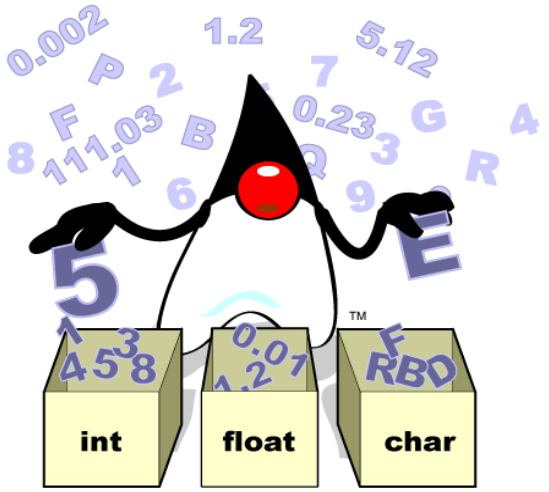
第3章 Java语言基础



3.1 变量(Variable)

- 变量是使用标识符命名的数据项，在程序中用于存放数据。

声明一个变量的方法：

变量数据类型 变量名

举例：

byte largestByte = Byte.MAX_VALUE (127)
char aChar = 'S'
boolean aBoolean = true

• 数据类型

变量的数据类型决定了变量存放的数据以及在数据上可以进行的操作。

例如：整型变量可以存放整数，可以对整数进行加减等算术运算。

Java语言中有两种数据类型：

基本型(primitive)和引用型(reference)

* 基本型变量可以存放相应格式的数据如：数值，字符或布尔变量等。

Java语言中的基本型变量：

整型: byte (8位有符号数), short (16 位有符号数), int (32 位有符号数),long (64 位有符号数)

实数类型: float (32 位IEEE754浮点数), double (64 位IEEE754浮点数)

字符型: char (16 位,unicode编码)
布尔型：boolean (8 位 / 1 位)

注意：基本型变量的格式和容量与所在平台无关。

对比： C语言中 long型， 在32位机器上是 4byte (32bit)
在64位机器上是 8byte (64bit)

Java语言中long 型，在32位机器上是8byte(64bit)
在64位机器上是8byte(64bit)

C/C++中采用 sizeof()运算符为数据项分配具体的字节数，以确保移植性。Java中没有sizeof()运算符。

数据类型长度的唯一性保证了Java代码的可移植性。

C语言支持无符号数和有符号数， eg. 12345U, 0x1234u
12345, 0x1234
unsigned ux,uy;

C语言中的有符号数和无符号数的转换可以显式完成，也可隐式完成， int tx; unsigned ux; tx = (int) ux; (显式)
tx = ux; (隐式）
注意： 隐式转换有时会得到程序员预料之外的结果，从而在程序中出现错误。

Java语言只支持有符号数

Unicode编码

Unicode编码采用4个字节表示字符，但目前大部分语言和操作系统支持采用2个字节的Unicode. Java语言中使用2个字节的Unicode. 特别地， 为了和ASCII编码兼容，在某些情况下也使用UTF-8编码方式的Unicode编码，即可变长Unicode编码，对ASCII字符，采用1字节编码，对中文采用3或4字节编码.有的语言，例如REBOL,为了使得代码长度变短，采用UTF-8进行编码。

变量可以直接赋值：

```
int anInt = 4;
```

数据类型和数值对照：

数值	数据类型
178, 0x43, 0b11110000	int
8864L	long
37.266	double
37.266D	double
87.363F	float
26.77e3	double
' C ' '\u0043'	char
true	boolean
false	boolean

基本型变量的类型转换

自动类型转换
eg. byte -> int, int-> long 数值范围小的量-> 数值范围大的量

int x; byte y; x = y;

强制类型转换
eg. int-> byte 数值范围大的量->数值范围小的量

int x; byte y; y = (byte)x; 注意可能发生的信息丢失
int x ; byte y; x = 226; y = (byte)x; y = ?

