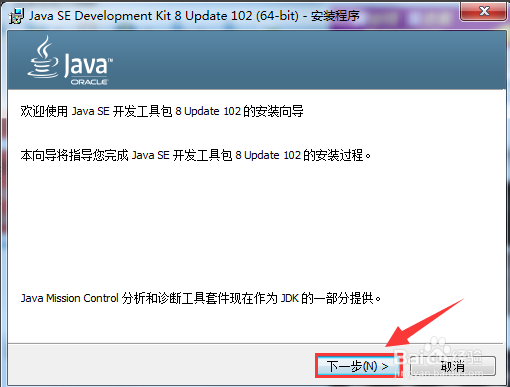
# 第一章：

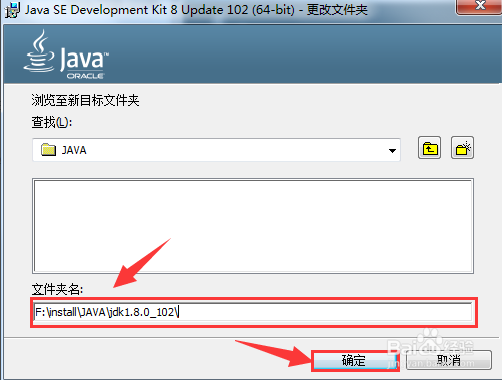
## JDK的安装及环境配置

### 安装：

1. 根据系统类型，我准备安装的是Windows X64，点击jdk-8u102-windows-x64.exe安装程序启动。



1. 这里建议更改安装路径，可以安装至D盘中。注意：此处可以新建一个名为JAVA的文件夹，包含jdk1.8.0\_102文件夹。

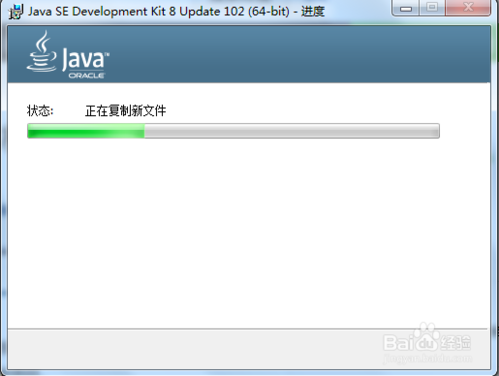
 

1. 下一步为安装jre（java runtime  environment）,注意将jre的安装在与上一步安装的jdk同一文件夹JAVA中。



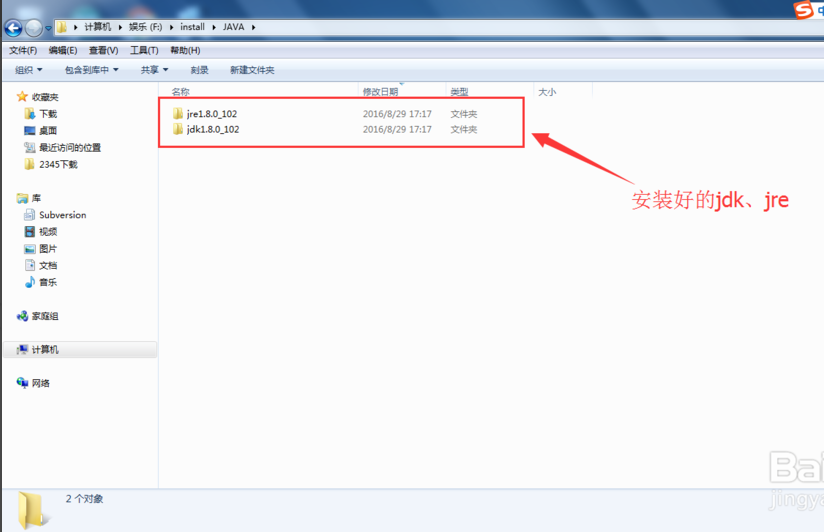
4、然后点击下一步，耐心等待jre安装完成.

安装完成后，直接点击关闭



1. 打开刚安装的路径，可检测是否有jdk和jre两个文件夹。若有，则成功，进入下一步骤：java 环境变量配置。



### 坏境配置：

JAVA\_HOME:

首先配置JAVA\_HOME，点击系统变量下的新建，变量名为JAVA\_HOME，变量值为安装的jdk所在的路径，我的为：F:\anzhuanglujing\java\jdk8，点击确定.

CLASSPATH变量：

与上一步相同，点击新建，变量名为CLASSPATH，变量值为

.;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\tools.jar;  ，注意最前面有个点不能省去，最后面要有英文的分号，点击确定.

PATH变量：

最后配置Path，在系统变量中找到Path并双击它，在变量值的最后加上：

%JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin;最后点击确定

环境变量配置好之后检测下：在cmd 中执行 javac 命令，若出现如下图所示，则安装成功。



## Java的加载与运行

**编译阶段**

**运行阶段**

**Java**

**源代码**

**.class**

**（字节码文件）**

**Java编译器**

**类加载器**

**Java虚拟机（JVM）**

**操作系统（OS）**

**硬件平台**

**网络或本地**

网络或本地

注：\*Java的整个过程需要经历两个阶段：编译阶段、运行阶段。

### 1、Java编译器

从.java源文件到.class字节码文件：是个编译的过程，就是检查代码是否符合Java语法要求的过程，即编译器的作用；

### 2、类加载器

类加载器的作用：每个.class文件可称为一个类，类加载器的作用是将.class文件通过环境变量CLASSPATH找到后装载在JVM中；

### 3、Java虚拟机

Java虚拟机的作用：是将.class文件解释为0101的操作系统能读懂的二进制数据；

### 4、操作系统

操作系统：凭借二进制文件和底层硬件平台进行交互，完成文件的读和写，数据的转移；

## public class和class声明区别详解

1、在一个\*.java的源文件中，只能有一个public class的声明，但是允许有多个class的声明，并且一个class会生成一个.class文件

2、一个.java源文件可以没有public的class，但是public class名字必须与文件名一致

3、public class是公共类的意思，public 是访问修饰符．

**java程序是从一个public类的main函数开始执行的，（其实是main线程），就像C程序是从main（）函数开始执行一样。**

## 标识符：

Java中凡是自己可以起名的都可以称之为标识符，严格区分大小写，由数字、字母、下划线组成;

## 关键字：

Sun在开发Java语言中，定义的一些具有特殊含义的字符序列，所有的关键字都是小写的;

## 数据类型：

Java语言中的字面值：

1、什么是字面值

一眼看上去就知道多少的数据，就是字面值。

1. 字面值本质：

字面值是有数据类型的；

整型 100

浮点型 3.14

布尔型 true/false

字符型 ‘a’

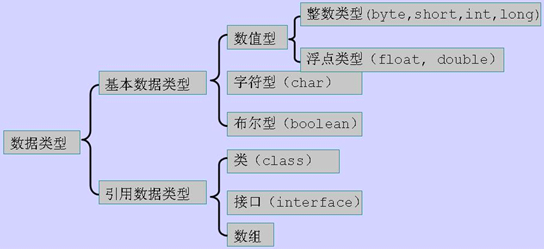
字符串型 “Abc”

字面值就是内存中的一块空间，这块空间有类型、有值。

只要字面值内存无法重复利用;

Java语言中所有的字符都采用单引号（‘’）括起来;

Java语言中所有的字符串都采用双引号（“”）括起来;



### 第一类:逻辑型boolean

Boolean类型数据只允许取值true或false，不可以0或1的整数替代true和false。

### 第二类:文本型char

文本型也可称为字符型：字符长量为用单引号括起来的单个字符，但是每个字符占两个字节（byte）,例如：

System.*out*.println('a');

a 就是字符型，单引号括起来的;

