

JavaSE复习笔记

第一章 Java概述

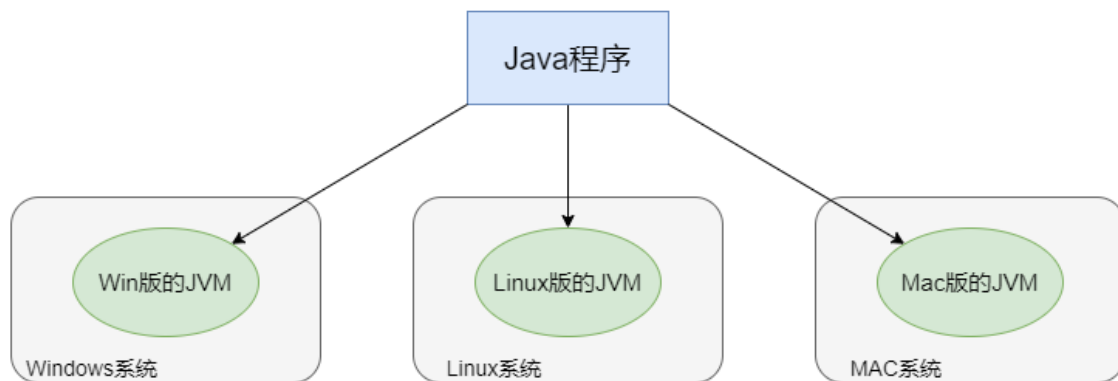
一、计算机语言

1. 机器语言：指令都是二进制的，执行效率高，不宜编写和读取
2. 汇编语言：使用助记符
3. 高级语言：需要转换为二进制，执行效率相对低，易读，易编写

二、Java跨平台原理

跨平台：一处编写，到处运行，在某个系统下编写编译的程序可以在其他系统下执行运行

Java借助JVM实现跨平台。



三、JVM、JRE和JDK的关系

- JVM：Java 虚拟机
- JRE：Java运行时环境=JVM+核心类库
- JDK：Java开发工具集=JRE+开发工具，比如java、javac等

四、JDK的安装与环境变量配置

安装了JDK就可以编写Java程序并运行测试。

为什么配置环境变量？

为了方便在任意路径下可以直接使用java、javac等工具，需要配置环境变量。

如何配置环境变量？

我的电脑-》右键属性-》高级系统设置-》环境变量-》系统环境变量-》

新建：变量名：JAVA_HOME，变量值：JDK安装目录，比如C:\Program Files\Java\jdk1.8.0_202

确定保存

编辑path变量：新建 %JAVA_HOME%\bin

连续确定保存。

测试：打开CMD命令行，输入javac或java不再提示不是内部命令。

五、第一个Java程序

步骤：

1. 编写源代码，保存为后缀为.java的文件
2. 使用javac命令编译源文件，生成后缀为.class的字节码文件
3. 使用java命令运行字节码文件，产生结果



第二章 Java基础语法

一、注释

给程序员开的说明性文字，不会被直接编译执行

1. 单行注释：`// 注释内容`
2. 多行注释：`/* 注释内容 */`
3. 文档注释：`/** 注释内容 */`

二、关键字

Java赋予了特殊含义的一些字符序列（单词），比如public、class、static等

Keyword:

(one of)

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

其中：const、goto是保留字

特殊值：true、false、null。

三、标识符

通常程序员用于命名的字符序列。比如给类、接口、方法、变量等命名时使用的字符序列。

1. 命名规则：必须遵守
 - 由大小写英文字符、数字、下滑线_、美元符号\$组成、
 - 数字不能开头
 - 不能使用关键字

- 严格区分大小写
2. 命名规范：建议遵守
- 见名知意
 - 给类、接口等起名通常遵守大驼峰法则，XxxYyyZzz比如：HelloWorld、FirstName
3. 给方法、变量等起名字通常遵从小驼峰发展，xxxYyyZzz 比如：bookName、productPrice、book
4. 给包起名，通常全部小写，由.隔开，xxx.yyy.zzz 比如：com.atguigu.test
5. 给常量起名字，通常全部由大写字母组成，多个单词用_隔开，XXX_YYY_ZZZ, 比如：MAX_VALUE

四、常量

程序运行过程中其值不会发生改变的量。

分类：

1. 自定义常量：使用final定义

```
final int num = 123; //最终变量或常量，num的值不能被修改
//num = 456; //报错，不能重新赋值
```

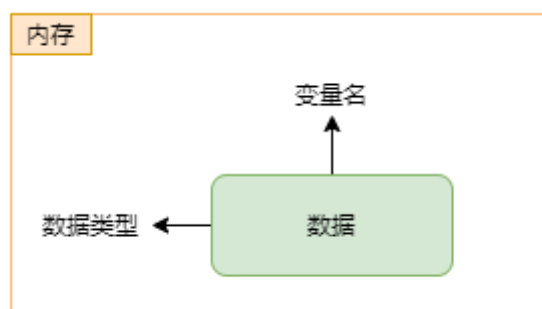
2. 字面量，字面值：

字面量分类	举例
字符串字面量	"HelloWorld"
整数字面量	12, -23
浮点字面量	12.34
字符字面量	'a', 'A', '0', '好'
布尔字面量	true, false
空值字面量	null

五、变量

程序运行过程中其值可以发生改变的量

1. 变量的作用：用于存储一个数据，本质实际是内存中的一块空间



2. 变量的声明：

数据类型 变量名；

```
int num;
```

3. 变量的赋值

变量名 = 变量值;

```
num = 123;  
  
//声明的同时并赋值  
int age = 18;  
//同时声明多个同类型变量并赋值  
int a=1,b=2,c=3;
```

4. 变量的输出

```
System.out.println(num);//123  
System.out.println("age="+age);//age=18
```

5. 变量的使用注意事项

- 变量必须先声明才能使用，否则报错找不到符号
- 变量必须先初始化才能使用，否则报错，未初始化值
- 变量有作用范围，其范围是声明变量所在的大括号内
- 同一个作用范围内容，不能同时声明1个以上的同名变量。

六、计算机如何存储数据

计算机底层存储任意数据都是以二进制形式存储

1. 计算机如何存储整数

计算机以补码形式存储整数。要得到补码需要先知道原码和反码：

原码：最高位为符号位（0表示正数，1表示负数），其他位为数值位。

- 正整数的原码、反码与补码都一致

以一个字节举例：25的原反补码：

```
原码：00011001  
反码：00011001  
补码：00011001
```

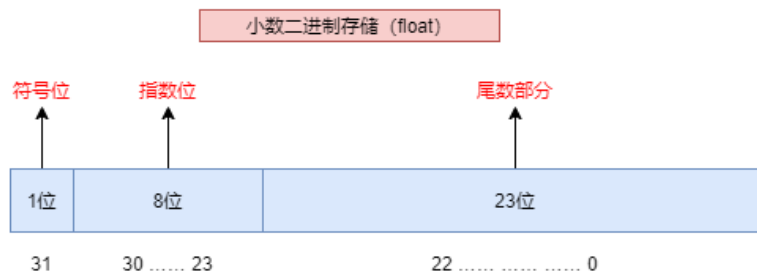
- 负整数的原码、反码与补码都不一样

以一个字节举例：-25的原反补码：

```
原码：10011001  
反码：11100110 原码基础上，符号位不变，其他为按位取反  
补码：11100111 反码基础上，加1
```

2. 计算机如何存储小数

①符号位②指数位③尾数位



符号位：0代表正数，1代表负数

指数位：把十进制小数转换为二进制小数形式，小数点（左右）移动到其左边只保留一位有效数字，小数点移动的位数加上127，即得指数位数值。（左移位数为正数，右移为负数）