x = 226

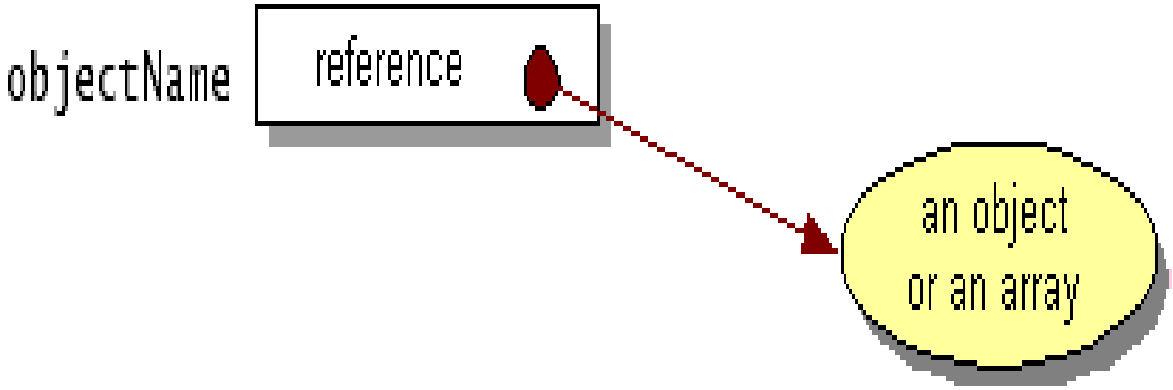
00000000000000000000000011100010 int

11100010 byte

补码->原码

y = 10011110 (-30)

* 引用型变量是对单个数值或数值集合（数组，类，接口）的引用。



注意：Java语言中的引用和C语言中的指针的相同和不同

- 变量名

简单命名（Simple name）
限定命名（Qualified name）

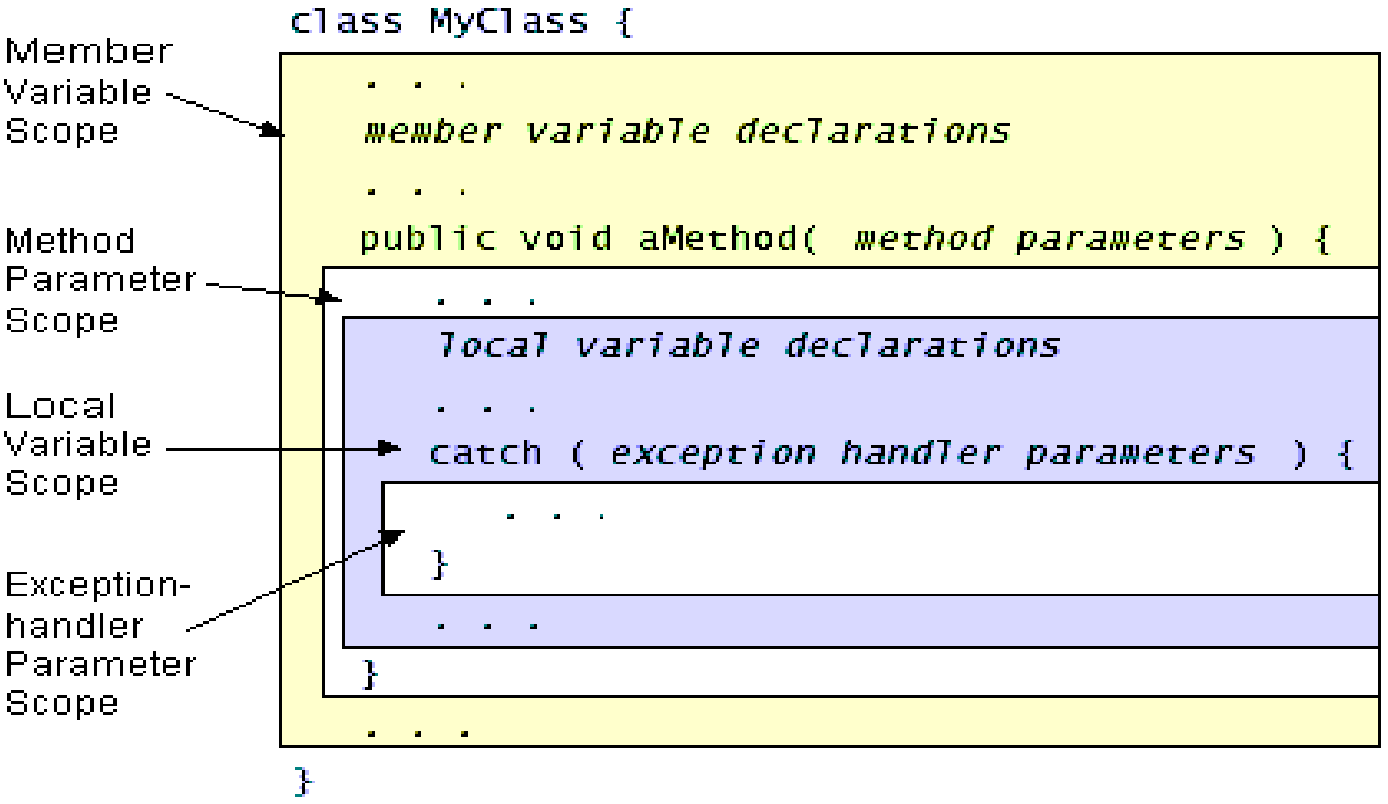
简单命名:
1. 为合法的标识符，即由以字母开头的无限制长度的Unicode字符组成。
2. 不能是关键字，布尔类型数值 (true或 false), 以及保留 null.
3. 在作用范围内必须唯一。不同作用范围内可以有同名变量。