转义字符：

\"：双引号

\'：单引号

\\：反斜线

### 第三类:整数型(byte、short、int、long)

char类型占2个字节

short从-32768到32767

int从-2147483648,到2147483647共10位

long从-9223372036854775808到9223372036854775807共19位

Java各整数类型有固定的表数范围和字段长度,其不受具体操作系统的影响以保证Java程序的可移植性。

Java语言整型常量的三种表示形式

◆十进制整数,如:12,-314,0。

◆八进制整数,要求以0开头,如:012。

◆十六进制数,要求0x或0X开头,如:0x12。

Java语言的整形常量默认为int型，声明long型常量可以后加‘I’或者‘L’，如：

◆int i1=600;∥正确 long l1=8888888L //必须加L否则会出错



**public** **static** **void** main(String[] args) {

// 十进制

**int** i1 = 10; // 最常用

// 八进制

**int** i2 = 010;

// 十六进制

**int** i3 = 0x10;

System.*out*.println(i1); // 10

System.*out*.println(i2); // 8

System.*out*.println(i3); // 16

// 将一个int类型的100字面值赋值给i变量

**int** i4 = 100;

// 将一个int类型的123字面值赋值给long类型的a1

// int类型容量小于long类型

// 以下程序存在类型转换,只不过叫"自动类型转换"

**long** a1 = 123;

// 由于123后面有个L,所以上来就会把123看做long类型。

// 所以以下程序不存在类型转换，只是一个普通的赋值运算;

**long** a2 = 123L;

// 规则：如果整数本身没有超出该类型的取值范围，可以将这个整数直接赋值给该类型

**byte** b1 = 10;

**byte** b2 = 127;

// error 虽然报错,但是可以强制转换(大容量-->小容量),却会损失精度,该方法不建议使用,直接定义为合适的类型即可

**byte** b3 = (byte)128; // 错是因为128已经超出了byte的最大值范围

}

byte-->short-->int-->long-->float-->double

char -->int-->long-->float-->double

Byte、short、char之间不会互相转换，首先先转换成int

在多种类型浑河运算的过程中，首先将所有的数据转换成容量最大的那种，在运算。

### 第四类:浮点型(float、double)

在数学中0到1有无数个浮点数；而计算机是离散的，所以表示的时候有误差，计算机用精度（小数点后几位来表示正确），比较浮点数时a==0.1是不合适的，应该a-0.1==0;如果a是0.1,则即使有误差 a-0.1==0因为a和0.1都被表示为一个有误差的计算机二进制

与整数类型类似,Java浮点类型有固定的表数范围和字段长度,不受平台影响Java浮点类型常量有两种表示形式

◆十进制数形式,例如:3.14 314.0 .314

◆科学记数法形式,如3.14e2 3.14E2 100E-2

Java浮点型常量默认为double型,如要声明一个常量为float型,则需在数后面加f或F,如:

◆double d=12345.6;∥正确 float f=123f;∥必须加f否则会出错



**double** a1 = 1.2;

// 强制类型转换

**float** a2 = (**float**) 1.3;

// 自动赋值运算

**float** a3 = 1.4f;

## 变量：

### Java语言中的变量：

1. java中如何定义变量，语法：

数据类型 变量名；

1. 如何给变量赋值，语法；

变量名 = 值；

1. 变量的本质是什么？

变量的本质就是内存中的一块空间，这块空间有类型、名字、值;

4、在同一个域中，变量可以重新赋值，但是不可以重复声明；

**int** a =10;

**int** a =15; //error

1. Java中的变量必须先声明，再赋值，才能使用，二者缺一不可；

//声明三个变量a1,a2,a3,类型都是int,a1,a2两个变量并没有赋值,只有a3赋值125;

**int** a1,a2,a3 = 125;

System.*out*.println(a3); //a1,a2两个变量并没有赋值，不能访问;

### 变量的作用域：

关于变量的作用域：有效范围、作用范围；

出了大括号就不认识了。

**public** **static** **void** m(){

**int** i =10;

}

**public** **static** **void** main(String[] args) {

//error

System.*out*.println(k);

//无法访问m方法中的k,因为他已经出了大括号，不在同一个域（方法）中;

}

### 成员变量：

如果局部变量的名字和成员变量的名字相同， 要想在该方法中使用成员变量，必须使用关键字this。

**public** **class** VarTest {

// 成员变量

String name = "类体中的变量"; // 类中的成员变量并赋值

**public** **void** spake() {

String name = "方法中的变量"; //在方法speak中定义和成员变量名字相同的name并赋值

System.*out*.println(name);

System.*out*.println(**this**.name); //通过this来访问类中的成员变量

}

//源文件中只能有一个类是public类，并且源文件的名字必须与这个类的名字完全相同，如果没有public类，那么源文件的名字只要和某个类的名字相同就可以了。

**public** **static** **void** main(String[] args) {

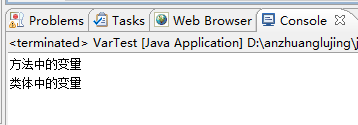
VarTest vt = **new** VarTest();

vt.spake();

}

}

输出结果：



成员变量和局部变量：

区别：

成员变量：

1、成员变量定义在类中，在整个类中都可以被访问。

2、成员变量随着对象的建立而建立，随着对象的消失而消失，存在于对象所在的堆内存中。

3、成员变量有默认初始化值。

局部变量：

1、局部变量只定义在局部范围内，如：函数内，语句内等，只在所属的区域有效。

2、局部变量存在于栈内存中，作用的范围结束，变量空间会自动释放。

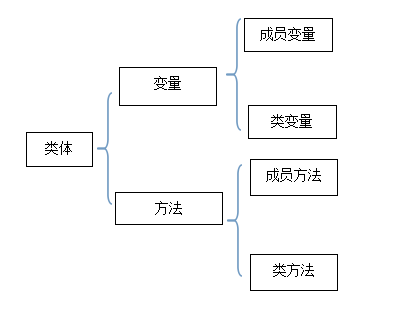
3、局部变量没有默认初始化值

在使用变量时需要遵循的原则为：就近原则

首先在局部范围找，有就使用；接着在成员位置找。

成员变量未赋值时，默认值为0；

成员变量和类变量：



由static修饰的变量称为静态变量，其实质上就是一个全局变量。如果某个内容是被所有对象所共享，那么该内容就应该用静态修饰；没有被静态修饰的内容，其实是属于对象的特殊描述。

不同的对象的实例变量将被分配不同的内存空间， 如果类中的成员变量有类变量，那么所有对象的这个类变量都分配给相同的一处内存，改变其中一个对象的这个类变量会影响其他对象的这个类变量，也就是说对象共享类变量。

**public** **class** MyAdd {

**int** count = 0; //成员变量计数器

**static** **int** *sum* = 0; //静态变量计数器

String name;

MyAdd(String name){

**this**.name = name;

}

**public** **void** myAddMethod(){

count++;

System.*out*.println(name + "调用成员变量后的值：" +count);

}

**public** **void** staticAddMethod(){

*sum*++;

System.*out*.println(name+"调用类后变量的值："+*sum*);

}

**public** **static** **void** main(String[] args) {

MyAdd add1 = **new** MyAdd("add1");

MyAdd add2 = **new** MyAdd("add2");

add1.myAddMethod();

add2.myAddMethod();

add1.myAddMethod();

add1.staticAddMethod();

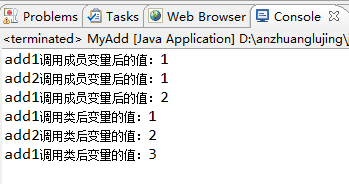
add2.staticAddMethod();

add1.staticAddMethod();