尾数部分：小数点右侧的小数部分的二进制表示

计算机无法精确存储浮点数，同样字节的小数存储范围远远大于整数的存储范围

3. 计算机如何存储字符

字符编码表：包含了每个字符与数值的对应关系。

ASCII码表：使用一个字节存储字符

Unicode统一码：使用两个字节存储字符。

七、数据类型

Java是强类型语言，每种类型的数据都定义了数据类型，用于表示数据的存储形式和占用的内存空间。

1. 分类

- 基本数据类型：四类八种
- 引用数据类型：比如类、接口、数组、String等

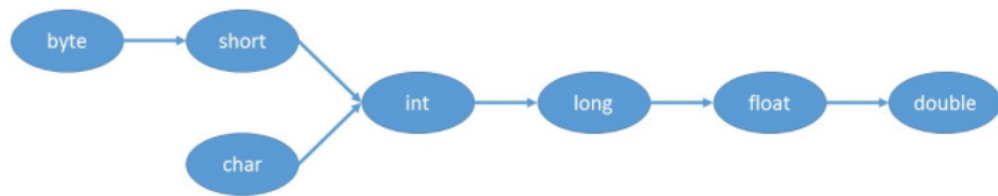
四类八种基本数据类型：

数据类型	关键字	内存占用(字节)	取值范围
整数	byte	1	-2的7次方到2的7次方-1 (-128~127)
	short	2	-2的15次方到2的15次方-1 (-32768~32767)
	int (默认)	4	-2的31次方到2的31次方-1 (-2147483648~2147483647)
	long	8	-2的63次方到2的63次方-1 (-9223372036854775808~9223372036854775807)
浮点数	float	4	负数：-3.402823E38到-1.401298E-45 整数：0.0 正数：1.401298E-45到3.402823E38
	double (默认)	8	负数：-1.797693E308到-4.940656E-324 整数：0.0 正数：4.940656E-324 到1.797693E308
字符	char	2	0-65535
布尔	boolean	1	true , false

2. 数据类型转换

除布尔类型外的7种基本数据类型之间可以相互转换。

基本数据类型按照取值范围从小到大的关系，如图所示：



■ 自动转换（隐式转换）

以下情况会发生自动类型转换

- 数值范围小的类型转换为数值范围大的类型时

```
double d = 10; // int 类型自动转换为 double 类型
int x = 'a'; // char 类型自动转换为 int 类型
```

- 不同类型之间混合运算时，全部提升为其中最大的类型运算，结果也为最大的类型

```
byte a = 1;
int b = 2;
double c = 3;
double d = a+b+c; // 全部提升为 double 进行运算，结果也为 double 型
```

- byte、short和char类型之间混合运算时，全部提升为int类型。

```
byte a = 1;
short b = 2;
char c = 'a';

int d = a+b+c; // 100
```

■ 强制转换

格式：目标数据类型 变量名 = (目标数据类型)要被转换类型的变量或数据

注意：强制类型转换有风险，可能数据有损失，慎重使用。

以下情况需要进行强制类型转换：

- 数值范围大的类型转换为数值范围小的类型

```
int a = (int)12.3; // 12.3 默认为 double 类型，需要强制转换才能存入 int 类型的变量 a 中

byte b = (byte)128; // 默认 128 为 int 类型，需要强制转换才能存入 byte 类型变量 b 中

int x = 10;
byte y = (byte)x; // int 类型的 x，需要强制换行才能存入 byte 类型的变量 y 中
```

- 强制类型提升

```
int a = 1;
int b = 2;
double c = (double)a/b; // 把 a 强制提升为 double 类型，再进行运算
```

- 字符串的特殊类型转换

字符串与任意类型的数据使用+连接，结果都是字符串类型。

```
String s = "hello"+1+'a';
```

八、运算符与标点符号

运算符是用于对数据进行运算的符号

1. 算数运算符

加+、减-、乘*、除/、求余%、自增++、自减--

自增自减运算符的使用

- 单独使用时，前置自增自减与后置效果完全相同

```
int a = 10;
a++; //自增1
++a; //自增1
System.out.println(a); //12
```

- 组合使用时，前置自增自减与后置效果不同

后置自增自减，先使用，再自增自减

```
int a = 10;
int b = a++; //先取出a的值赋值给b，然后a再自增1
//a=11, b=10
```

前置自增自减，先自增自减，再使用

```
int a = 10;
int b = ++a; //先a自增1，再赋值给b
//a=11, b=11
```

2. 赋值运算符

=、+=、-=、*=、/=等

```
int a = 10;
a += 5; //等价于 a = a + 5;

byte x = 5;
//x = x+1; //编译失败，x+1结果为int类型，再赋值给byte类型的x，类型不匹配
x += 1; //正常，底层已经处理了类型转换问题
```

3. 关系运算符

==、!=、>、<、>=、<=等

关系运算符运算后的结果一定是boolean类型

```
System.out.println(10>5); //true
```

4. 逻辑运算符

与&、或|、非!、异或^、短路与&&、短路或||

逻辑运算符左右两边都是boolean类型的数据，运算后结果也是boolean类型。

- 与&的运算逻辑是：**两边都为真，结果才为真。**
- 或|的运算逻辑是：**两边如果有一边为真，结果就为真。**
- **短路与&&、短路或||**的运算逻辑与&、|相同，
- 不同的是：**如果通过左边可以已知最终结果，那么运算符右边不再执行，效率高，推荐使用**

5. 条件运算符、三目运算符

格式：表达式 ? 结果1 : 结果2;

运算逻辑：表达式结果为true，运算的最终结果为结果1，否则是结果2。

```
int age = 18;  
System.out.println(age >= 18 ? "成年了" : "未成年");
```

6. 位运算符（了解）

按照二进制位进行运算的符号。位运算效率高。


按位与&，按位或|，按位取反~，按位异或^

左移<<，左移n位相当于乘以2的n次幂。

右移>>，二进制位上的数值右移，左边最高位原来是1补1，是0补0；右移n位相当于除以2的n次幂，除不尽，向下取整。

无符号右移>>>，二进制位上的数值右移，左边最高位全部补0。

7. 运算符的优先级

. ()	<div>高</div>  <div>低</div>
++ -- ~ !	
* / %	
+ -	
<< >> >>>	
< > <= >= instanceof	
== !=	
&	
^	
&&	
? :	
= *= /= %=	
+= -= <<= >>=	
>>>= &= ^= =	

第三章 流程控制语句

一、键盘录入

从键盘接收数据，使用`java.util.Scanner`键盘扫描器

1. 使用步骤示例：

```
import java.util.Scanner; //导包
public class Demo {
    public static void main(String[] args) {
        //1. 创建扫描器
        //java.util.Scanner in = new java.util.Scanner(System.in); //创建扫描器，存入变量 in
        Scanner in = new Scanner(System.in);
        //2. 提示信息
        System.out.println("请输入数据：");
        //3. 接收整数
        int num = in.nextInt();

        System.out.println(num);
    }
}
```

2. Scanner接收其他类型数据

```
in.nextShort(); //接收短整型
in.nextByte(); //接收字节类型
in.nextLong(); //接收Long类型
in.nextFloat(); //接收float类型
in.nextBoolean(); //接收布尔类型
//.....
in.next(); //接收字符串。只接收空白字符之前的内容。
in.nextLine(); //接收字符串。接收一个行字符串。遇到回车键结束接收。
//注意使用nextLine()时，前面不要使用其他方式接收数据。

//接收字符类型
in.next().charAt(0); //接收一个字符
```