• **作用范围(Scope)**

作用范围即在程序中一个变量可以用简单命名方式来访问的范围，它决定了系统何时为变量分配存储空间和销毁存储空间。

变量在程序中的位置决定了变量的作用范围，分为4种类型：

- *** 成员变量(member variable)**
- *** 本地变量(local variable)**
- *** 方法参数(method parameter)**
- *** 异常处理器参数 (exception-handler parameter)**



成员变量的作用范围为整个类的声明范围。
本地变量的作用范围为从其定义开始到定义该变量的语句体的结束位置。
方法参数的作用范围为整个方法的声明范围。
异常处理器参数的作用范围为**catch**语句后面从**{**到**}**的语句体。

举例：

```
if (...) {
    int i = 17;
    ...
}
System.out.println("The value of i = " + i);    // error
```

- 变量的初始化

本地变量和成员变量可以在声明的时候同时使用赋值操作符(=)进行初始化。数值的类型必须与变量类型一致。
举例：

```
int largestInteger = Integer.MAX_VALUE;

char aChar = 'S';

boolean aBoolean = true;
```

注意：方法参数和例外处理器参数不能用这种方法初始化，需要在调用的时候通过参数传递进行赋值。

- 比较实例变量和局部变量的默认初始化

实例变量
编译器会为未初使化的实例变量赋默认的初值

表 3-1　数据类型的默认值

数据类型	（字段）默认值
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
boolean	false
char	'\u0000'
String（或任意对象）	null

局部变量
编译器不会为未初使化的局部变量赋初始值。

表 3-1　数据类型的默认值

数据类型	（字段）默认值
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
boolean	false
char	'\u0000'
String（或任意对象）	null

- **终止变量 (Final Variables)**

终止变量一旦被初始化后就不能改变其数值。等同与其他编程语言中的常数。

举例:

```
final int blankfinal;

blankfinal =  1;
. . .
blankfinal = 2;  // compile-time error.
```

3.2 操作符(Operators)

操作符在单个，两个或三个操作数上进行某种操作并返回一个数值。

* 一元操作符 **i++, --k**
* 二元操作符 **i= m+n**
* 三元操作符 **i?j:k**

操作符分类：

* 算术运算操作符
* 关系运算和条件运算操作符
* 移位和逻辑运算操作符
* 赋值操作符
* 其他操作符

- 算术运算操作符

*** 二元算术运算操作符**

+	**op1 + op2**	**op1 和 op2相加**
-	**op1 - op2**	**op1 和 op1 相减**
*****	**op1 * op2**	**op1 和 op2 相乘**
/	**op1 / op2**	**op1 除以 op2**
%	**op1 % op2**	**op1 除以 op2 取余**

注意：算术运算符的返回类型

返回类型	操作数类型
long	**操作数为整型(任何一个操作数都不是浮点数(float或double)),至少一个操作数是long.**
int	**操作数为整型(任何一个操作数都不是浮点数(float或double)),任何一个操作数都不是long.**
double	**至少一个操作数是double.**
float	**至少一个操作数是float,任何一个操作数都不是double.**

举例: int i= 40; double j= 12.33

i + j = 52.33

*** 增量减量运算符**

++	**op++**	**op加1前计算op值**
++	**++op**	**op加1后计算op值**
--	**op--**	**op减1前计算op值**
--	**--op**	**op减1后计算op值**

举例:

```
int i = 1; int j = 0;
 j = i++;   // j = 1, i = 2     ( j = ++i;  // j = 2, i = 2 )
```

•关系和条件运算符

* 关系运算符（返回**true**或**false)**

>	**op1 > op2**
>=	**op1 >= op2**
<	**op1 < op2**
<=	**op1 <= op2**
==	**op1 == op2**
!=	**op1 != op2**

关系运算符通常和条件运算符在一起使用构成判定表达式。

***条件运算符(操作数是true或false,结果返回true或false)**

命题逻辑运算 AND, OR, NOT,XOR

&&	**op1 && op2**	**有条件地计算op2**
||	**op1 || op2**	**有条件地计算op2**
!	**! op**	
&	**op1 & op2**	**op1和op2都被计算**
|	**op1 | op2**	**op1和op2都被计算**
^	**op1 ^ op2**	