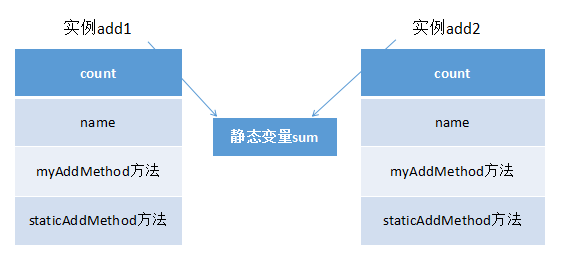
}

}

输出结果：



结构图：



成员变量和类变量的区别：

1、两个变量的生命周期不同

成员变量随着对象的创建而存在，随着对象的回收而释放。

静态变量随着类的加载而存在，随着类的消失而消失。

2、调用方式不同

成员变量只能被对象调用。

静态变量可以被对象调用，还可以被类名调用。

3、别名不同

成员变量也称为实例变量。

静态变量也称为类变量。

4、数据存储位置不同

成员变量存储在堆内存的对象中，所以也叫对象的特有数据。

静态变量数据存储在方法区（共享数据区）的静态区，所以也叫对象的共享数据。

关键字，是一个修饰符，用于修饰成员(成员变量和成员函数)。

特点：

1、想要实现对象中的共性数据的对象共享。可以将这个数据进行静态修饰。

2、被静态修饰的成员，可以直接被类名所调用。也就是说，静态的成员多了一种调用方式。类名.静态方式。

3、静态随着类的加载而加载。而且优先于对象存在。

弊端：

1、有些数据是对象特有的数据，是不可以被静态修饰的。因为那样的话，特有数据会变成对象的共享数据。这样对事物的描述就出了问题。所以，在定义静态时，必须要明确，这个数据是否是被对象所共享的。

2、静态方法只能访问静态成员，不可以访问非静态成员。

因为静态方法加载时，优先于对象存在，所以没有办法访问对象中的成员。

3、静态方法中不能使用this，super关键字。

因为this代表对象，而静态在时，有可能没有对象，所以this无法使用。

什么时候定义静态成员呢？或者说：定义成员时，到底需不需要被静态修饰呢？

成员分两种：

1、成员变量。（数据共享时静态化）

该成员变量的数据是否是所有对象都一样：

如果是，那么该变量需要被静态修饰，因为是共享的数据。

如果不是，那么就说这是对象的特有数据，要存储到对象中。

2、成员函数。（方法中没有调用特有数据时就定义成静态）

如果判断成员函数是否需要被静态修饰呢？

只要参考，该函数内是否访问了对象中的特有数据：

如果有访问特有数据，那方法不能被静态修饰。

如果没有访问过特有数据，那么这个方法需要被静态修饰。

1、成员变量所属于对象。所以也称为实例变量。

静态变量所属于类。所以也称为类变量。

2、成员变量存在于堆内存中。

静态变量存在于方法区中。

3、成员变量随着对象创建而存在。随着对象被回收而消失。

静态变量随着类的加载而存在。随着类的消失而消失。

4、成员变量只能被对象所调用 。

静态变量可以被对象调用，也可以被类名调用。

所以，成员变量可以称为对象的特有数据，静态变量称为对象的共享数据。

//for循环

**for** (**int** i = 0; i < 10; i++) {

System.*out*.println(i); //此处可以访问,在同一个域中。

}

//error

System.*out*.println(i); //此处不能访问i,因为i是属于for循环的，已经出了大括号。

## 运算符：

### 算术运算符：

\* + 求和

- 相减

\* 乘积

/ 相除

++ 自加

-- 自减

% 取余、求模

**public** **static** **void** main(String[] args) {

**int** a = 10;

**int** b = 3;

System.*out*.println(a + b); // 13

System.*out*.println(a - b); // 7

System.*out*.println(a \* b); // 30

System.*out*.println(a / b); // 3

System.*out*.println(a % b); // 1

// 关于++

**int** k = 10;

k++;

System.*out*.println("k=" + k); // 11

// ++出现在变量的后面

**int** m = 10;

**int** e = m++; // ++如果出现在变量的后面,先赋值后自加1

System.*out*.println("e=" + e); // 10

System.*out*.println("m=" + m); // 11

// ++出现在变量的前面

**int** f = 10;

++f;

System.*out*.println("f=" + f); // 11

**int** c = 10;

**int** d = ++c; // ++如果出现在变量的前面,先加1后赋值

System.*out*.println("d=" + d); // 11

System.*out*.println("c=" + c); // 11

**int** z = 10;

//10 在程序里该变量是int类型,但是底层是字符串,z++相当于

**XXX（字符串）** = z++

System.*out*.println("z++ = " + z++); //10

System.*out*.println("z= " + z); //11

**int** y = 10;

System.*out*.println("++y = "+ ++y); //11

System.*out*.println("y=" + y); //11

**int** x =10;

System.*out*.println("x= " + x); //10

System.*out*.println("++x = " + ++x); //11

**int** s =10;

System.*out*.println("s= " + s); //10

System.*out*.println("s++ = " + s++); //10

}

### 关系运算符：

> 大于

>= 大于等于

< 小于

<= 小于等于

== 判断是否相等(基本数据类型)

!= 不等于

**public** **static** **void** main(String[] args) {

**int** a = 10;

**int** b = 10;

// 判断两个基本数据类型是否相等,必须使用"==","="是赋值运算符

**boolean** c = (a == b); // 不知道运算符优先级的可以加括号

System.*out*.println(a > b); // false

System.*out*.println(a >= b); // true

System.*out*.println(a < b); // false

System.*out*.println(a <= b); // true

System.*out*.println(a == b); // true

System.*out*.println(a != b); // false

}

### 布尔（逻辑）运算符：

& 逻辑与 两边都是真,结果就是真

| 逻辑或 一边是真,结果就是真

! 逻辑非 取反, !true-->false

^ 逻辑异或 两边只要不一致就是true

&& 短路与 用法和 & 一样

|| 短路或 用法和 | 一样

布尔运算符两边的算子必须是布尔类型,整个表达式的运算结果也是一个布尔类型。

例：5>3 &(并且) 6>4 -->true

**public** **static** **void** main(String[] args) {

System.*out*.println(5 > 3 & 5 > 4); // true

System.*out*.println(5 > 3 & 5 > 6); // false 逻辑与只要有一个是false，结果就是false

System.*out*.println(5 > 2 | 5 > 6); // true

System.*out*.println(5 > 7 | 5 > 6); // false

System.*out*.println(!**false**); // true

System.*out*.println(**true** ^ **false**); // true

System.*out*.println(**false** ^ **true**); // true

System.*out*.println(**true** ^ **true**); // false

System.*out*.println(**false** ^ **false**); // false

// 关于短路与

**int** a = 10;

**int** b = 10;

// 短路：后面的表达式不执行

// 什么时候发生短路逻辑与？ 第一个结果为false的时候

// 什么时候发生短路逻辑或？ 第一个结果为true的时候

System.*out*.println(a < b && a++ >= b); // false

System.*out*.println(a); // 11

逻辑与、短路与 和 逻辑或、短路或的区别

// & | 符号前后的条件都会判断,都会执行

// && || 符号前面的条件一旦为true，后面 的条件不执行

}

### 赋值类运算符：

基本算术运算符

= 赋值运算符 C = A + B将把A + B得到的值赋给C

扩展算术运算符

+ = 加和 C + = A等价于C = C + A

- = 减和 C - = A等价于C = C - A

\* = 乘和 C \* = A等价于C = C \* A

/ = 除和 C / = A等价于C = C / A

(%)= 取模和 C%= A等价于C = C%A

<< = 左移位赋值运算符 C << = 2等价于C = C << 2

>> = 右移位赋值运算符 C >> = 2等价于C = C >> 2

& = 按位与赋值运算符 C & = 2等价于C = C & 2

^ = 按位异或赋值操作符 C ^ = 2等价于C = C ^ 2

| = 按位或赋值操作符 C | = 2等价于C = C | 2

**public** **static** **void** main(String[] args) {

**int** a = 10;

// 加和

a += 10; // a= a+ 10;

System.*out*.println(a); // 20

// a=20

a -= 5; // a = a-5

System.*out*.println(a); // 15

// a=5

a \*= 3; // a= a \* 3

System.*out*.println(a); // 45

// a=45

a /= 3; // a=a / 3

System.*out*.println(a); // 15

// a=15

a %= 4; // a= a % 4

System.*out*.println(a); // 3

// 基本运算符和扩展运算符的区别:

**byte** i = 10;

// 编译无法通过,运算结果是int类型,前边的类型是byte类型

// i = i + 10;

// 语法：扩展运算符不改变运算结果类型,可能损失精度.

i += 10;