二、流程控制语句分类

流程控制语句可以实现对代码执行流程的控制。

1. 顺序结构：代码自上而下逐行执行。
2. 分支结构-选择结构：根据条件，选择执行某一些代码。
3. 循环结构：重复执行某一些代码。

三、分支结构语句

1. if语句

格式一：

```
if(布尔表达式){
    代码块;
}
```

格式二：

```
if(关系表达式) {  
    语句体1;  
}else {  
    语句体2;  
}
```

格式三：

```
if (判断条件1) {  
    执行语句1;  
} else if (判断条件2) {  
    执行语句2;  
}  
...  
}else if (判断条件n) {  
    执行语句n;  
} else {  
    执行语句n+1;  
}
```

if语句的不同格式之间可以嵌套使用，不建议嵌套太深，易读性下降。

2. switch语句

格式：

```
switch(表达式){  
    //小括号内的结果必须是一个常量值，而且类型只能是6种之一：byte、short、  
    char、int、String、枚举  
    case 常量值1://case后的常量值不能重复。  
        语句块1;  
        break;//跳出switch语句，如果不加break，会继续执行下一个case的代码，直到break  
        或全部执行完。  
    case 常量值2:  
        语句块2;  
        break;  
    ...  
    default://缺省分支，当以上case都不匹配时，执行此分支。  
        语句块n+1;  
        break; //最后一个break可以省略  
}
```

执行流程：如果小括号内的结果与某个case后的常量值匹配，那么执行这个case后的代码，遇到break跳出switch语句。

3. if语句与switch语句的比较

if语句的条件是boolean值，可以判断常量或范围，switch语句的条件是一个常量值。

如果条件是常量值。if语句和switch语句可以相互转换。

如果条件是常量值并且个数超过3个，建议使用switch语句，效率高。

四、循环结构语句

1. for循环格式：

```
for(初始化语句①; 循环条件语句②; 迭代语句③){
    循环体语句④
}
//执行流程: 1234 234 ....2

for(;;){
    循环体语句块; //如果循环体中没有跳出循环体的语句, 那么就是死循环
}
```

```
for(int i=0;i<5;i++){
    System.out.println(i);
}
//输出: 0 1 2 3 4
```

2. while循环

```
while (循环条件语句①) {
    循环体语句②;
}
//执行流程: 121212....1
while(true){
    循环体语句; //如果此时循环体中没有跳出循环的语句, 也是死循环
}
```

```
int i = 0;
while(i<5){
    System.out.println(i);
    i++;
}

//输出: 0 1 2 3 4
```

3. do...while循环

```
do {
    循环体语句①;
} while (循环条件语句②);
//执行流程: 1212121....2
```

```
int i = 0;
do{
    System.out.println(i);
    i++;
}while(i<5);
//输出结果: 0 1 2 3 4
```

4. 循环语句的区别:

1. 从循环次数角度分析

- do...while循环至少执行一次循环体语句
- for和while循环先循环条件语句是否成立, 然后决定是否执行循环体, 至少执行零次循环体语句

2. 从循环变量的生命周期角度分析

- for循环的循环变量在for()中声明的，在循环语句结束后，不可以被访问；
- while和do...while循环的循环变量因为在外边声明的，所以while和do...while结束后可以被继续使用的；

3. 如何选择

- 遍历有明显的循环次数（范围）的需求，选择for循环
- 遍历没有明显的循环次数（范围）的需求，循环while循环
- 如果循环体语句块至少执行一次，可以考虑使用do...while循环
- 本质上：三种循环之间是可以互相转换的，都能实现循环的功能

4. 三种循环结构都具有四要素：

- （1）循环变量的初始化表达式
- （2）循环条件
- （3）循环变量的修改的迭代表达式
- （4）循环体语句块

5. 死循环比较

- for(;;){循环体} ，除循环体外不需要执行其他语句，性能略高
- while(true){ 循环体}，除循环体外还需要执行小括号里的表达式

5. 关键字break和continue

- break
 - 在switch语句中，用于跳出switch语句
 - 在循环语句中，用于跳出当前循环体

```
for(int i=0;i<5;i++){  
    if(i==3)  
        break;  
    System.out.print(i);  
}  
//输出: 0 1 2
```

- continue
 - 用在循环语句中，用于结束本次循环，直接开始下一次循环

```
for(int i=0;i<5;i++){  
    if(i==3)  
        continue;  
    System.out.print(i);  
}  
//输出: 0 1 2 4
```

6. 循环嵌套

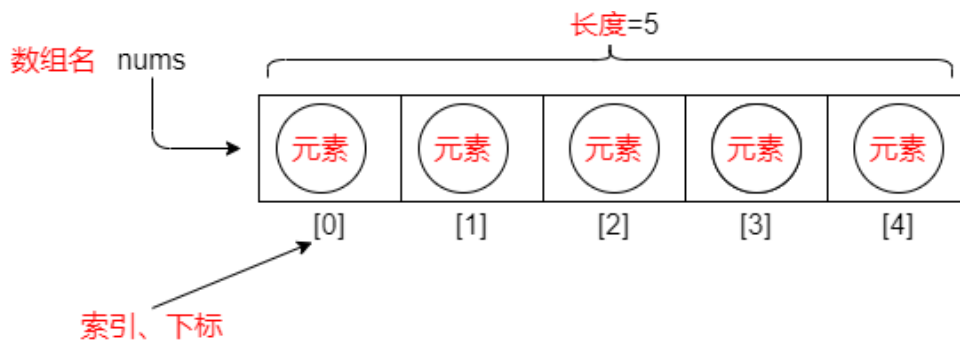
for循环嵌套

```
int n = 5;
//外循环控制行数
for(int i=0;i<n;i++){
    //内循环控制列数
    for(int j=0;j<n;j++){
        System.out.print("*");
    }
    System.out.println();
}
```

第四章 数组

一、数组概述

1. 数组：是个数据容器，可以存储多个同类型的数据
2. 相关概念：
 - 数组名：数组的名字
 - 元素：数组中存储的数据
 - 索引、下标：从0开始的一个编号，用于访问每个元素
 - 数组的长度：数组可以存储的最大元素个数



3. 数组的特点
 - 数组一旦创建，长度不可改变。
 - 数组在内存中一块连续的内存空间，用于存储的元素类型必须一致
 - 数组通过索引访问元素，效率高。

二、数组的声明与初始化

1. 数组的声明

格式：元素的数据类型[] 数组名；

```
int[] ages; //声明整型数组，元素必须是int类型
String[] names; //声明String类型数组，元素是String类型
int a[]; //不推荐的声明格式
```

2. 数组的初始化（开辟内存空间，并初始化值）

- 静态初始化

特点：程序员指定元素的初始值，长度由系统决定（不能指定）

格式：数据类型[] 数组名 = new 数据类型[] {元素1, 元素2, 元素3,};

```
ages = new int[] {18,19,20,18};
int[] arr = new int[] {11,22,33};
//简化写法
int[] arr = {11,22,33};
```

- 动态初始化

特点：程序员指定数组的长度，元素的初始值由系统给出（默认初始值）

格式：**数据类型[] 数组名 = new 数据类型[长度];**

```
names = new String[3]; //先声明，再初始化
String[] arr = new String[3]; //声明的同时并初始化
```

三、数组元素的访问与遍历

1. 数组元素的访问

通过数组名和索引的组合来直接访问数组元素

- 获取指定索引位置的元素，格式：数组名[索引值]
- 设置指定索引位置的元素值，格式：数组名[索引值] = 元素值

```
int[] arr = {11,22,33};
int x = arr[0]; //获取元素
System.out.println(arr[1]); //获取元素

arr[0] = 100; //设置元素值
```

2. 数组的遍历

逐个获取数组的每个元素

数组的长度：**数组名.length**

```
for(int i=0;i<arr.length;i++){
    System.out.println(arr[i]);
}
```

