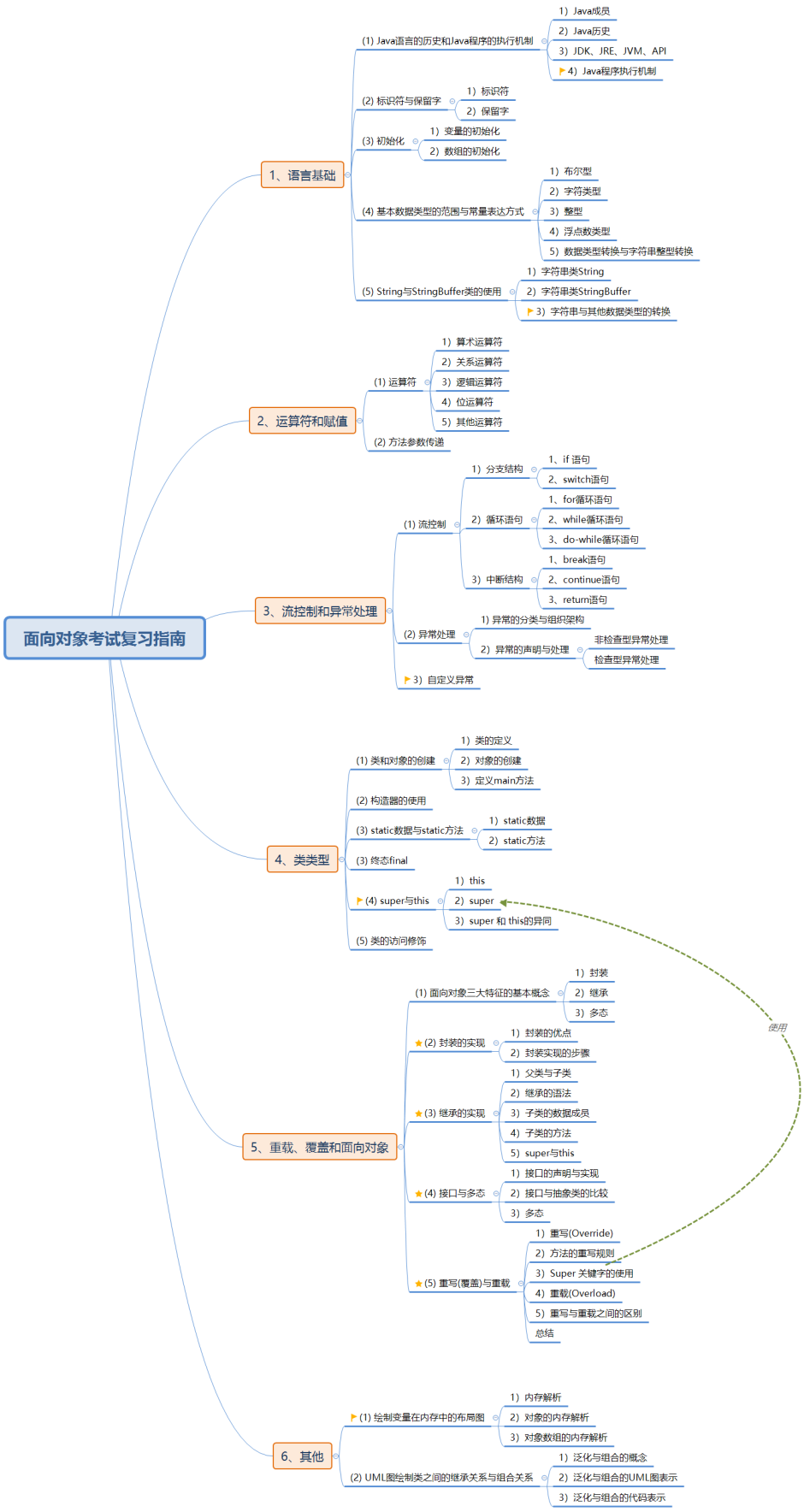
面向对象考试复习指南

- 1、语言基础
  - (1) Java语言的历史和Java程序的执行机制
    - 1）Java成员
    - 2）Java历史
    - 3）JDK、JRE、JVM、API
    - 4）Java程序执行机制
  - (2) 标识符与保留字
    - 1）标识符
    - 2）保留字
  - (3) 初始化
    - 1）变量的初始化
    - 2）数组的初始化
  - (4) 基本数据类型的范围与常量表达方式
    - 1）布尔型
    - 2）字符类型
    - 3）整型
    - 4）浮点数类型
    - 5）数据类型转换与字符串整型转换
  - (5) String与StringBuffer类的使用
    - 1）字符串类String
    - 2）字符串类StringBuffer
    - 3）字符串与其他数据类型的转换
- 2、运算符和赋值
  - (1) 运算符
    - 1）算术运算符
    - 2）关系运算符
    - 3）逻辑运算符
    - 4）位运算符
    - 5）其他运算符
  - (2) 方法参数传递
- 3、流控制和异常处理
  - (1) 流控制
    - 1）分支结构
      - 1、if 语句
      - 2、switch语句
    - 2）循环语句
      - 1、for循环语句
      - 2、while循环语句
      - 3、do-while循环语句
    - 3）中断结构
      - 1、break语句
      - 2、continue语句
      - 3、return语句
  - (2) 异常处理
    - 1) 异常的分类与组织架构
    - 2）异常的声明与处理
      - 非检查型异常处理
      - 检查型异常处理
    - 3）自定义异常
- 4、类类型
  - (1) 类和对象的创建
    - 1）类的定义
    - 2）对象的创建
    - 3）定义main方法

面向对象考试复习指南


面向对象考试复习指南
1、语言基础
(1) Java语言的历史和Java程序的执行机制
1）Java成员
2）Java历史
3）JDK、JRE、JVM、API
4）Java程序执行机制
(2) 标识符与保留字
1）标识符
2）保留字
(3) 初始化
1）变量的初始化
2）数组的初始化
(4) 基本数据类型的范围与常量表达方式
1）布尔型
2）字符类型
3）整型
4）浮点数类型
5）数据类型转换与字符串整型转换
(5) String与StringBuffer类的使用
1）字符串类String
2）字符串类StringBuffer
3）字符串与其他数据类型的转换
2、运算符和赋值