// 此时的1=20 虽然编译通过,但是精度损失严重.btye 类型的最大值127.

i += 1000;

System.*out*.println(i);// -4

}

### 字符串连接运算符：

关于字符串连接运算符

1、做加法运算(两边只要都是字符串类型,一定是加法运算)

2、字符串连接(两边任意一边只要是字符串类型,则一定是字符串连接)

**public** **static** **void** main(String[] args) {

// 字符串连接

System.*out*.println("PI" + 3.14); // 运算结果是字符串类型

**int** a = 10;

**int** b = 12;

System.*out*.println("a+b=" + (a + b)); // a+b=22

System.*out*.println("a+b=" + a + b); // a+b=1012

System.*out*.println(a + "+" + b + "=" + (a + b)); // 10+12=22

}

### 条件（三目）运算符：

boolean表达式?表达式1:表达式2

boolean 表达式的结果是true,则整个表达式的结果就是表达式1的结果,相反则是表达式2的结果。

**public** **static** **void** main(String[] args) {

**boolean** sex = **true**;

**char** c = sex ? '男' : '女';

System.*out*.println(c); // 男

**boolean** status = **true**;

**char** a = status ? 'A' : "disable"; // error disable是字符串类型,而前面是char类型

System.*out*.println(status ? 'A' : "disable"); // println既可以输出char类型也可以输出字符串类型

}

# 第二章：

## 控制语句：

Java控制语句可分为7种

### 控制选择结构语句

If-else：（4种）

//第一种结构

if(boolean表达式){

java语句

}

//第二种结构

if(boolean表达式){

java语句

}else{

java语句

}

//第三种结构

if(boolean表达式){

java语句

}else if (boolean表达式) {

java语句

}else if (boolean表达式) {

java语句

}else if (boolean表达式) {

java语句

}

//第四种结构

if(boolean表达式){

java语句

}else if (boolean表达式) {

java语句

}else if (boolean表达式) {

java语句

}else{

java语句

}

例如：

按年龄判断年龄段;

0—6岁 童年

7—17岁 少年

18—40岁 青年

41—65岁 中年

66岁以后 老年

//第一种写法(初级写法)

**int** age =65;

//if条件一旦不成立就会执行下一个条件

**if** (age>=1 && age<=6) {

System.*out*.println("童年");

//age>=7 && age<=17

} **else** **if**(age<=17){

System.*out*.println("少年");

} **else** **if**(age<=40){

System.*out*.println("青年");

} **else** **if**(age<=65){

System.*out*.println("中年");

}**else** {

System.*out*.println("年老");

}

//第一种写法(高级写法)

**int** age =25;

String grade = "老年";

**if** (age>=1 && age<=6) {

grade="童年";

} **else** **if**(age<=17){

grade="少年";

} **else** **if**(age<=40){

grade="青年";

} **else** **if**(age<=65){

grade="中年";

}

System.*out*.println(grade);//符合以上条件时重新赋值,都不符合时直接输出grade

Switch：

语法：

switch (int类型) {

case int类型:

Java语句;

Java语句;

break;

case int类型:

Java语句;

Java语句;

break; //此处没有break,下面的Java语句也会执行,这种现象叫case穿透现象

case int类型:

Java语句;

Java语句;

break;

default：

Java语句;

}

注：\*break语句、default语句可以没有;

执行原理：

Switch的int 类型跟case的int 类型进行匹配,匹配成功遇到break结束，匹配失败与下一个case的int类型匹配，若都没有匹配成功执行default下的Java语句。

注：\* switch后面只能跟int类型，也只能判断int类型；

**public** **static** **void** main(String[] args) {

**int** a=97;

**switch** (a) {

**case** 99:

System.*out*.println("差一点");

// break; //case穿透 输出结果为 差一点 差两点(遇到break结束)

**case** 98:

System.*out*.println("差两点");

**break**;

**case** 100:

System.*out*.println("正确");

**break**;

**default**:

System.*out*.println("都不是");

}

//case合并

**int** a=77;

**switch** (a) {

//case合并 符合一个case条件即可

**case** 99:**case** 98:**case** 97:

System.*out*.println("90多");

**break**; //

**default**:

System.*out*.println("90以下");

}

例如：

需求：

1. 系统给定学生的成绩，成绩可以带小数。(0 - 100)

2、根据学生成绩判断成绩等级；

[0 - 60) 不及格

[60 - 70) 及格

[70 - 90) 良好

[90 - 100] 优秀

注：\* 包含与不包含

**double** score=100;

**int** grade = (**int**) (score/10); //此处若不处理,只能用if-else实现

**switch** (grade) {

**case** 10:**case** 9:

System.*out*.println("优秀");

**break**;

**case** 8:**case** 7:

System.*out*.println("良好");

**break**;

**case** 6:

System.*out*.println("及格");

**break**;

**default**:

System.*out*.println("不及格");

}

### 循环语句：

for:

语法：

//语法

for (表达式1; 表达式2; 表达式3) {

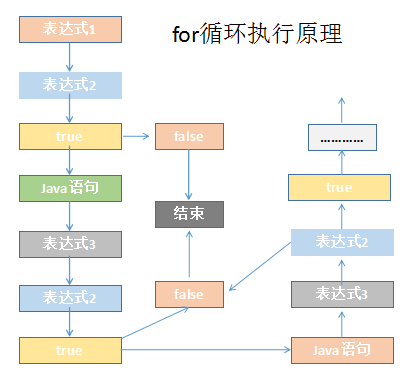
Java语句;

}

//表达式1 是初始化表达式,最先执行,只执行一次;

//表达式2必须是boolean类型的表达式。

//表达式3是操作表达式;



对于for循环，循环条件总比循环体多执行一次。

//例1：

**for** (**int** i = 0; i < 10; i++) {

System.*out*.println(i); //0 1 2 3 4 5 6 7 8 9

}

//例2：

**int** a; //此处的变量a是main方法的变量,i 是for循环的变量

**for** (**int** a1 = 0; a1 < 10; a1 +=2) { // i +=2等价于 i=i+2

System.*out*.println(a1);// 0 2 4 6 8

}

//例3：

**for** (**int** i = 10; i > 0; i -=2) { // i -=2等价于 i=i-2

System.*out*.println(i);// 10 8 6 4 2

}

for循环的嵌套：

所有的控制语句都可以嵌套在一起。

例如：

例1： 九九乘法表

**public** **static** **void** main(String[] args) {

**for** (**int** i = 1; i <= 9; i++) { //外循环用于控制行数

**for** (**int** j = 1; j <= i; j++) {

System.*out*.print(i + "\*" + j + "=" +(j\*i) + " ");

}

System.*out*.println(); //换行的作用

}

}

例2:实现100以内奇数的和

//效率低,但是准确

**public** **static** **void** main(String[] args) {

**int** sum = 0;

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

**if** (i % 2 == 1) {

sum += i;

}

}

System.*out*.println(sum);

}

//效率高,但有可能错误

**public** **static** **void** main(String[] args) {

// 效率高

**int** sum = 0;

**for** (**int** i = 1; i <= 99; i += 2) {

sum += i;

}

System.*out*.println(sum);

}

#### While：

//语法

while (boolean表达式) {

Java语句;

}

该循环执行的次数：0 - N

例1：

**public** **static** **void** main(String[] args) {

**int** i =0;

**while** (i<10) {

i++;

System.*out*.println(i); // 1 2 …… 10

}

}

例2:

**public** **static** **void** main(String[] args) {

**int** i =0;

**while** (i<10) {

System.*out*.println(i); // 0 1 2 …… 9

i++;

}

}

#### do…While…：

//语法

/\*

do {

Java语句

} while (boolean表达式);

\*/

//该循环执行的次数：1 - N 因为他是先执行后判断,至少执行一次

**public** **static** **void** main(String[] args) {

**int** i =0;

**do** {

i++;

System.*out*.println(i); // 1 2 3 ....10

} **while** (i<10);

}

### 改变控制语句顺序：

#### break:

// 1、可以用在switch语句中,结束分支语句.