四、数组的默认初始值

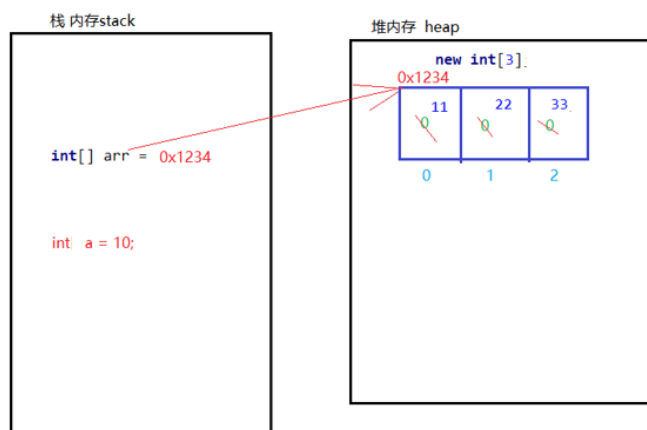
当动态初始化数组时，系统给出数组元素的默认初始值

数组元素类型	元素默认初始值
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0
char	0 或写为:'\u0000'(表现为空)
boolean	false
引用类型	null

五、数组的内存分析

```
public class Demo5 {
    public static void main(String[] args) {
        int[] arr = new int[3];
        System.out.println(arr);
        //[1b6d3586 内存地址转换而来的一个字符串
        System.out.println(arr[0]); //0
        arr[0] = 11;
        arr[1] = 22;
        arr[2] = 33;
        System.out.println(arr[0]); //11
    }
}
```

数组是引用数据类型
int[] arr;
arr是引用数据类型变量



六、数组的常见算法

1. 统计相关算法：求和、求平均值、统计个数等
2. 查找相关算法：顺序查找指定元素及索引、查找最大、最小值及索引位置等
3. 排序算法：冒泡排序、直接选择排序
4. 数组的反转：反转数组中的元素，前后倒置
5. 数组的扩容：当数组容量不足，需要扩容。本质创建长度更大的新数组
6. 数组元素的插入：先扩容，再插入新元素，插入位置之后的每个元素需要后移一位。
7. 数组元素的删除：删除指定位置的元素，指定位置后的每个元素需要前移一位。
8. 二分查找、折半查找，前提是元素是排序后的。

七、数组工具类Arrays

java.util.Arrays类提供了一些方便操作数组的功能。

```
int[] arr={12,33,14,5,61,16,13};
//排序
Arrays.sort(arr);
//二分查找:前提排序的数组
int index = Arrays.binarySearch(arr,12);
//把数组变成字符串--包含了所有元素
String s = Arrays.toString(arr);
//复制数组
int[] arr2 = Arrays.copyOf(arr,10);
```

八、二维数组

1. 二维数组的理解：本质上是一个元素为一维数组的数组
2. 二维数组的声明

格式：数据类型[][] 数组名；

```
int[][] arr;
int[] arr[]; //不推荐
```

3. 二维数组的初始化

- 静态初始化

```
int[][] arr = new int[][]{{1,2,3},{4,5,6},{7,8,9}};
//简化
int[][] arr = {{1,2,3},{4,5,6},{7,8,9}};
```

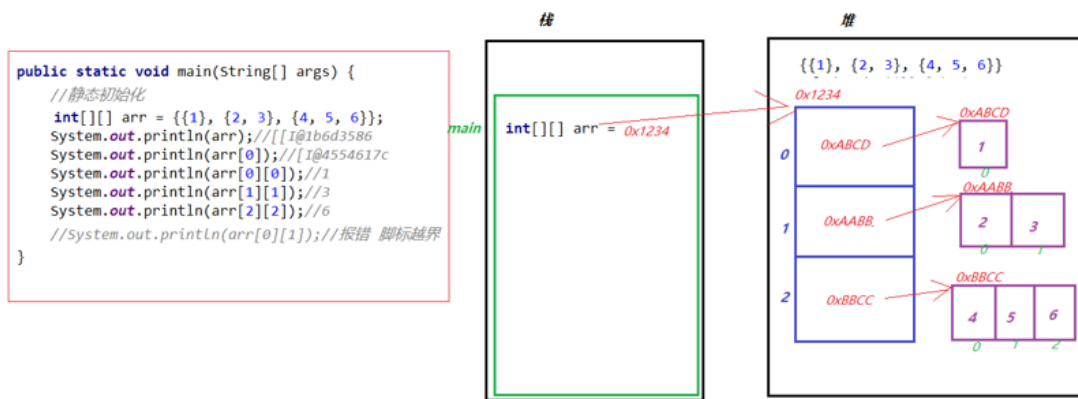
- 动态初始化

```
int[][] arr = new int[5][3]; //表示二维数组长度为5，其元素为一维数组，每个一维数组的长度为3
//-----
int[][] arr = new int[3][]; //表示二维数组长度为3，其元素为null
arr[0] = new int[]{1,2,3};
arr[1] = new int[]{4,5};
```

4. 二维数组元素的访问与遍历

```
int[][] arr = {{1,2,3},{4,5,6},{7,8,9}};
//外循环遍历二维数组
for(int i=0;i<arr.length;i++){
    //内循环遍历每个一维数组
    for(int j=0;j<arr[i].length;j++){
        System.out.print(arr[i][j]+" ");
    }
    System.out.println();
}
```

5. 二维数组的内存分析



6. 二维数组的常见异常

```
int[][] arr = new int[3][];  
arr[0] = new int[]{1}  
system.out.println(arr[0][1]); //脚标越界异常  
system.out.println(arr[1][1]); //空指针异常
```

第五章 面向对象基础（上）

一、面向对象思想概述

面向对象是一种编程思想，参照现实生活中的事物来设计程序。现实生活中的万事万物都可以通过两个维度来描述：静态特征（属性）和动态特征（行为功能），程序中主要关注的内容也是两方面：数据和功能，在程序中用类来描述事物，类中包含了属性和方法。

二、类和对象

1. 类与对象的概念与理解

- 对象：具体特定属性和功能的事物
- 类：一类具体共同属性和功能的事物的集合
- 类与对象的关系：
 - 类是抽象的，是对象的模板，通过类创建对象(这个过程称为实例化)
 - 对象是具体的个体

2. 类的定义格式

```
【修饰符】 class 类名{  
    //类的成员：  
    //静态特征-属性  
    //动态特征-行为功能  
}
```

```
public class Student{  
    //成员变量  
    String name;  
    //成员方法  
    public void study(){  
    }  
}
```

3. 对象的创建

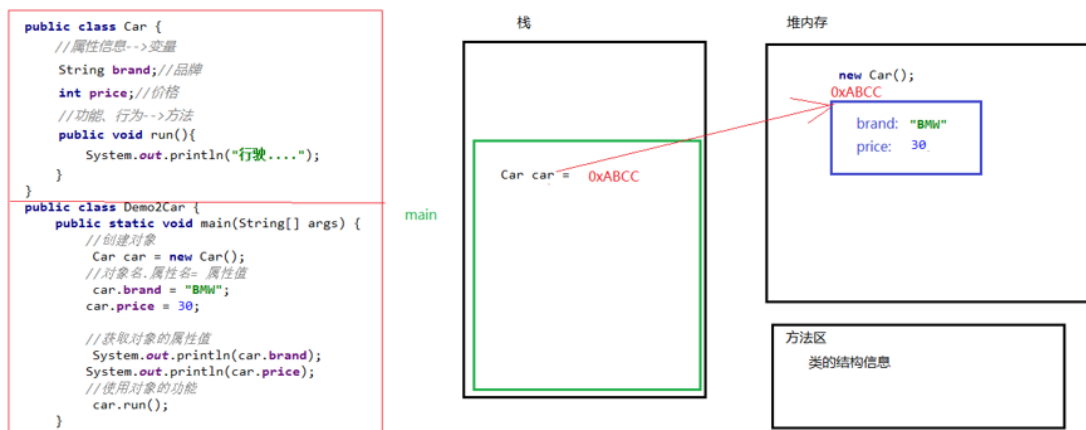
```
类名 对象名 = new 类名();
```

```
Student s = new Student();
Student s1;
s1 = new Student();
```