(1) 运算符
1）算术运算符
2）关系运算符
3）逻辑运算符
4）位运算符
5）其他运算符
(2) 方法参数传递
3、流控制和异常处理
(1) 流控制
1）分支结构
1、if 语句
2、switch语句
2）循环语句
1、for循环语句
2、while循环语句
3、do-while循环语句
3）中断结构
1、break语句
2、continue语句
3、return语句
(2) 异常处理
1）异常的分类与组织架构
2）异常的声明与处理
非检查型异常处理
检查型异常处理
3）自定义异常
4、类类型
(1) 类和对象的创建
1）类的定义
2）对象的创建
3）定义main方法
(2) 构造器的使用
(3) static数据与static方法
1）static数据
2）static方法
(3) 终态final
(4) super与this
1）this
2）super
3）super 和 this的异同
(5) 类的访问修饰
5、重载、覆盖和面向对象
(1) 面向对象三大特征的基本概念
1）封装
2）继承
3）多态
(2) 封装的实现
1）封装的优点
2）封装实现的步骤
(3) 继承的实现
1）父类与子类
2）继承的语法
3）子类的数据成员
4）子类的方法
5）super与this
(4) 接口与多态
1）接口的声明与实现
2）接口与抽象类的比较
3）多态
(5) 重写(覆盖)与重载
1）重写(Override)
2）方法的重写规则
3）Super 关键字的使用
4）重载(Overload)
5）重写与重载之间的区别
总结
使用
6、其他
(1) 绘制变量在内存中的布局图
1）内存解析
2）对象的内存解析
3）对象数组的内存解析
(2) UML图绘制类之间的继承关系与组合关系
1）泛化与组合的概念
2）泛化与组合的UML图表示
3）泛化与组合的代码表示

1、语言基础

(1) Java语言的历史和Java程序的执行机制

1）Java成员

Java SE： Java最通用的版本，用于Pc的标准平台，适合初学使用

Java EE：工业中应用最广泛

Java ME：应用于嵌入式平台开发，现在很少用

2）Java历史

1996年1月：JDK 1.0第一次发布

如今（2021/9/20）已经更新到了16

3）JDK、JRE、JVM、API

JDK（Java Development Kit）：Java开发工具包

JRE(Java Runtime Enviroment)：是Java的运行环境

JVM(java virtual machine)：即Java虚拟机。引入Java语言虚拟机后，Java语言在不同平台上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言编译程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。