// 2、break用于完全结束一个循环，跳出循环体执行循环后面的语句。

**public** **static** **void** main(String[] args) {

**for** (**int** i = 0; i < 10; i++) {

System.*out*.println(i);

**if** (i == 5) {

// break; //用于结束循环语句 属于循环级别的

执行结果：0 1 2 3 4 5 hello everyone

return; //用于结束整个main方法 属于方法级别的

执行结果：0 1 2 3 4 5

}

}

System.*out*.println("hello everyone");

}

深入break：

例1：

**public** **static** **void** main(String[] args) {

**for** (**int** i = 5; i > 0; i--) {

**for** (**int** j = 0; j < i; j++) {

**if** (j==2) {

**break**;

}

System.*out*.print(j + " ");

}

System.*out*.println(); //换行作用

}

}

例2：

**public** **static** **void** main(String[] args) {

// 通过以下程序得知break可以指定结束某个循环:指定outerfor循环,通过 break结束outerfor循环体;

outerfor: **for** (**int** i = 5; i > 0; i--) {

**for** (**int** j = 0; j < i; j++) {

**if** (j == 2) {

**break** outerfor;

}

System.*out*.print(j + " ");

}

System.*out*.println(); // 换行作用

}

}

#### continue:

**public** **static** **void** main(String[] args) {

**for** (**int** i = 0; i < 5; i++) {

**if** (i == 3) {

**continue**; //continue是跳过当次循环中剩下的语句,直接进行下一次循环,继续执行。

}

System.*out*.println(i); // 0 1 2 4 没有3

}

}

## 方法：

### 方法的定义：

**什么是方法：**方法就是可以完成特定功能的一段代码片段,并且可以重复利用。

**静态方法与成员方法的区别：**

有**static**的称为静态方法,没有**static**的方法称为成员方法。

//静态方法

**public** **static void** m1**(){**

System.*out*.println("Hello World");

**}**

//成员方法

**public** **void m1(){**

System.*out*.println("Hello World");

**}**

注意：加有static的方法，调用的时候必须采用“类名.方法”的方式调用。没static的不能用此方式调用,可以通过引用的方式调用，“引用.”

**例如：**

**public** **class** MethodTest {

**public** **static** **void** main(String[] args) {

MethodTest.*sunInt*(10, 20);

}

**public** **static** **void** sunInt(**int** a, **int** b) {

**int** c = a + b;

System.*out*.println(a + "+" + b + "=" + c);

}

}

定义方法的语法方法的修饰符列表] 方法的返回值类型 方法名(方法的形式参数列表){

java语句

}

1.[方法的修饰符列表]是可选项。现在暂时先写成:public static

2.方法的返回值类型,可以是Java语言中的任何一种数据类型(基本数据类型和引用数据类型)

3.如果该方法执行结束之后,并没有任何返回值,那么定义方法的时候,返回值类型写:void

4.方法名只要是合法的标识符即可

5.方法的形式参数列表,可以有参数,也可以没有参数,如果有多个参数的话使用迅号隔开。

public static void ml ( ){ }

public static修饰符

void 这个方法执行结束之后没有返回值

m1就是方法名,没有形式参数

public static int m2 (int a,boolean b){ }

public static 修饰符

int 这个方法的返回值类型

m2方法名

int a, boolean b是形式参数列表

6.如果一个方法的返回值类型不是void,那么在方法体中必须使用 return语句来返回数据

return语句一且执行,则方法结束

方法的形式参数列表中起决定作用的是参数的类型;

### 实参和形参：

**public** **class** MethodTest02 {

**public** **static** **void** main(String[] args) {

MethodTest02.*sum*("长者先,幼着后"); // 长者先,幼着后 实际参数(实参)

// 当返回值类型不是void时,可以采用下面的方式调用方法,并打印出方法中的参数。

**int** retValu = MethodTest02.*sum1*(10, 11);

System.*out*.println(retValu);

}

**public** **static** **void** sum(String a) { // age 是形式参数(形参)

System.*out*.println(a);

}

**public** **static** **int** sum1(**int** a, **int** b) { // age 是形式参数(形参)

**int** c = a + b;

**return** c;

}

}

### 方法调用：

**public** **class** MethodTest03 {

**public** **static** **void** main(String[] args) {

sum(); // error 原因是当前类中没有sum()方法,sum()方法存在于类A1中

A1.*sum*(); // Correct 正确

// 在当前类中类名可以省略,可以不需要(类名.方法名)的调用方式了,直接调用方法即可.如下：

*m1*();

}

**public** **static** **void** m1() {

System.*out*.println("m1方法......");

}

}

**class** A1 {

**public** **static** **void** sum() {

System.*out*.println("你好");

}

}

方法调用深入：

**public** **class** MethodTest04 {

**public** **static** **void** main(String[] args) {

**int** i = MethodTest04.*m1*(**true**);

System.*out*.println(i); // 0

// 下面 m1(true)执行的结果是int类型的0,符合m1方法的返回值类型int,所以下面的方法也可实现对m1方法的调用

System.*out*.println(*m1*(**true**)); // 0 方法的深入调用

}

**public** **static** **int** m1(**boolean** a) {

// if (a==true) {

// return 0;

// } else {

// return 1;

// }

// 升级

**if** (a == **true**) {

**return** 0;

}

**return** 1;

}

}

如何结束返回值是void的方法：

//return 只能用在返回值类型为void的方法中

**public** **class** MethodTest05 {

**public** **static** **void** main(String[] args) {

*m1*(0); // 把0作为参数传给i,然后进行for循环

}

**public** **static** **void** m1(**int** i) {

**for** (; i < 10; i++) {

**if** (i == 5) {

// return abc 或 0; //error 对于是void的返回值类型 ,return后面不允许跟任何类型的值

**return**; // 单独的return可以结束返回值类型为void的方法 此处打印的结果是 0 1 2 3 4

// break ; //打印的结果是 0 1 2 3 4 hello word

// 因为break用于完全结束一个循环，跳出循环体执行循环后面的语句。

}

System.*out*.println(i);

}

System.*out*.println("hello word");

}

}

## 方法的重载（Overload）：

举例说明：

不使用方法重载：

**public** **class** MethodTest06 {

/\*

\* 以下方法不使用方法重载,分析缺点。 1、程序员需要记忆多个方法名; 2、代码不美观

\*/

**public** **static** **void** main(String[] args) {

System.*out*.println(Computer.*sumInt*(10, 20));

System.*out*.println(Computer.*sumDouble*(10.0, 20.0));

System.*out*.println(Computer.*sumLong*(10L, 20L));

}

}

**class** Computer {

**public** **static** **int** sumInt(**int** a, **int** b) {

**return** a + b;

}

**public** **static** **double** sumDouble(**double** c, **double** d) {

**return** c + d;

}

**public** **static** **long** sumLong(**long** a2, **long** b2) {

**return** a2 + b2;

}

}

使用方法重载：

**public** **class** MethodTest07 {

/\*

\* 以下方法使用方法重载,分析优点。 1、程序员只需要记忆很少的方法名; 2、代码更加美观

\*/

**public** **static** **void** main(String[] args) {

System.*out*.println(Computer.*sum*(10, 20));

System.*out*.println(Computer.*sum*(10.0, 20.0));

System.*out*.println(Computer.*sum*(10L, 20L));

}

}

**class** Computer{

**public** **static** **int** sum(**int** a, **int** b) {

**return** a + b;

}

**public** **static** **double** sum(**double** c, **double** d) {

**return** c + d;

}

**public** **static** **long** sum(**long** a2, **long** b2) {

**return** a2 + b2;

}

}

方法重载的定义：

在同一个类中可以定义多个方法,它们可以有相同的名字，但必须具有不同的参数，即参数的个数或者类型不同。调用方法时通过传递给它们的不同个数和类型的参数来决定具体使用哪个方法。