注意：

&& and || 造成的逻辑运算表达式“短路(short circuit)”现象

举例：

```
String unset = null;

if ((unset != null) && (unset.length () > 5)){.....do sth. with unset}
// null pointer exception avoid!
```

- 移位和位逻辑运算

* 移位运算

>>	op1 >> op2	算术右移
<<	op1 << op2	算术左移
>>>	op1 >>> op2	逻辑右移

举例:

128 >> 1 gives 64 (00000080H -> 00000040H)
256 >> 4 gives 16 (00000100H -> 00000010H)
-256>> 4 gives -16 (FFFFFF00H-> FFFFFFF0H)
-256>>>4 gives (FFFFFF00H-> 0FFFFFF0H)
128 >> 32 gives 128 (modulo 32 for int type)

- **位逻辑运算(按位布尔运算)**

&	**op1 & op2**	**与**
|	**op1 | op2**	**或**
^	**op1 ^ op2**	**异或**
~	**~op2**	**求补**

```
int x = 1; int y =2; int z =3;
boolean b = (x < y) & ( y < z );

byte x = 0x55; byte y = 0x50;
byte z = (byte) (x & y);
```

```
//BitwiseDemo.java
//在程序中用单个标志表示状态
public class BitwiseDemo {

    static final int VISIBLE = 1;
    static final int DRAGGABLE = 2;
    static final int SELECTABLE = 4;
    static final int EDITABLE = 8;

    public static void main(String[] args)
    {
        int flags = 0;

        flags = flags | VISIBLE;
        flags = flags | DRAGGABLE;
```

```
if ((flags & VISIBLE) == VISIBLE) {
        if ((flags & DRAGGABLE) == DRAGGABLE) {
            System.out.println("Flags are Visible and Draggable.");
        }
    }

    flags = flags | EDITABLE;

    if ((flags & EDITABLE) == EDITABLE) {
         System.out.println("Flags are now also Editable.");
    }
  }
}
```

- 赋值运算操作符

* 基本赋值运算 =

* 复合赋值运算符

+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op

- 其他操作符

?:	等同 **if-else** 语句
[]	用于声明，创建以及访问数组
.	用于构成限制命名
(params)	构成参数表
(type)	强制类型转换(Cast)
new	创建新的对象或数组
instanceof	判别前面的操作数是否为后面操作数的实例

3.3 表达式，语句和语句体(expression,statements and blocks)

数值,变量和操作符 ->表达式 -> 语句 ->语句体

- 表达式

表达式由若干变量，操作符以及方法调用构成。
表达式实现程序的具体功能。

举例:

```
i = 0

System.out.println("hello")
```

操作符优先级

高

后缀运算符	**[] . (params) expr++ expr--**
一元运算符	**++expr --expr +expr -expr ~ !**
创建或类型转换	**new (type)expr**
乘除	*** / %**
加减	**+ -**
移位	**<< >> >>>**
关系	**< > <= >= instanceof**
等值	**== !=**
按位与	**&**
按位异或	**^**
按位或	**|**
逻辑与	**&&**
逻辑或	**||**
if-else条件	**? :**
赋值	**= += -= *= /= %= &= ^= |= <<= >>= >>>=**

低

举例:

```
int k =1; int i =1; int j =0;

j = k + i++;
```

i =? j =?

- 语句

语句构成程序的执行单元。

*** 表达式语句 (以分号结束)**

```
i =0;
++k;
System.out.println("hello");
A  a = new A();
```

*** 声明语句**

```
int i = 0;
```

*** 控制流语句**

- 语句块

语句块由成对的大括号{}中的0条或多条语句构成。

举例:

```
if (Character.isUpperCase(aChar)) {
   System.out.println("The character " + aChar + " is upper case.");
} else {
   System.out.println("The character " + aChar + " is lower case.");
}
```

3.4 控制流语句

循环	**while, do-while , for**
判定	**if-else, switch-case**
例外处理	**try-catch-finally, throw**
分支	**break, continue, label:, return**

- **while and do-while 语句**

```
while (expression) {
    statement
}
```

举例

```
int i = 0;
while (i < 10){
 System.out.println("Are you finished yet?");
 i++;
}
System.out.println("Finally!");
```

```
do {
    statement(s)
} while (expression);
```

举例:

```
int i = 0;
do {
 System.out.println("Are you finished yet?");
 i++;
}while(i<10);
System.out.println("Finally!");
```