4. 对象的使用

```
Student s = new Student();
//对象.属性
s.name = "tom";
System.out.println(s.name);
//对象.功能
s.study();
```

5. 对象的内存分析



三、包

包的在操作系统中本质其实就是文件夹（目录），比如：`com.atguigu.package`

1. 包的作用

1. 可以避免类重名：有了包之后，类的全名称就变为：包.类名
2. 分类组织管理众多的类，比如java.lang包下放最核心的类，java.util包下放一些工具类等
3. 可以控制某些类型或成员的可见范围（配合权限修饰符实现）

2. 包的声明：

在定义类时，需要声明类所在的包。使用关键字package，声明语句必须在类文件的第一行

3. 访问不同包下类的方式：

跨包实现类时，需要导包，使用关键字import。但是java.lang包下的类使用不用导包。

四、成员变量

用来描述类的属性信息

1. 变量分类

- 局部变量：定义在方法内、方法参数上等局部区域内。

- 成员变量：定义类的直接成员位置，与方法并列的

- 类变量-静态变量：有static修饰（后面再讲）
- 实例变量-非静态变量：没有static修饰

2. 成员变量的声明

```
class Student{  
    //成员变量  
    String name;//实例变量  
    int age;//实例变量  
    static int x;//类变量  
}
```

3. 实例变量的特点

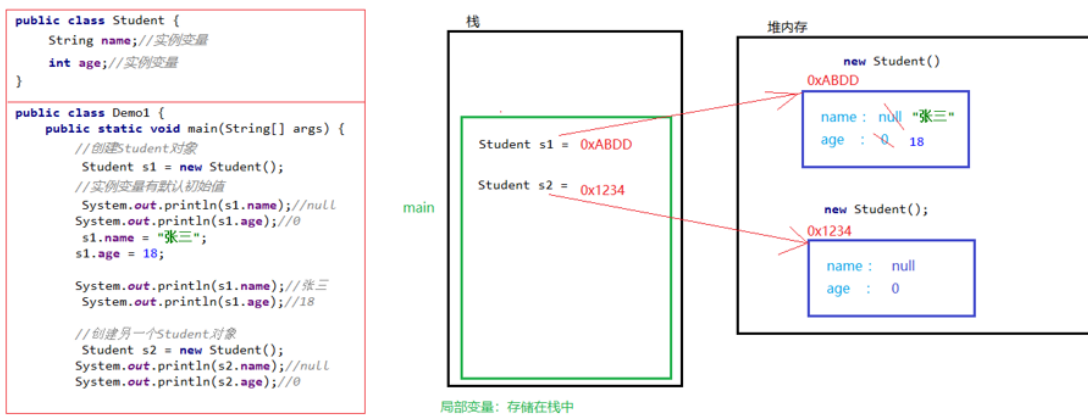
- 每个对象独有一份实例变量（值）
- 必须通过对象才能访问实例变量
- 实例变量都有默认初始值（类同数组元素的默认初始值）

4. 实例变量的访问

对象.实例变量

```
Student s = new Student();  
//必须通过对象来访问实例变量  
s.name = "tom";  
System.out.println(s.name);
```

5. 实例变量的内存分析



6. 实例变量与局部变量的区别

	实例变量	局部变量
声明的位置	直接声明在类的成员位置	声明在方法体中或其他局部区域内（方法声明上，构造方法，代码块等）
修饰符	public、private、final等	不能使用访问权限修饰符，可以使用final
内存加载位置	堆	栈
初始化值	有默认初始化值	无默认初始化值
生命周期	同对象的生命周期	随着方法的调用而存在，方法调用完毕即消失

五、方法

1. 概念理解：方法是一个特定功能的封装。这样做的目的，方便重复利用代码。

2. 方法分类：

- 类方法-静态方法：有static修饰（后面再讲）
- 实例方法-非静态方法：没有static修饰

3. 方法的声明

方法必须声明在类的成员位置，而且方法不能嵌套定义，所有的方法都是并列关系。

```
public class 类名{  
    //在类的成员位置-声明方法  
    【修饰符】 返回值类型 方法名(【参数列表】)【throws 异常列表】{  
        //方法体：用于实现功能  
        【return 返回值】  
    }  
  
}
```

4. 格式说明

- 一个完整的方法 = 方法头 + 方法体。
 - 大括号内为方法体，主要来实现功能；
 - 大括号之前的内容是方法头，也称为方法签名。通常调用方法时只关注方法头即可。方法头包含5部分，有些部分可以缺省。
- **修饰符**：修饰符后面详细讲，例如：public，static等都是修饰符
- **返回值类型**：表示方法运行的结果的数据类型，与“return 返回值”搭配使用
 - 无返回值：void
 - 有返回值：可以是任意基本数据类型和引用数据类型
- **方法名**：给方法起一个名字，要符合标识符的命名规则，尽量见名知意，能准确代表该方法功能的名字
- **参数列表**：方法内部需要用到其他方法中的数据，需要通过参数传递的形式将数据传递过来，可以是基本数据类型、引用数据类型、也可以没有参数，什么都不写
- throws 异常列表：可选，在异常章节再讲
- **方法体**：特定功能的代码
- **return**：结束方法，可以返回方法的运行结果
 - 可以返回不同类型的数据，对应匹配的**返回值类型**。

- 如果方法无返回值，可以省去return，并且返回值类型为**void**

5. 示例

```
public class Student{  
    //实例方法：  
    //无参数，无返回值  
    void study(){  
        System.out.println("学习...")  
    }  
    //有参数，无返回值  
    void testParam(int a,String b){
```

```

    }
    //有参数，有返回值
    int testReturn(int a,int b){
        int c = a+b;
        return c;
    }
}

```

6. 实例方法的调用

对象.方法(【实参列表】)

```

Student s = new Student();
//调用方法
s.study();
s.testParam(123,"abc");
int num = s.testReturn(11,22); //此方法有返回值，可以使用变量接收返回的数据
System.out.println(s.testReturn(1,2)); //此方法有返回值，也可以直接输出其结果
s.testReturn(11,44); //如果不接受返回值，后面代码无法再使用此返回值
//其他代码...

```

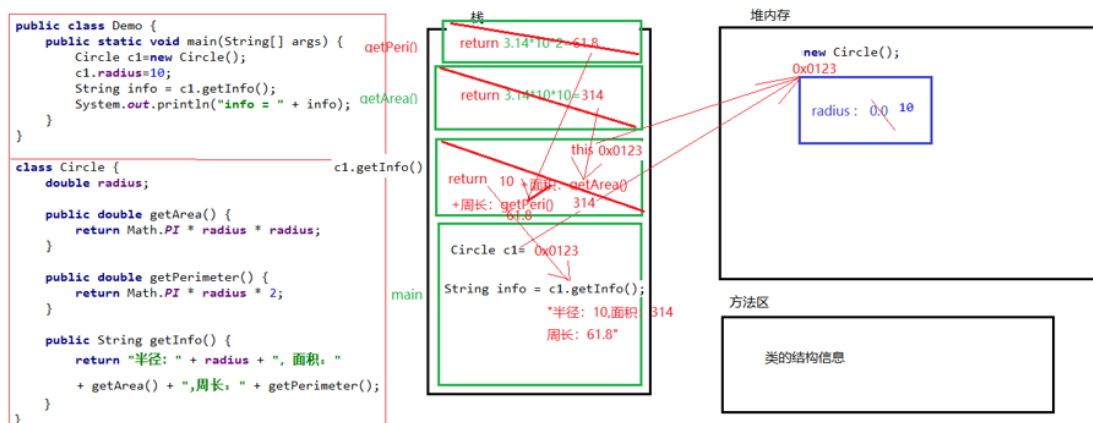
形参：方法声明上的参数列表，是形式上的参数。

实参：调用方法时传递的实际参数。

注意：

- 方法调用时实参的个数、类型及类型顺序必须与形参列表一致。
- 方法不调用，不执行，调用异常执行一次。
- 方法如果有返回值，那么可以使用变量接收此值，或直接使用输出语句输出此值。如果没有使用变量接收或直接输出，那么返回值将丢失。