什么时候会发生方法的重载：

1. 发生在统一各类中；
2. 方法名相同；
3. 参数列表不同（个数，类型，顺序）；
4. 与方法的返回值类型跟修饰符无关

注1：\* 方法的重载跟方法的返回值类型无关,例如：

**public** **static** **void** m1(){

}

**public** **static** **int** m1(){

}

注2：\* 方法的重载跟方法的访问修饰符无关,例如：

**static** **void** **int** m1(){}

**public** **static** **int** m1(){}

System.*out*.println();中的println就是方法重载，jdk的源码中可以查到

out 是个成员变量，.后面的字符串没有()就是个成员变量，有()的就是方法;

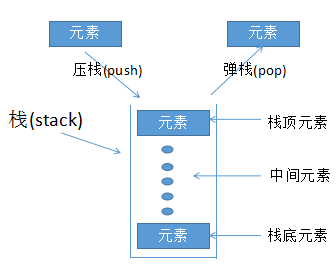
System.*out*是sun提供的PrintStream类型。

Println 是 *PrintStream* 类的方法，该方法已经构成了方法重载。

数据结构：

栈：

栈：也可称为堆栈，只有一个出入口的队列，存储数据的特点：后进先出；



## 方法的执行原理：

1. 方法在调用的时候，才会给该方法在内存中分配空间；

如果这个方法只是定义没有调用，则不会在内存在分配空间。

1. 方法在调用的时候在‘栈’中分配空间。（JVM内存中有一块内存是栈内存）

方法调用其实就是“压栈”，

方法结束其实就是“弹栈”。

**public** **class** MethodTest08 {

**public** **static** **void** main(String[] args) {

**int** i = 100;

*m1*(i);

System.*out*.println("Hello Word");

}

**public** **static** **void** m1(**int** a) {

*m2*(a);

}

**public** **static** **void** m2(**int** b) {

*m3*(b);

}

**public** **static** **void** m3(**int** c) {

System.*out*.println(c);

}

//不会给m4方法分配空间。因为只定义,没调用

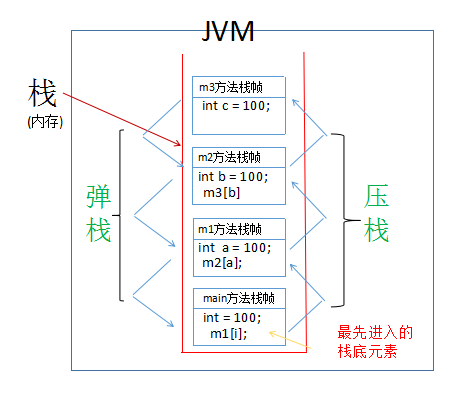
**public** **static** **void** m4(**int** d) {

System.*out*.println(d);

}

}

**方法执行过程：**



注：\* 栈底元素是最先调用的方法。

## 递归：

### 方法的递归调用：

1. 方法的递归调用就是方法自身调用自身；
2. 下面的m1方法执行过程中只有压栈动作,没有弹栈动作,最终会导致栈的内存溢出错误.所以递归必须要有结束条件。

**public** **class** Test {

// 入口

**public** **static** **void** main(String[] args) {

*m1*();

}

// 该方法执行过程中只有压栈动作,没有弹栈动作,最终会导致栈内存溢出错误（StackOverflowError）.

**public** **static** **void** m1(**int** a) {

*m1*();

}

}

//不用递归算出1 ....100 的和;

**public** **class** RecursionTest {

**public** **static** **void** main(String[] args) {

**int** n = 100;

**int** retValue = *sum*(n);

System.*out*.println(retValue);

}

**public** **static** **int** sum(**int** n) {

**int** sum = 0;

**for** (**int** i = 0; i <= n; i++) {

sum += i;

}

**return** sum;

}

}

//用递归算出1 ....5的和;

**public** **class** RecursionTest1 {

**public** **static** **void** main(String[] args) {

**int** n = 5;

**int** retValue = *method*(n);

System.*out*.println(retValue);

}

**public** **static** **int** method(**int** n) {

**if** (n == 1) {

**return** 1;

} **else** {

**return** n + *method*(n - 1);

}

}

// 当n==5的时候,不满足n==1,else中的语句便为:return 5 + method(5-1)=4;

// 当n==4的时候,不满足n==1,else中的语句便为:return 4 + method(4-1)=3;

// 当n==3的时候,不满足n==1,else中的语句便为:return 3 + method(3-1)=2;

// 当n==2的时候,不满足n==1,else中的语句便为:return 2 + method(2-1)=1;

// 当n==1的时候,满足if条件,if中的语句便为:return 1;

// method(5) = return 5 + method(4);

// method(4) = return 4 + method(3);

// method(3) = return 3 + method(2);

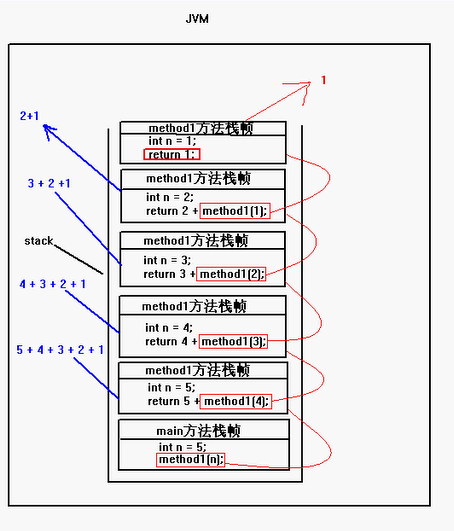
// method(2) = return 2 + method(1);

// method(1) = return 1;

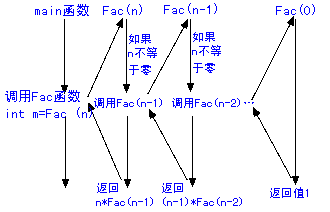
// 由以上五个等式可得：return 5 + 4 + 3 + 2 +1;

}

执行流程图：



递归执行过程图解



计算N的阶乘：

不用递归：

**public** **class** RecursionTest2 {

**public** **static** **void** main(String[] args) {

System.*out*.println(*method*(6)); // 720

}

**public** **static** **int** method(**int** n) {

**int** result = 1;

**for** (**int** j = 1; j <= n; j++) {

result \*= j;

}

**return** result;

}

}

使用递归：

**public** **class** RecursionTest2 {

**public** **static** **void** main(String[] args) {

System.*out*.println(*method*(6)); //此方法也可调用method方法

}

**public** **static** **int** method(**int** n) {

**if** (n == 1) {

**return** 1;

} **else** {

**return** n \* *method*(n - 1);

}

}

}

注：\*//此处的递归会产生多次压栈，而for循环只会发生一次压栈，所以不建议使用递归，除非不得不用的时候（比如查询章节）；

# ：

## 面向对象：

Java是纯面向对象语言；

面向过程与面向对象的区别：（盖饭、蛋炒饭）

为什么会出现面向对象分析方法？

因为现实世界太复杂多变，面向过程的分析方法无法满足。

面向过程？

采用面向过程必须了解整个过程，每个步骤都有因果关系，每个因果关系都构成了一个步骤多个步骤就构成了一个系统，因为存在因果关系每个步骤很难分离，非常紧密，当任何一骤出现问题，将会影响到所有的系统。如:采用面向过程生产电脑，那么他不会分CPU主板和硬盘，它会按照电脑的工作流程一次成型。耦合度高、扩展复杂。

面向对象？

面向对象会将现实世界分割成不同的单元(对象)，实现各个对象，如果完成某个功能，只需要将各个对象协作起来就可以。耦合度低、扩展能力强。

### 面向对象的三大特征：

封装

继承

多态

### 类与对象的概念：

**类：**

类就是现实世界中一堆具有共同特征的事务，进行抽象就形成了类，实际上不存在，大脑抽象的一个结果。

通常根据名词（概念）发现的，如成绩管理系统中的：学生、成绩、班级、课程。

**对象：**

对象是实际存在的个体，类到对象是实例化的过程，对象到类的过程是抽象的过程。

学生——张三；

成绩——78；

班级——六年级3班；

课程——英语；

以上 张三、78、六年级3班、英语他们是具体存在的，称为对象，也叫实例；

也就是一个类的具体化，就是对象或实例。

对象A 和对象B 具有共同特征，经过抽象形成一个类， 对象A到类是抽象的过程，类到对象A是实例化的过程。

对象与对象具有共同特征，但是每个特征对应的具体值不同。

比如：名字 张三 李四

通过对象访问特征（属性）；

为什么对象会成为主流技术，主要就是因为更符合人的思维模式，更容易的分析现实世界。所以在程序中也采用了面向对象的技术，从软件的生命周期来看，基于面向对象可分为三个阶段：

