValid`，`canRemove`，`shouldMigrate`，`hasObservers`，`needsMigrate`。

# 附录5、捕捉错误

```
public class CatchError {
    public static void main(String args[]){
        int x;
        try {
            // 打开例外处理语句
            for (int i = 5; i >= -2; i--) {
                x = 12 / i;
                // 出现例外(x==0)后将不执行后面的语句
                System.out.println("x=" + x); // 直接跳到catch内执行
            }
        }
        catch (Exception e) {
            // 捕捉例外信息。可以并列用多个catch
            System.out.println("Error:"+e.getMessage()); // 显示当前错误信息
            // e.printStackTrace(); // 显示系统错误信息
        }
        finally{
            // 出现例外必须执行这里的语句
            x=0;
        }
        System.out.println(x);
    }
}
```

执行结果:

x=2  
x=3  
x=4  
x=6  
x=12  
Error:/ by zero  
0

如果一个方法不愿意处理一些例外，可以采用throws定义方法的例外，把这些例外交给调用者去处理。

```
public class DivideClass {  
    void divide() throws Exception {           // 出错后交给调用程序处理  
        for (int i = 5; i >= -2; i--) {  
            int x = 12 / i;                     // 出现例外(x==0)后将不执行后面的语句  
            System.out.println("x=" + x);  
        }  
    }  
}  
  
public class ThrowError {  
    public static void main(String args[]){    // main为主程序入口  
        try {                                  // 打开例外处理语句  
            DivideClass div = new DivideClass();  
            div.divide();  
        }  
        catch (Exception e) {                 // 捕捉例外信息。可以并列用多个catch  
            System.out.println("Error:"+e.getMessage()); // 显示当前错误信息  
        }  
    }  
}
```

执行结果：  
x=2  
x=3  
x=4  
x=6  
x=12  
Error:/ by zero

# 附录6、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

[Integer Float](#)

# 附录7、日期和字符串之间的转换

DateConversion.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. cal.getTime().
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();

或

Long time= new Date().getTime();

两种方法都取到距离新纪元时间(epoch time)1970年1月1日0点0分0秒的毫秒数。

# 附录8、Math

MathFunc.java

<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>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？



# 附录9、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);
}
```

# 附录10、参考资料

- Bruce Eckel, Java编程思想（第4版），机械工业出版社，2012
- Cay S.Horstmann, G.Cornell, Java核心技术，机械工业出版社，2012
- Y. Daniel Liang，Java语言程序设计（第8版），机械工业出版社，2011
- <http://docs.oracle.com/javase/8/docs/>
- <http://api.apkbus.com/reference/java/io/package-summary.html>
- <http://docs.oracle.com/javase/tutorial/>