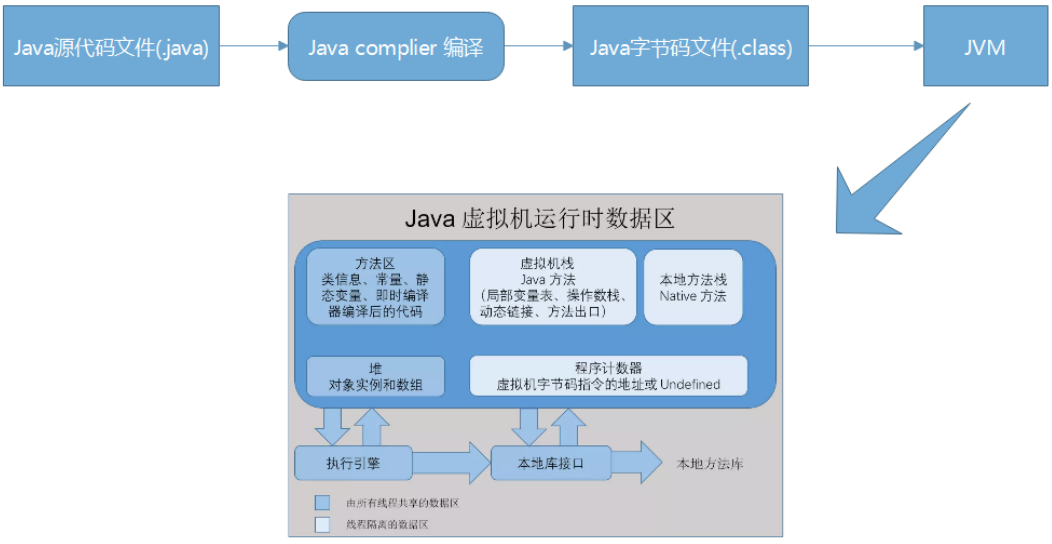
API：是一些预先定义的函数，目的是提供应用程序与开发人员基于某软件或硬件的以访问一组例程的能力，而又无需访问源码，或理解内部工作机制的细节。类似于Java的字典。

JDK是面向开发者的，JRE是面向使用JAVA程序的用户

4）Java程序执行机制

运行一个Java程序的步骤：
1、编辑源代码xxx.java
2、编译xxx.java文件生成字节码文件xxx.class
3、JVM中的类加载器加载字节码文件
4、JVM中的执行引擎找到入口方法main()，执行其中的方法

执行图解如下：



(2) 标识符与保留字

1）标识符

标识符的长度不限，但第一个字符必须是这些字符之一:大写字母(A-Z). 小写字母(a-z)、下划线、$符号，标识符的第二个字符及后继字符可以包括数字字符(0~ 9)。

2）保留字

`abstract`	`continue`	`for`	`new`	`switch`
assert	default	goto	package	synchronized
boolean	`do`	if	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

(3) 初始化

1）变量的初始化

1. 变量定义包括变量名、变量类型和变量值几个部分，定义变量的基本格式为:

```
数据类型 变量名 = 值
    int n = 5;
```

Java中变量的默认初始值都是确定的

```
布尔类型：false        整数变量：0        浮点数：0.0        引用（类）变量：null
```

2）数组的初始化

- Java数组在创建时会为每个元素赋予默认初始值也可以不采用默认值方式，而是对数组元素进行初始化，例如:

```
int a[]={22, 33, 44, 55};
int[] anArray = { 100,  200,  300,400,  500,  600,  700, 800};
String wkdays[] = {"mon" , "Tue", "Wed", "Thu", "Fri"};
```

- 或者等到数组创建完毕后，再分别为数组元素赋予适当的值之后再使用。例如，为int类型数组元素赋值方式如下:

```
int orange[ ] = new int[100]; // 创建100个int型元素,默认初始值为0
orange[7] = 32; // 为其中一个元素赋值32
```

- 基本数据类型数组的初始化比较简单，如果数组的元素是引用类型，这样的数组称为**“对象数组”**，对象数组的初始化相对复杂-一 些，不仅需要为数组分配空间，而且要**为数组中的每个元素分配空间**，使用前还需要给所有元素一一赋值(即new一个对象)。

(4) 基本数据类型的范围与常量表达方式

1）布尔型

- 布尔型只有true和false
- Java中的boolean和int完全不同，true不等于1，false不等于0
- 定义为：`boolean b = true`
- 可参与逻辑运算 `&& || == != !`