OOA（面向对象的分析）

OOD（面向对象的设计）

OOP（面向对象的编程）————Java就是纯面向对象语言

我们进一步的展开，首先看看学生：

学生：学号、姓名、性别、地址、班级

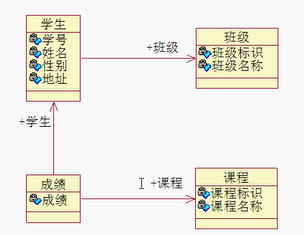
班级：班级代码、班级名称

课程：课程代码、课程名称

成绩：学生、课程、成绩

大家可以看到以上我们分析的都是类的属性：

接下来用简易图形来描述一下，来描述我们的概念（来源于成绩管理系统的概念，来源于灵域的概念，这个领域就是成绩管理系统领域）。



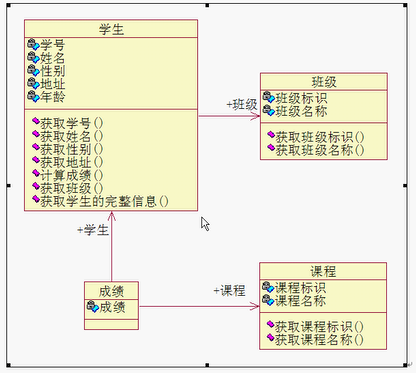
上面的的图形是用Rational Rose工具画制而成，安装可百度搜素步骤（需导入一个钥匙文件）；

上面的图形是一种语言（UML）

UML（统一建模语言）：

（UML，UnifiedModelingLanguage）是面向对象软件的标准化建模语言。UML因其简单、统一的特点，而且能表达软件设计中的动态和静态信息，目前已成为可视化建模语言的工业标准。在软件无线电系统的开发过程中，统一建模语言可以在整个设计周期中使用，帮助设计者缩短设计时间，减少改进的成本。

以上描述的是类的属性，也就是状态信息，接下来做进一步分析。



通过以上分析，大家可以了解到：

类 = 属性 + 方法；

属性来源于类的状态，而方法来源于动作：

以上模型完全可以使用面向对象的语言，如Java来实现。

**public** **class** Student {

//属性

**int** id; //成员变量,实例变量,非静态变量

//id 是对象级别的,必须先有对象才能访问,不能使用类直接访问。

String name;

String sex;

**int** age;

String address;

//方法

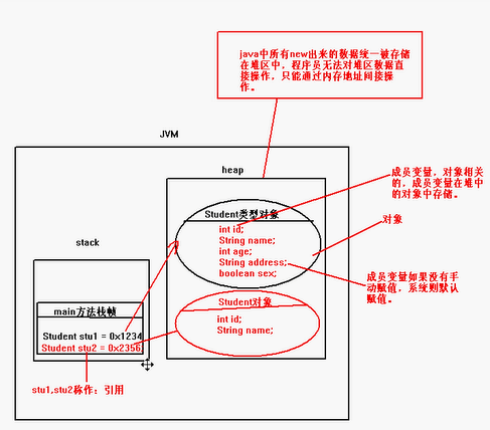
}

### 对象的创建和使用：

Java中所有new (new 关键字的作用是在堆里面开辟空间) 出来的数据统一被存在堆区中，程序员无法对堆区数据直接操作，只能通过内存地址间接操作。

对象：对象是在堆中，存放的是所有new 出来的东西，程序员不能对堆区直接进行操作，只能通过 引用 间接的访问。

引用：保存的是对象在堆中的内存地址



**public** **class** Test {

**public** **static** **void** main(String[] args) {

// 1、创建对象

// stu是一个局部变量,这个局部变量stu是Student类型,也就是一个引用类型。

// stu该局部变量称作引用,引用中保存了对象在推中的内存地址,通过引用间接的访问堆中的对象。

Student stu = **new** Student();

// 一个类可以创建多个对象,如：

Student stu1 = **new** Student();

Student stu2 = **new** Student();

// 2、使用对象

// 访问成员变量语法： 引用.属性(也就是成员变量)

// 读取过程

System.*out*.println("id = " + stu.id); // 0

System.*out*.println("name = " + stu.name); // null

System.*out*.println("sex = " + stu.sex); // null

System.*out*.println("age = " + stu.age); // 0

System.*out*.println("address = " + stu.address); // null

// 赋值

stu.id = 1;

stu.name = "张三";

stu.sex = "男";

stu.age = 19; //此处的值可为负值,虽然编译通过,但是不符合显示逻辑

stu.address = "北京市朝阳区";

// 读取过程

System.*out*.println("id = " + stu.id); // 1

System.*out*.println("name = " + stu.name); // 张三

System.*out*.println("sex = " + stu.sex); // 男

System.*out*.println("age = " + stu.age); // 19

System.*out*.println("address = " + stu.address); // 北京市朝阳区

/\*

\* 关于系统默认值： 基本数据类型： byte、short、int 0

\* long 0L float 0.0f double 0.0d

\* char '\u0000' boolean false

\* 引用数据类型：null

\*/

}

}

### 面向对象的封装性：

**面向对象的封装性指的是：**

1. 属性私有化；
2. 对外提供公开的set get 方法。

**public** **class** Customer {

**private** **int** age; // 为了防止外部程序可以随意访问age,程序不安全,Java语言引入了private,而private修饰的数据只能在本类中访问。

**public** **int** getAge() { // 获取 这是一个成员方法,必须使用"引用."的方式调用

**return** age;

}

**public** **void** setAge(**int** age) { // 赋值 此处是方法，可以自己写Java语句,

**if** (age < 0 || age > 100) {

System.*out*.println("年龄不合法");

**return**;

}

**this**.age = age;

}

// 但是为了外部程序能在需要的时候访问到age,可以对外提供两个公开的方法,由此引入set、get方法。

}

**public** **class** CustomerTest {

**public** **static** **void** main(String[] args) {

Customer cs = **new** Customer();

//赋值

cs.setAge(10);

//读取

**int** a = cs.getAge();

**if** (a==0) { //赋值后判断完成 ,一旦年龄不合法,默认值为0时,终结main方法,不打印。

}

System.*out*.println("年龄为：" + a + "岁");

}

}

### 构造函数（构造方法、构造器、Constructor）：

/\*

关于Java类的构造函数

1、构造方法语法：

[修饰符列表] 构造方法名 (形式参数列表){ //与普通方法的区别是：没有返回值,关键字void的也不能加入,一旦加入就是普通方法。

方法体;

}

2、构造方法的方法名必须与类名一致;

3、构造方法的作用

1)、可以创建类的实例化对象

2)、给成员变量赋值/或者称为初始化成员变量

4、构造方法如何调用

new 构造方法名(实参); 在堆中开辟空间存储对象

5、如果一个类没有提供任何构造方法,系统默认提供无参构造方法。一旦手动创建了构造方法,系统不再提供任何构造方法。

6、成员变量什么时候赋值？

类加载时不会赋值,只有在调用构造方法时才会给成员变量赋值。

\*/

**public** **class** User {

//成员变量,属性,实例变量,非静态变量

String name;

**int** age;

//定义构造方法：(构成了方法的重载)

**public** User() { //表面看似没有代码,实际是有的 name=null age=0

System.*out*.println("User 无参 构造方法执行!");

}

//一旦手动创建了构造方法,系统不再提供任何构造方法。

// public User(String a) {

// System.out.println("User 有参 构造方法执行!");

// }

**public** User(**int** \_age) {

age = \_age;

System.*out*.println("年龄为：" + age );

}

**public** User(String \_name) {

name = \_name;

System.*out*.println("姓名是：" + name );