7. 方法的调用内存分析



方法被调用时（执行时），入栈（在栈内存中为每个方法开辟内存，主要存储方法内的局部变量）
方法执行完毕后，立即释放所占的栈内存空间，如果有返回值，返回到方法的调用处。

六、方法参数的值传递问题

调用方法时，传入的实参会影响形参，那么形参的改变会影响实参吗？

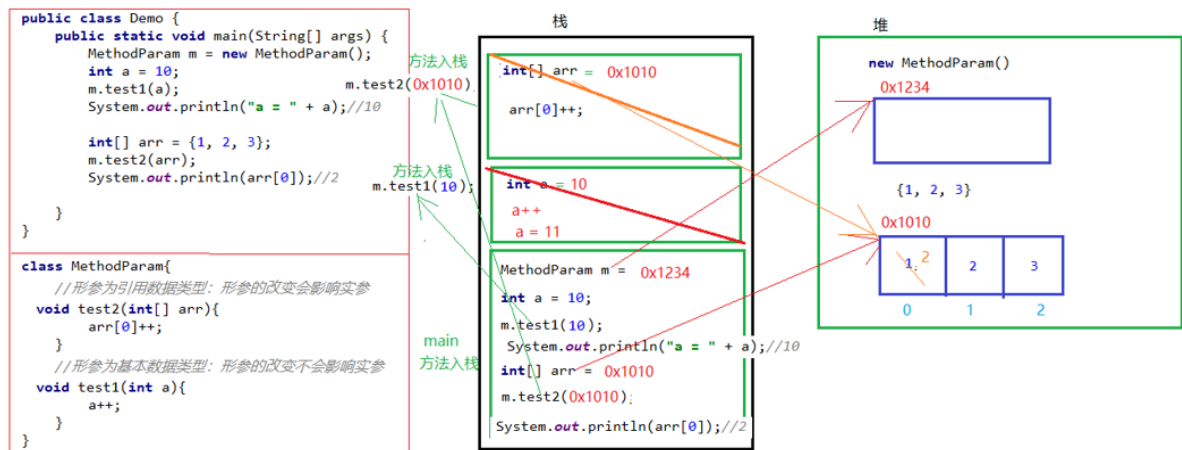
1. 当形参为基本数据类型时，形参的改变不会影响实参。
2. 当形参为引用数据类型时，形参的改变可能会影响实参。
3. 特殊情况：当形参为String或包装类类型时形参的改变不会影响实参。

```

public void change1(int a){
    a++;
}
public void change2(int[] arr){
    arr[0]++;
}
//--调用方法--
int a = 10;
change1(a);
System.out.println(a); //10

int[] arr={1,2,3}
change2(arr); //arr数组元素被修改了
System.out.println(arr[0]); //2

```



七、方法重载Overload

1. 概念理解：同一个类中，存在多个同名方法，但参数列表不同，这就是方法重载。与方法的返回值无关。（参数列表不同指的是：参数的个数、类型或类型顺序不同）
2. 示例：

```

public class MyTools{
    public int add(byte a,byte b){
        return a+b;
    }

    public int add(int a, int b){
        return a+b;
    }

    public double add(double a,double b){
        return a+b;
    }
    public int add(char a,char b){
        return a+b;
    }
}

```

八、可变参数

当一个方法的参数类型确定，但个数不确定时，可以把它声明为可变参数。

```
//求任两个以上整数的和
public int sum(int a,int b,int... c){//c是一个数组
    int sum = a+b;
    for(int i=0;i<c.length;i++){
        sum+=c[i];
    }
    return sum;
}

//-----
sum(1,2);
sum(1,2,3);
sum(1,2,3,4);
```

注意:

- 一个方法最多只能有一个可变参数，并且必须在最后一个参数位置
- 调用方法时，可变参数可以传入任意个此类型的数据，在方法内会被封装成一个数组。

九、递归

1. 概念：方法自身调用自身的情况

2. 分类：

- 直接递归：方法自己直接调用自己
- 间接递归：方法自己间接调用自己

3. 递归的注意事项：

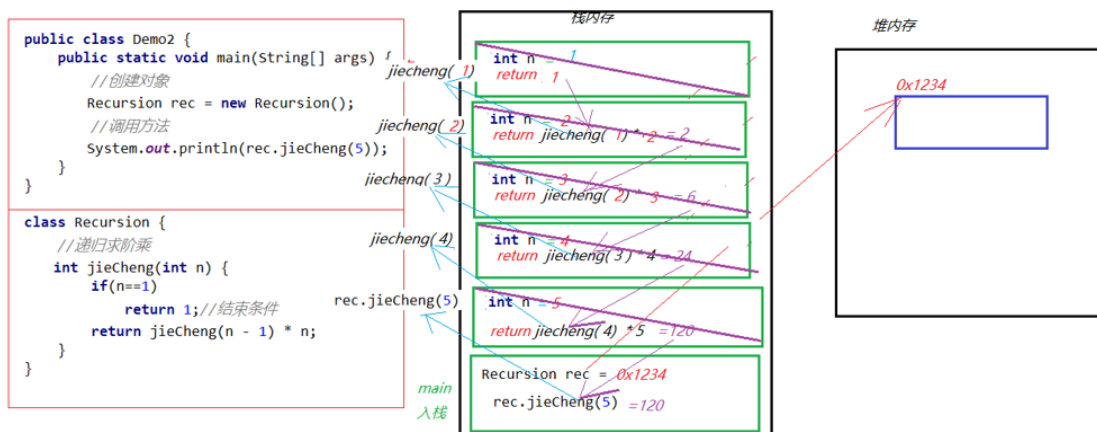
- 递归必须有出口（结束条件），否则就是无穷递归，类似死循环，会导致栈内存溢出
- 递归即使有出口，也不应该递归的次数太多，否则还是可能会导致内存溢出

4. 示例

求n的阶乘

```
//f(n)=f(n-1)*n , f(1)=1
public int jiecheng(int n){
    if(n == 1)
        return 1;
    return jiecheng(n-1)*n;
}
```

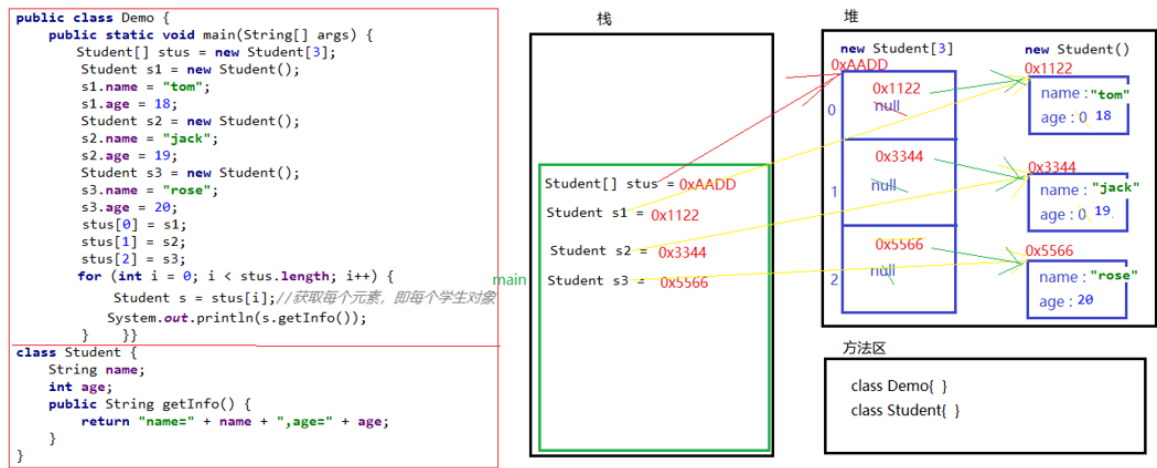
5. 递归内存分析



十、对象数组

对象数组：数组元素为引用数据类型的数组。

```
//对象数组
Student[] stus = new Student[3];
//给数组元素赋值
stus[0] = new Student();
stus[1] = new Student();
stus[2] = new Student();
```