2）字符类型

- 表示单个字符，占有两个字节，注意加单引号' '
- 字符变量
  - Java中涉及char、byte、short的运算操作，都会先将这些值转换为int，然后再对int类型进行计算，所以int和char'的运算结果为int

```
char c ='1'
System.out.println(c+0) //结果为49
char d = '\u0031'
System.out.println(d) //结果为1
```

3）整型

- Java各整数类型有固定的表数范围和字段长度，不受具体OS的影响，以保证java程序的可移植性。
- Java的整型常量默认为 int 型，声明long型常量须后加'l'或'L'
- java程序中变量通常声明为int型，除非不足以表示较大的数，才使用long

类 型	占用存储空间	表数范围
byte	1字节=8bit位	-128 ~ 127
short	2字节	-2^15 ~2^15-1
int	4字节	-2^31 ~ 2^31-1 (约21亿)
long	8字节	-2^63 ~ 2^63-1

4）浮点数类型

- Java 浮点类型也有固定的表数范围和字段长度，不受具体操作
- 浮点型常量有两种表示形式：
  - 十进制数形式：如：5.12　512.0f　.512 (必须有小数点)
  - 科学计数法形式:如：5.12e2　512E2　100E-2

- float:单精度，尾数可以精确到7位有效数字。很多情况下，精度很难满足需求。double:双精度，精度是float的两倍。通常采用此类型。
- Java 的**浮点型常量默认为double型**，声明float型常量，须后加‘f’或‘F’。

类 型	占用存储空间	表数范围
单精度float	4字节	-3.403E38 ~ 3.403E38
双精度double	8字节	-1.798E308 ~ 1.798E308

5）数据类型转换与字符串整型转换

- 数据类型可以自动类型转换，int、long、float数据可以混合运算，混合运算是，转换由低级向高级

  byte、short、char —>int —> long —> float —> double

- boolean类型不能与其它数据类型运算

(5) String与StringBuffer类的使用

1）字符串类String

Java提供了两种处理字符串的类String和StringBuffer。Java将String类作为字符串的标准格式，Java 编译器把字符串转换成String对象。

1 字符串声明及初始化

Java中的字符串分为常量和变量两种，常量初始化可由直接给一个String对象赋值完成，字符串变量在使用前同样要声明和初始化，初始化过程一般有 下面几种形式。

1）直接用字符串常量来初始化字符串:

```
String s3 = "Hello! ";
```

2）由字符数组创建字符串:

```
char ch[ ] = {'s', 't, 'o, 'r, 'y'};
```

3）创建一个String类对象并赋值:

```
String s2 = new String ("Hel1o");
```

4）字符串数组形式:

```
String[] strArray;
strArray = new Str
sttarray,onew sString[8]
strArray[1]= "World";
```

2 字符串连接

String类的concat()方法可将两个字符串连接在一起

```
string1.concat (string2) ;
```

string1调用concat()将返回一个string1和string2连接后的新字符串。字符串连接通常还有另一种更为简洁的方式，通过运算符+连接字符串:“abc”+“def" =“abcdef" ;

“+”不仅可以连接多个字符串，而且可以连接字符串和其他的基本数据类型，只要+ 两端其中-个是字符串，另一个非字符串的数据也会被转换为字符串,然后进行字符串连接运算。

示例：测试String类的常用方法，实现字符串替换、单个字符检索、查找子串、比较、去空格等功能

```
public class Ex3_StringMethodTest {
    //测试String类的常用方法，实现字符串替换、单个字符检索、查找子串、比较、去空格等功能
    public static void main(String[] args) {
        String str = "Welcome to Java";
        System.out.println(str+"的字符长度为: "+str.length());
        System.out.println(str+"中第五个字符是："+str.charAt(5));
        System.out.println(str+"与hello world相同："+ str.equalsIgnoreCase("hello
world"));
        System.out.println(str+"用'L'代替'l'以后为："+str.replace("l","L"));
        System.out.println(str+"用'J'结尾："+str.endsWith("J"));
        System.out.println(str+"从第五个字符开始的字串为："+str.substring(5));
        System.out.println("  Thanks  "+"去掉开头和结尾的空格为："+"  Thanks
".trim());
    }
}
```

运行结果如下：

```
Welcome to Java的字符长度为: 15
Welcome to Java中第五个字符是：m
Welcome to Java与hello world相同：false
Welcome to Java用'L'代替'l'以后为：WeLcome to Java
Welcome to Java用'J'结尾：false
Welcome to Java从第五个字符开始的字串为：me to Java
  Thanks  去掉开头和结尾的空格为：Thanks
```