}

**public** User(String \_name,**int** \_age) {

name = \_name;

age = \_age;

System.*out*.println("姓名是：" + name + " 年龄为：" + age);

}

}

**public** **class** ConstructorTest {

**public** **static** **void** main(String[] args) {

User us = **new** User(); //调用无参构造方法User()

User us1 = **new** User("季枫"); //调用有参构造方法User()

User us2 = **new** User(12);

User us3 = **new** User("季小雨",17);

System.*out*.println(us.name); //null

System.*out*.println(us.age); //0

System.*out*.println(us1.name); //季枫

System.*out*.println(us1.age); //0

System.*out*.println(us2.name); //null

System.*out*.println(us2.age); //12

System.*out*.println(us3.name); //季小雨

System.*out*.println(us3.age); //17

}

}

类中可以出现的东西：

1、成员变量 2、成员方法 3、构造方法 4、静态方法

### 对象和引用：

对象：对象是在堆中，存放的是所有new 出来的东西，程序员不能对堆区直接进行操作，只能通过 引用 间接的访问。

引用：保存的是对象在堆中的内存地址

Java内存的主要划分：

**Java虚拟机管理的内存**

存放所有new 出来的对象，此对象由垃圾收集器收集，垃圾收集器，针对的是堆区

**堆区**

**栈区**

**方法区**

**.class文件**

**装载器**

**其他**

存储的是.class文件的代码，包括所有的方法,静态变量，常量

Java虚拟机中除了堆区和找区、方法区外的其他东西

每调用一个方法，会创建一个栈帧，存放局部变量

拓展\*：

1. 对象和引用的区别；
2. java虚拟机内存划分；
3. Java的垃圾回收机制。

推荐：《深入Java虚拟机》.pdf 可详情了解JVM

方法调用一次，在栈中分配空间

代码，静态变量，常量

如果堆中的对象没有更多的引用指向它，则该对象变成了垃圾，等待垃圾回收器的回收。

JVM

Stack

方法区

Heap

new出来的对象

对象没有更多的引用指向，则变成垃圾。

**例如1:**

**public** **class** ConstructorTest01 {

**public** **static** **void** main(String[] args) {

//us1是引用,保存内存地址指向堆中的对象

User us1 = **new** User();

//程序执行到此处,us1不再指向堆中的对象。

//对象变成了垃圾,等待垃圾回收器的回收。

us1 = **null**;

System.*out*.println(us1.name); //此时编译能通过是因为语法上没有错误,但us1此时的引用没有任何意义,

//是因为User()该构造方法是没有任何形式参数的,所以运行会报空指针异常。

}

}

**例如2：**

**public** **class** ConstructorTest02 {

**public** **static** **void** main(String[] args) {

//使用一个空的引用去访问成员,会出现什么问题？

Student st = **new** Student();

st = **null**;

st.m1(); // 空指针 因为上面已经st = null 引用st不再指向堆区

}

}

**class** Student {

// 成员方法

**public** **void** m1() {

System.*out*.println("这是一个m1方法");

}

}

### 方法调用时的参数传递：

/\*

程序在执行的过程中,参数的传递问题

1、传递的数据是基本数据类型;

2、传递的数据是引用数据类型;

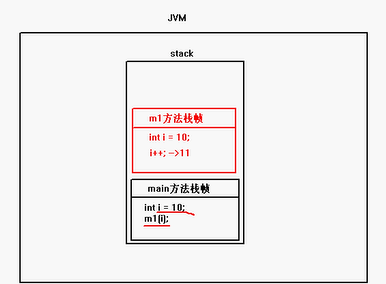
\*/

基本数据类型传递：

//以下程序传递的是基本数据类型

**public** **class** ConstructorTest03 {

**public** **static** **void** m1(**int** i) {



i++;

System.*out*.println("m1--->" + i); // 11

}

// 入口

**public** **static** **void** main(String[] args) {

//局部变量

**int** i = 10;

// 调用

*m1*(i);

System.*out*.println("main--->" + i); // 10

}

}

引用数据类型传递：

//以下程序传递的是引用数据类型

**public** **class** ConstructorTest04 {

**public** **static** **void** m1(Animal a) {

a.age++;

System.*out*.println("m1--->" + a.age); // 11

}

// 入口

**public** **static** **void** main(String[] args) {

// 1、创建Animal 对象

Animal an = **new** Animal(10);

*m1*(an);

System.*out*.println("main--->" + an.age); // 11

}

}

**class** Animal {

**int** age;

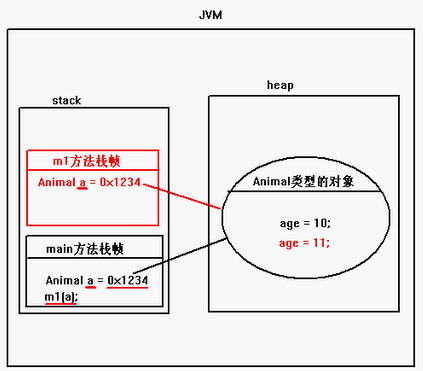
**public** Animal(**int** \_age) {

age = \_age;

}

}

执行流程图解:



### this关键字：

1、什么是this

1)、this是一个引用类型,在堆中的每一个Java对象上都有this,

this保存的内存地址指向自身。

2、this能用在那些地方

1)、this可以用在成员方法中

2)、this可以用在构造方法中

\*/

this保存的内存地址指向自身

**public** **class** ThisTest {

**public** **static** **void** main(String[] args) {

MyDate md = **new** MyDate(2008, 8, 8);

System.*out*.println(md.year + "年" + md.month + "月" + md.day + "日");

MyDate md1 = **new** MyDate(2012, 12, 12);

System.*out*.println(md1.year + "年" + md1.month + "月" + md1.day + "日");

}

}

// 日期

**class** MyDate {

**int** year;

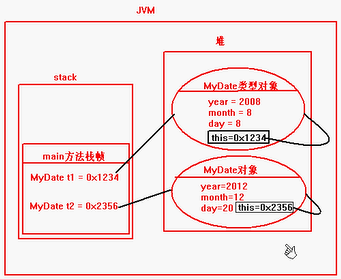
**int** month;

**int** day;

**public** MyDate() { }

**public** MyDate(**int** \_year, **int** \_month, **int** \_day) {

year = \_year;



month = \_month;

day = \_day;

}

}

this在成员方法中：

//this指的就是当前对象。

**public** **class** ThisTest01 {

**public** **static** **void** main(String[] args) {

// 创建对象

Employee e = **new** Employee(1001, "张继科");

e.work();

e.m1();

}

}

**class** Employee {

// 员工编码

**int** empno;

// 员工姓名

String name;

// 构造方法

**public** Employee() {

}

Employee(**int** \_empno, String \_name) {

empno = \_empno;

name = \_name;

}

// 提供一个员工工作的方法

// this用在成员方法中:谁去调用这个成员方法,this就代表谁。

// this指的就是当前对象。

**public** **void** work() {

// System.out.println(name + "在工作"); //this. 可以省略

System.*out*.println(**this**.name + "在工作");

}

**public** **void** m1() {

**this**.m2(); // 第一次调用m2 this.m2() 等价于 m2() this可以省略

m2(); // 第二次调用m2

}

**public** **void** m2() {

System.*out*.println("Test");

}

}

this可以用来区分成员变量和局部变量：

**public** **class** ThisTest02 {

**public** **static** **void** main(String[] args) {

Manager ma = **new** Manager("王珂"); // 调用有参数的构造方法

Manager ma1 = **new** Manager(); // 调用无参数的构造方法

ma1.setName("林姗");

System.*out*.println(ma.getName()); // 王珂

System.*out*.println(ma1.getName()); // 林姗

}

}

**class** Manager {

// 成员变量

**private** String name;

// 此处可以创建构造方法

**public** Manager() {

}

Manager(String name) {

**this**.name = name;

}

**public** String getName() {

**return** name; // 底层是 return this.name;

}

**public** **void** setName(String name) {

**this**.name = name; // 此处的this访问的是成员变量name

}

}