第六章 面向对象基础（中）

一、封装

- 1. 概念理解：把事物内部的细节隐藏起来，不让外界轻易访问，而是提供对外的安全的访问途径。
- 2. 权限修饰符：
权限修饰符共有4种，分别为public，protected、缺省、private；权限修饰符可以使得数据在一定范围内可见或者隐藏。

修饰符	本类	本包	其他包子类	任意位置
private	√	×	×	×
缺省	√	√	×	×
protected	√	√	√	×
public	√	√	√	√

- 权限修饰符可以修饰：
- 外部类：public和缺省
 - 成员变量、成员方法、构造器、成员内部类：public,protected,缺省,private
3. 类的封装
- 私有化成员变量，提供公共的标准的get、set方法。

```
public class Student{
    //属性私有化
```



```

private String name;
private int age;

//公共的get、set方法
public void setName(String name){
    this.name = name;//this代表当前对象，这里主要是为了区分成员变量与局部变量
}

public String getName(){
    return name;
}

public void setAge(int age){
    this.age = age;
}

public int getAge(){
    return age;
}
}

```

二、继承

1. 概念理解：类似生活中的继承，Java类可以有父子关系，子类可以继承父类的成员变量和方法。描述的是一种is-a的关系
2. 继承的好处：
 - 提高代码的复用性：父类中声明的成员变量和方法，子类无需重复声明
 - 提高程序的扩展性：子类可以继承父类的属性和功能，并且可以扩展新的功能。
 - 是多态的前提

弊端：增强了类与类之间的耦合度。不要为了继承而继承。

3. 继承的格式：

定义子类时，通过关键字extends来继承父类

【修饰符】 `class 子类 extends 父类`{子类的成员}

```

//父类-超类-基类
public class Animal{
    public String name;
    public void eat(){
        System.out.println("吃饭...")
    }
}

//子类-派生类
public class Cat extends Animal{

}

```

4. 继承的特点：

- 子类会继承父类的所有成员变量和方法，但是无法直接访问私有的成员。
- Java类只支持单继承。一个类只能有一个父类。
- Java类支持多层继承，一个类可以继承另一个类的同时，还可以有自己的子类.A->B->C

- 一个类可以有多个子类。

三、方法重写

1. 概念理解：继承关系中，子类继承了父类的实例方法，但是不满足需求，而重写一个相同的方法，一般方法体不同。
2. 示例

```
public class Animal{
    public void eat(){
        System.out.println("吃饭...")
    }
}

public class Cat extends Animal{
    //方法重写
    public void eat(){
        System.out.println("猫吃鱼...");
    }
}
```

3. 方法重写的具体要求：

- 子类与父类中的声明的方法名和参数列表必须完全一致。
- 子类方法的权限修饰符不能比父类的小
- 子类方法的返回值类型不能比父类的大
- 子类方法声明的异常不能比父类的大，父类没有声明异常，子类也不能有。（异常时讲）

注意：父类私有的方法不能被重写。static和final修饰的方法也不能重写。

四、多态

1. 概念：事物在不同的条件下呈现不同的特征状态，Java中的多态有类似概念
2. Java多态的引用形式：

父类类型 变量 = 子类对象

此处的父类类型指的是继承关系中的父类或接口与实现类关系中的接口

3. 多态的表现（理解）

编译看左边（父类类型），运行看右边（子类类型）

```
Animal a = new Cat();
a.eat(); //是否能编译通过，要看左边父类中是否有此方法。但是运行时确是执行右边子类中重写的方法
a = new Dog();
a.eat();
```

4. 多态的好处

运行时，看“子类”，如果子类重写了方法，一定是执行子类重写的方法；变量引用的子类对象不同，执行的方法就不同，实现动态绑定。代码编写更灵活、功能更强大，可维护性和扩展性更好了。

- 提高程序的扩展性（参考引入案例分析）
- 降低类与类之间的耦合度（参考引入案例分析）

5. 多态的应用

- 用在成员变量属性，用在方法的参数上
声明父类类型的成员变量，赋值子类对象
声明父类类型的形参，实际传入的参数为子类对象

```
public class Person{  
    private Pet pet;//父类类型的属性  
  
    //领养宠物  
    public void adopt(Pet pet){//形参为父类类型  
        this.pet = pet;  
    }  
}
```

- 用在数组
声明父类类型的数组，元素为子类对象

```
Pet[] pets = new Pet[2];//父类Pet类型的数组  
pets[0] = new Cat();//元素为子类对象  
pets[1] = new Dog();
```

- 用在方法的返回值类型
声明方法的返回值类型为父类类型，实际返回值为子类对象

```
public class PetShop{  
    //返回值为父类Pet类型  
    public Pet sale(String type){  
        switch(type){  
            case "cat":  
                return new Cat();  
            case "dog":  
                return new Dog();  
            default:  
                return null;  
        }  
    }  
}
```

6. 向上转型与向下转型

- 向上转型：子类类型转换为父类类型，自动转换

```
Animal a = new Cat();//向上转型  
Dog dog = new Dog();  
Animal a2 = dog;//向上转型
```

- 向下转型：父类类型转换为子类类型，需要强制转换

```
Animal a = new Cat();
//a.catchMouse();//无法直接调用子类特有的方法
Cat c = (Cat)a;//强制向下转型
c.catchMouse();//可以调用
```

7. 关键字instanceof

强制类型转换有风险，可能会发生类型转换异常问题。为了避免此问题发生，可以使用关键字 instanceof 来判断某个对象运行时类型是否属于某种类型，再进行强制转换。

```
Animal a = new Cat();
//Dog d = (Dog)a;//会发生ClassCastException问题。
if(a instanceof Cat){
    Cat c = (Cat)a;
}else if(a instanceof Dog){
    Dog d = (Dog)a;
}
```

8. 虚方法（理解）

虚方法指的是可能被重写的方法（实例方法）。编译期不能取得到底执行哪个方法，只有在运行时才能确定。

多态引用形式下，虚方法的执行过程？

1. 静态分派：编译期在父类中找到并确定最匹配的方法（父类中多个重载方法都匹配）
2. 动态绑定：运行时，执行之前找到的最匹配方法的重写的方法

成员变量和非虚方法不具有多态性。

五、构造器

1. 构造器的作用：在创建对象时为实例变量初始化赋值

2. 格式：

【权限修饰符】 构造器名(【参数列表】){构造体--为实例变量赋值语句}

```
public class Student{
    private String name;
    private int age;
    //空构造器：每个类中默认自带空参构造器。当显示给出一个构造器时，默认空参构造器不再存在
    public Student(){
    }
    //有参构造器
    public Student(String name){
        this.name = name;
    }
}
```

3. 格式说明：类似普通方法

- 权限修饰符可以是任意四种的一种
- 构造器名称必须与当前类名相同
- 构造器没有返回值类型，void也没有
- 构造体中代码通常是实例变量赋值

4. this调用构造器

在一个类内部，可以存着多个构造器重载，构造器之间可以通过 `this(【参数】)` 进行相互调用。

`this(【参数】)` 这句代码必须在构造器内的第一行

```
public class Student{
    private String name;
    private int age;
    //空构造器
    public Student(){
    }
    //有参构造器
    public Student(String name){
        this();//必须在第一行
        this.name = name;
    }

    public Student(String name,int age){
        this(name);//调用本类的一个参数为String的构造器，必须在第一行
        this.age = age;
    }
}
```