2）字符串类StringBuffer

StringBuffer类也是用来处理字符串的，它提供的功能很多与String类相同，但比String 更丰富些。两者的内部实现方式不同，String 类对象创建后再更改就产生新对象，而StringBuffer 类的对象在创建后，可以改动其中的字符，这是因为改变字符串值时，只是**在原有对象存储的内存地址上进一步操作，不生成新对象，内存使用上比String有优势，比较节省资源。**所以在实际开发中，如果经常更改字符串的内容，比如**执行插入、删除等操作，使用StringBuffer更合适些**，但StringBuffer不支持单个字符检索或子串检索。

示例：测试StringBuffer类，实现字符串的内容替换、反转等功能

```
public class Ex3_StringBufferTest {
    //测试StringBuffer类，实现字符串的内容替换、反转等功能
    public static void main(String[] args) {
        String str1 = "Welcome to Java";
        StringBuffer sf1 = new StringBuffer();
        sf1.append(str1);
```

```
        System.out.println("字符串sf1为："+sf1);
        System.out.println("字符串sf1的长度为："+sf1.length());
        System.out.println("字符串sf1的容量为："+sf1.capacity());
        sf1.setCharAt(2,'E');//更改字符串中的字母
        System.out.println("修改以后的字符串为："+sf1);

        sf1.reverse();
        System.out.println("倒转以后的字符串为："+sf1);

        sf1.replace(0,5,"hello");
        System.out.println("用hello替代以后的字符串为："+sf1);
    }
}
```

运行结果如下：

```
字符串sf1为：Welcome to Java
字符串sf1的长度为：15
字符串sf1的容量为：16
修改以后的字符串为：WeEcome to Java
倒转以后的字符串为：avaJ ot emocEeW
用hello替代以后的字符串为：helloot emocEeW
```

3）字符串与其他数据类型的转换

如何将字串 String 转换成整数 int?

有2个方法:

```
1). int i = Integer.parseInt([String]); 或

    i = Integer.parseInt([String],[int radix]);

2). int i = Integer.valueOf(my_str).intValue();

//注: 字串转成 Double, Float, Long 的方法大同小异.
```

如何将整数 int 转换成字串 String ?

有3种方法:

```
1.) String s = String.valueOf(i);

2.) String s = Integer.toString(i);

3.) String s = "" + i;

//注: Double, Float, Long 转成字串的方法大同小异.
```

字符串转化为字符数组

```
String str="123456";
char[] c = str.toCharArray() ;
```

2、运算符和赋值

(1) 运算符

1）算术运算符

运算符	运算	范例	结果
+	正号	+3	3
-	负号	b=4; -b	-4
+	加	5+5	10
-	减	6-4	2
*	乘	3*4	12
/	除	5/5	1
%	取模(取余)	7%5	2
++ ++	自增（前）：先运算后取值 自增（后）：先取值后运算	a=2;b=++a; a=2;b=a++;	a=3;b=3 a=3;b=2
- - - -	自减（前）：先运算后取值 自减（后）：先取值后运算	a=2;b=- -a a=2;b=a- -	a=1;b=1 a=1;b=2
+	字符串连接	“He”+”llo”	“Hello”

2）关系运算符

运算符	运算 范例 结果
==	相等于 4==3 false
!=	不等于 4!=3 true
<	小于 4<3 false
>	大于 4>3 true
<=	小于等于 4<=3 false
>=	大于等于 4>=3 true
instanceof	检查是否是类的对象 “Hello” instanceof String true

- 比较运算符的结果都是boolean型，也就是要么是true，要么是false。
- 比较运算符“**==**”不能误写成“**=**”

3）逻辑运算符

&—逻辑与	| —逻辑或	！—逻辑非
&& —短路与	|| —短路或	^ —逻辑异或

a	b	a&b	a&&b	a|b	a||b	!a	a^b
true	true	true	true	true	true	false	false
true	false	false	false	true	true	false	true
false	true	false	false	true	true	true	true
false	false	false	false	false	false	true	false

- 逻辑运算符用于连接布尔型表达式，在Java中不可以写成3<x<6，应该写成x>3 & x<6 。
- “&”和“&&”的区别：
  - 单&时，左边无论真假，右边都进行运算；
  - 双&时，如果左边为真，右边参与运算，如果左边为假，那么右边不参与运算。
- “|”和“||”的区别同理，||表示：当左边为真，右边不参与运算。
- **常用！、&&、||**