- **for语句**

```
for (initialization; termination; increment) {
    statement
}
```

举例:

```
for ( ; ; ) {    //无限循环
    ...
}
for (int i = 0; i< 10; i++){
System.out.println("Are you finished yet?");
}
System.out.println("Finally!");
for(int i = 0, int j = 0; j < 10 ; i++, j++){.....  }
```

新的for循环句法 (Java 1.5后版本加入)

for (variable : collection){}

可用于集合以及数组,避免集合遍历出错。

```
String [] s = { "Java ",  “New",   “Feature"};
  for ( String str : s ) {
        System.out.println( "Array element is: " + str );
  }
```

- **if -else 语句**

```
if (expression) {
    statement(s)
}else {
   statement(s)
}
```

举例:

```
if (Character.isUpperCase(aChar)) {
   System.out.println("The character " + aChar + " is upper case.");
} else {
   System.out.println("The character " + aChar + " is lower case.");
}
```

也可改写为:

```
System.out.println("The character " + aChar + " is "
    +   (Character.isUpperCase(aChar) ? "upper" :
        "lowercase.");
```

注意：
Java 不同于C/C++， if() 必须采用布尔表达式，不能使用数值

if (x) {.... } // x is int　　错误

if (x!= 0) {....}　正确

- **Switch 语句**

```
switch (expression1) {
  case  expression2: statement(s); break;
  case  expression3: statement(s); break;
  default: statement(s); break;
}
```

注意：
expression1 必须为整型以及其兼容类型(short,int,byte or char),
不能是float(double) 或long .

在java 1.7版本后expression1 增加String类型。

举例:

```
int month = 8;
. . .
switch (month) {
   case 1:  System.out.println("January"); break;
   case 2:  System.out.println("February"); break;
   case 3:  System.out.println("March"); break;
   case 4:  System.out.println("April"); break;
   case 5:  System.out.println("May"); break;
   case 6:  System.out.println("June"); break;
   case 7:  System.out.println("July"); break;
```

```
    case 8:  System.out.println("August"); break;
    case 9:  System.out.println("September"); break;
    case 10: System.out.println("October"); break;
    case 11: System.out.println("November"); break;
    case 12: System.out.println("December"); break;
    default: System.out.println("Hey, that's not a valid month!"); break;
}
```

- 异常（例外）处理语句

Java语言提供异常处理机制来帮助程序报告以及处理可能的错误，当一个错误发生时，程序抛出一个异常。

```
try {
   statement(s) //可能抛出特定异常的代码
} catch (exceptiontype name) {
   statement(s) //如果有对应异常抛出时执行的代码
} finally {
   statement(s) //无论有无异常抛出，始终将被执行的代码
 }
```

举例:

```
try{
    startFaucet();
    waterLawn();
}catch(exception e){
 ....
}catch(exception e){....
 ....
}finally{
 stopFaucet();
}
```

异常将在第7章详细讲述

- 分支语句 （在循环结构中使用）

* **break [label]** 用于结束循环
* **continue [label]** 用于结束本次循环
* **label: statement(for, while, do-while)**
* **return**

举例:

```
1   loop: while (true) {
2      for (int i = 0; i< 100; i++) {
3            switch ( c = in.read()) {
4                  case -1:
5                  case '\n':
6                        //jump out of while-loop to line #12
7                        break loop;
8      ...
9                   }
10           }
11    }
12
```

```
13    test: for (...) {
14            ...
15            while(...){
16                    if ( j> 10){
17                            //jumps to next iteration of for-loop at line #13
18                            continue test;
19                            }
20                    }
21            }
```

return 语句

使用return从当前方法中退出

返回一个数值　　return 100;

不返回数值　　return; (方法声明为void.)

3.5 Java注释

Java语言支持下面类型的注释方法

/* text */
编译器忽略 /* 到 */之间的任何内容.

// text
编译器忽略 // 到该行末尾的任何内容

/** documentation */
称为文档注释，与/*...*/相同，并且可以使用JDK的文档工具javadoc利用这些注释来自动生成文档

```
/* hello word app */
package hello;
/**
 *
 * hello world app, my first program! print out Hello,NJUPT!
 */
public class HelloWorldApp {

    /**
     *
     * method main, using System.out stream
     */
    public static void main(String args[]){
     //print out hello njupt
    System.out.println("Hello, NJUPT!\n");
    }
}
```

Java 文档注释可以采用 Javadoc 命令行工具 生成HTML格式的标准Java文档。

javadoc -d mydoc HelloWorldApp.java

也可以使用集成环境工具，如Netbeans在项目文件夹中生成文档。