5. super调用父类构造器

在继承关系中，子类构造器内可以使用 `super(【参数】)` 来调用父类的构造器。

- 子类中每个构造器内都默认隐藏`super()`,表示调用父类空参构造器
- 当子类构造器中显示使用`super([参数])`时或`this([参数])`，默认隐藏的`super()`不存在了
- `super([参数])` 这句代码必须在构造器内第一行，所以不能跟`this([参数])`共存

通过子类任意构造器创建对象时，必须要直接或间接通过调用`super([参数])`来先执行父类构造器。

六、非静态代码块（了解）

1. 作用：为实例变量初始化值
2. 格式示例：

```
class Student{
    String name;
    //非静态代码块-构造代码块-实例化代码块
    {
        this.name="tom";
    }
}
```

3. 执行特点：

每次创建对象时都会执行，并且先于构造器执行。

七、实例初始化过程（了解）

1. 实例初始化过程的目的主要是为实例变量初始化值--为实例变量赋予有效值。
2. 实例变量初始化值的方式：
 1. 直接显示赋值语句
 2. 非静态代码块-构造代码块

3. 构造器

当然，如果没有通过以上方式为实例变量初始化值，那么实例变量也有默认初始值。

```
public class Student{
    private String name;//有默认初始值
    private int age = 18;//直接显示赋值语句
    private char gender;//有默认初始值
    //非静态代码块
    {
        this.name = "tom";
    }

    //构造器
    public Student(){}
    //有参构造器
    public Student(char gender){
        this.gender = gender;
    }
}
```

3. 实例初始化方法

Java类被编译后，以上实例变量初始化方式都会被整合进一个或多个实例初始化方法{}中（与构造器对应）。这个方法包含4部分内容：

1. super(【参数】)：用于调用父类的构造器
2. 直接显示赋值语句
3. 非静态代码块中的语句
4. 构造器中其他语句

特别注意这4部分代码的执行顺序：1一定最先执行，2、3根据书写先后顺序执行，最后执行构造器中其余代码。

4. 实例初始化执行特点：

- 创建对象时，才会执行
- 每new一个对象，都会完成该对象的实例初始化
- 调用哪个构造器，就是执行它对应的实例初始化方法
- 子类super()还是super(实参列表)实例初始化方法中的super()或super(实参列表)不仅仅代表父类的构造器代码了，而是代表父类构造器对应的实例初始化方法。

八、部分关键字

1、this和super关键字

1. 使用位置和含义

1. this表示当前对象
 - 构造器中和构造代码块中：表示正创建对象
 - 实例方法中：表示调用此方法的对象
2. super表示调用父类的成员（不代表对象）
 - 构造器中、构造代码块中、实例方法中

2. 使用方式

1. this.实例变量和方法：调用本类的实例成员变量和方法
2. this([参数])：调用本类的其他构造器

3. super.实例变量和方法：调用父类的实例变量和方法
4. super([参数])：调用父类构造器
3. this和super还可以用于区分父子类中同名的成员变量（尽量避免）和方法

2、native关键字

用于修饰方法，表示本地方法，方法体不是java编写，而是其他代码编写。我们可以直接调用。

3、final关键字

表示最终的，不可修改的

1. 修饰的变量是常量
2. 修饰的类不能被继承
3. 修饰的方法不能被重写

九、Object类

Java类的根父类，所有的Java类都直接或间接的继承自Object类，包括数组都实现了其方法。

相关方法：

1. int hashCode();

返回对象的哈希码值，默认返回对象的内存地址转换而来的一个整数。

建议所有子类都重写此方法。使其返回值与对象的属性值有关。

2. Class getClass();

返回对象的运行时类型

```
Student s = new Student();  
String name = s.getClass().getName(); // 获取对象s的运行时类型名称
```

3. String toString()

把对象转换为一个字符串形式

默认返回值为：对象的运行时类型名称+'@'+对象的哈希码值的16进制形式

建议所有子类都重写此方法，使其返回值为对象的属性信息

打印输出语句打印一个对象时，默认会调用对象的toString方法。

```
Student s = new Student();  
System.out.println(s);  
System.out.println(s.toString()); // 等价于上句
```

4. boolean equals(Object obj);

用于比较两个对象是否“相等”。这两个对象分别是：此方法的调用者和传入的参数

默认比较的是两个对象的内存地址

建议所有子类都重写此方法，用于比较两个对象的内容是否相等

```

class Student{
    private int x;
    private int y;
    //自己重写的，不够严谨
    public boolean equals(Object obj){
        Student s = (Student)obj;
        if(this.x==s.x&&this.y==s.y){
            return true;
        }
        return false;
    }
}

```

5. void finalize()

当对象变为垃圾时，垃圾回收器会回收此对象，并提前调用对象的此方法。

十、标准JavaBean

JavaBean 是 Java 语言编写类的一种标准规范。标准的 JavaBean 一般需遵循以下规范：

- (1) 类必须是具体的和公共的，
- (2) 并且具有无参数的构造方法，
- (3) 成员变量私有化，并提供用来操作成员变量的 set 和 get 方法。

(4) 实现 `java.io.Serializable` 接口？ `toString` 方法

第七章 面向对象基础（下）

一、static 静态

1、静态变量

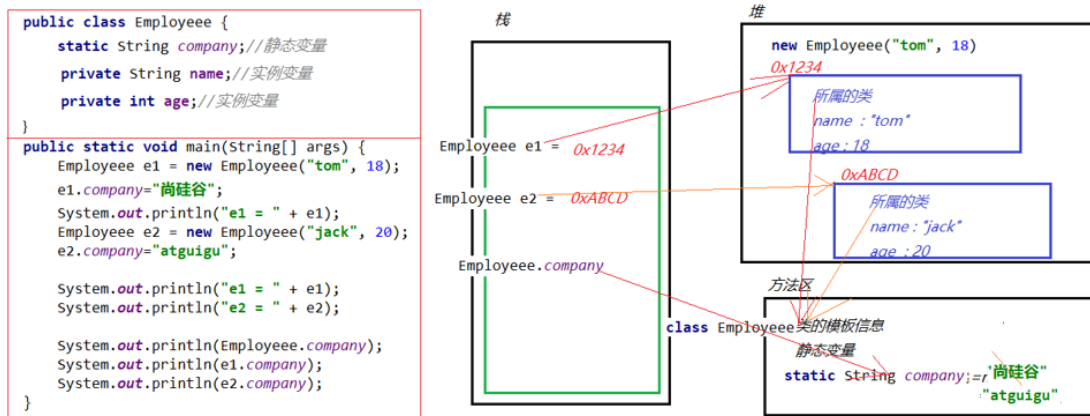
1. 静态变量：static 修饰的成员变量，也称为类变量

2. 静态变量特点：

所有对象共享的一份数据

在内存中的位置是方法区

3. 内存分析



4. 访问方式：

类名.静态变量 (推荐)

对象.静态变量 (不推荐)

2、静态方法

1. 静态方法：static修饰的方法，也称为类方法。工具类中的方法通常都是静态方法，方便调用。

2. 访问方式：

类名.静态方法 (推荐)

对象.静态方法 (不推荐)

3. 示例

```
public class MyTools{
    public static int sum(int a,int b){
        reutrn a+b;
    }
}

//-----调用-----
int sum = MyTools.sum(1,2);
```

4. 特点：

- 静态方法可以被子类继承，但不能被子类重写。
- 静态方法的调用都只看编译时类型。

3、静态代码块

1. 静态代码块：static修饰的代码块

2. 作用：通常用于为类变量初始化值

3. 格式示例

```
public class Student{
    private static String school;
    //静态代码块
    static{
        school = "尚硅谷";
    }
}
```

4. 执行特点（区别非静态代码块）：

静态代码块：在类初始化时执行，并且只执行一次。

非静态代码块：在实例初始化过程中执行，每次new对象都会执行。

类初始化先于实例初始化执行