4）位运算符

	位运算符	
运算符	**运算**	**范例**
<<	左移	3 << 2 = 12 --> 322=12
>>	右移	3 >> 1 = 1 --> 3/2=1
>>>	无符号右移	3 >>> 1 = 1 --> 3/2=1
&	与运算	6 & 3 = 2
|	或运算	6 | 3 = 7
^	异或运算	6 ^ 3 = 5
~	取反运算	~6 = -7

位运算是直接对整数的二进制进行的运算

5）其他运算符

条件赋值：

```
op1?op2:op3
 (y<5)?y+6:y//当y<5时，计算结果为y+6；否则计算结果为y
```

(2) 方法参数传递

方法中不可避免使用到参数，接下来具体分析Java方法中的参数是如何传递的。Java参数可以是基本数据类型，也可以是引用(类)类型。在Java语言中，**基本数据类型作为参数时，均采用传值(passing-by-value)的方式完成**，对形参的任何改动都不会影响到实参。而**引用类型变量作为参数传递时，采用的是引用传递(passing-by-reference)的方式，在方法体中对形参的改动将会影响到实参。**

简单解释一下**实参和形参**的概念，实参是在调用时传递给方法的实际参数;形参是在定义方法名和方法体时使用的参数，目的是接收调用该方法时传人的参数。实参可以是常量.变量、表达式、方法等，无论实参采用何种类型，在调用方法时，它们都必领具有确定的
值，以便把这些值传送给形参，因此应预先用赋值、输人等办法使实参获得确定值。

当以基本数据类型为参数时，采用传值方式实现，形参仅是实参的一个拷贝， 它们的值相同，但是各自占有独立的内存地址空间，任何对形参的更改都不会影响到实参。而当以引用类型数据为参数时，采用引用传递方式实现，形参即为实参的别名，形参指向实参的内存地址空间，使用时便如同使用实参一样，任何对形参的更改都是对实参的更改。在实际开发中，常常会遇到需要改变对象数据的问题，这时应该考虑使用引用传递的方式来完成。

然而String类型虽然属于引用类型，但作为参数时采用的是**传值方式**来完成。String 类型对象一旦创建后就不可更改，重新赋予新值实际上是另外开辟内存地址进行存储，相当于创建了两个对象。所以方法中传递的参数类型为String时，形参和实参是两个对象，它们值相同，但各占一份独立的内存地址空间，对形参的任何更改都不会影响到实参，实际为传值效果，使用时需要注意。

3、流控制和异常处理

(1) 流控制

Java中的流程控制结构大体上分为三种：分支结构（if-then、if-then-else、switch)、循环语句（for、while、do-while)以及中断语句（break、continue、return)

1）分支结构

1、if 语句

语法如下：

```
if(op1)
    exec1;
else if(op2)
    exec2;
...
else execn;
```

2、switch语句

switch语句是一个多分支的选择语句，可以对多种情况进行判断并决定是否执行语句，其结构如下：

```
switch(表达式)
{
        case 值1：语句1；break;
        case 值2：语句2；break;
        case 值3：语句3；break;
        default: 语句4
}
```

switch语句使用时，首先判断表达式的值，如果表达式的值和某个case后面的值相同，则从该case之后开始执行，若满足值1，则执行语句1，满足值2，则执行语句2，直到break语句为止。default可有可无，若没有一个常量与表达式的值相同，则从default之后开始执行。

2）循环语句

1、for循环语句

for循环结构是Java三个循环语句中功能较强、使用较广泛的一个，结构上可以嵌套。for循环需要在执行循环体之前测试循环条件，其一般语法格式如下:

- ```
  for(初值表达式; boolean测试表达式 ; 改变量表达式){
      语句或者语句块
  }
  ```

2、while循环语句

执行while语句，当他的控制表达式为真时，while语句重复执行一个语句或者语句块，通用格式如下

```
while(条件表达式){
    语句或者语句块
}
```

举例：使用while循环接受并输出从键盘读入的字符，直到输入的字符为回车符

```
char ch = 'a';
while(ch != '\n'){
    System.out.println(ch);
    ch=(char)System.in.read();//接收键盘输入
}
```

3、do-while循环语句

do-whie的一般语法结构如下:

```
do{
    语句或语句块
}while(条件表达式)
```

do-while的使用与while语句很类似，不同的是**它首先无条件地执行一遍循环体**，再计算并判断循环条件，若结果为true,则重复执行循环体，反复执行这个过程，直到条件表达式的值为false为止;反之，若结果为false跳出循环，则执行后面的语句。

3）中断结构
```

1、break语句

break语句不单独使用，常运用于swtich、while、do-while 语句中，使程序从当前执行中跳出，不执行剩余部分，转移到其他部分。

break语句在for循环及while循环结构中，用于终止break语句所在的**最内层循环**。示例代码片段如下:

```
int i = 0;
while(i<10){
    i++;
    if(i==5){
        break;
    }
}
```

该循环在变量i的值等于5时，执行break语句，结束整个循环，接着执行循环后续的代码。

与C/C++不同，Java 提供了一种**带标签的break 语句**，用于跳出标号标识的循环。这种break语句多用于跳出多重嵌套的循环语句，适合需要中断外部的循环，即采用标签语句来标识循环的位置，然后跳出标签对应的循环。示例代码片段如下:

```
label1:
for(int i = 0; i < 10 ; i++){
    System.out.println(j);
    if(j==3){
        break label1;//中断外部循环
    }
}
```

这里的label1是标签的名称，可以用任意的标识符定义，放在对应的循环语句上面，以冒号结束。在该示例代码中，label1在外循环上面，将会中断外循环，实现时在需要中断循环的位置，采用break后面跟着标签。

2、continue语句

continue语句必须用于循环结构中，功能是跳过该次循环，继续执行下一次循环。在while和do-while语句中，continue 语句跳转到循环条件处开始继续执行，而在for语句中，continue 语句跳转到for语句处开始继续执行。和break语句类似，continue 语句也有两种使用格式:不带标签的continue语句和带标签的continue语句，不带标签的continue语句将终止当前这一轮的循环，不执行本轮循环的后一部分，直接进入当前循环的下一轮。下面以while 语句为例，说明不带标签的continue语句用法，代码片段如下:

```
int i = 0;
while(i < 4){
    i++;
    if(i == 2){
        continue;
    }
    System.out.println(i);
}
```

在i值等于2时，执行continue语句，则后面未执行完的循环体将被跳过，不会执行System.out.println(2); 而是直接回到while 处，进入下一次循环，所以打印结果没有2。

带标签的continue语句将使程序跳出多重嵌套的循环语句，直接跳转到标签标明的循环层次，标签必须放置在最外层的循环之前，紧跟一个冒号。 和break类似，这种语句用于跳过外部的循环，需要使用标签来标识对应的循环结构，代码片段如下:

```
label1:
for(int i = 0; i < 10 ; i++){
    for(int j = 0; j < 5; j++){
        System.out.println(j);
        if(j == 3){
            continue label1;
        }
    }
}//在执行continue语句时，直接跳转到i++语句，而不执行j++
```

3、return语句

return语句总用在方法之中，有如下两个作用

1 返回方法指定类型的值，格式为：`return 表达式;`

2 结束方法的执行并返回至调用这个方法的位置，格式为：`return;`

(2) 异常处理

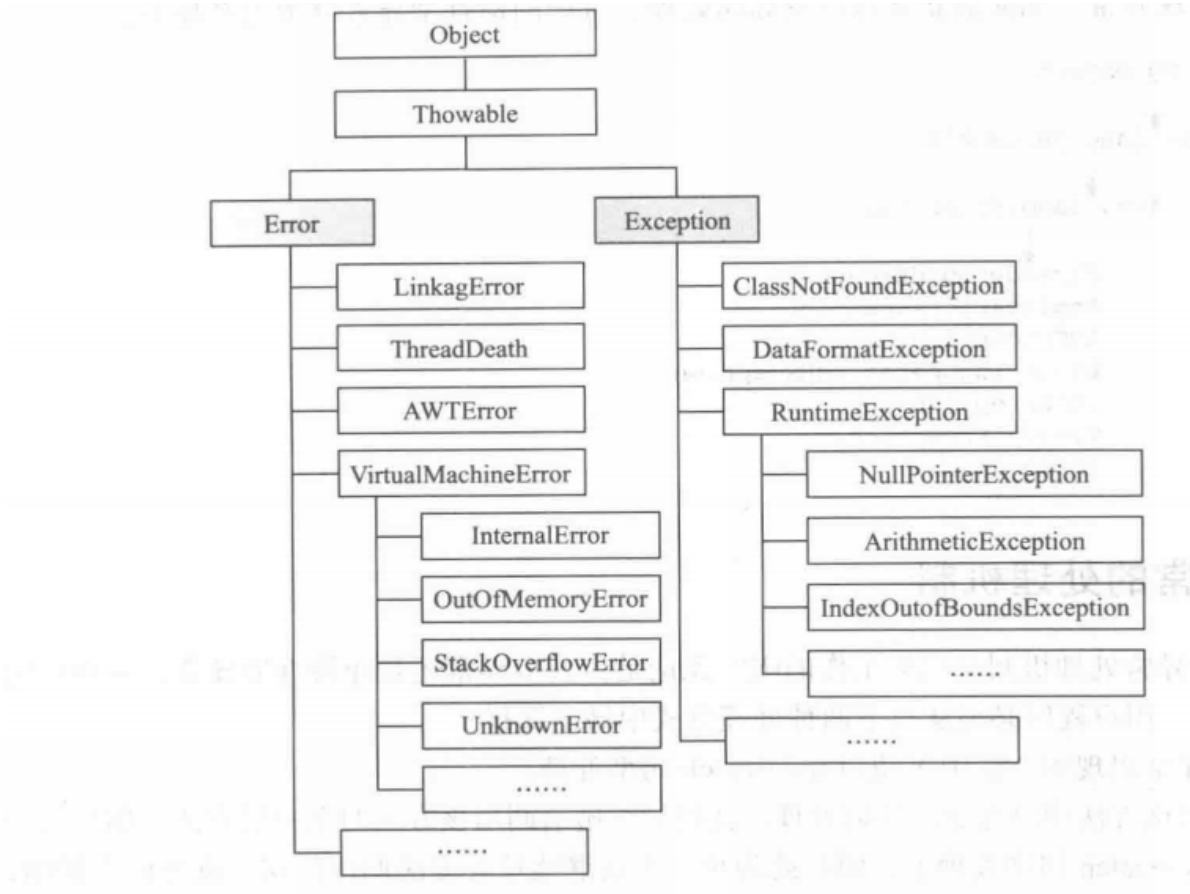
1) 异常的分类与组织架构

Java API的每个类库包中几乎都定义有异常类，如图形用户界面AWTException、输入/输出异常类IOException、数据库异常类sQLException.运行时异常类RuntimeException、算术运算异常类ArithmeticException等。

javalang 下的Throwable类是所有异常和错误的父类，它包含两个子类Error和Excption,分别表示错误和异常

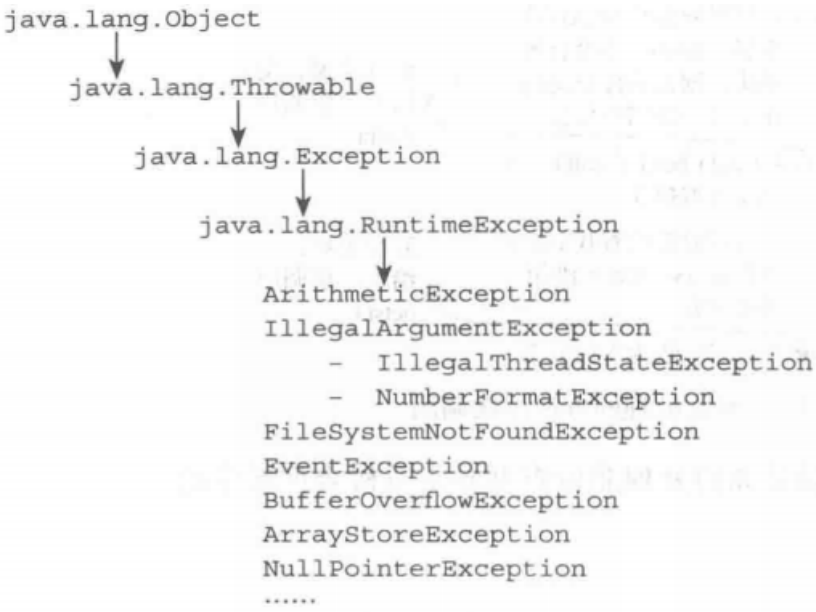
错误Error类用于处理致命性的、用户程序无法处理的系统程序错误，如硬件故障、虚拟机运行时错误(VritualMachineError)、线程死亡(ThreadDeath).动态链接失败(Linkagerror),这些错误用户程序是不需要关注的，一旦运行时出错，就由系统自动报错

Error类和Exception类的继承关系如下：

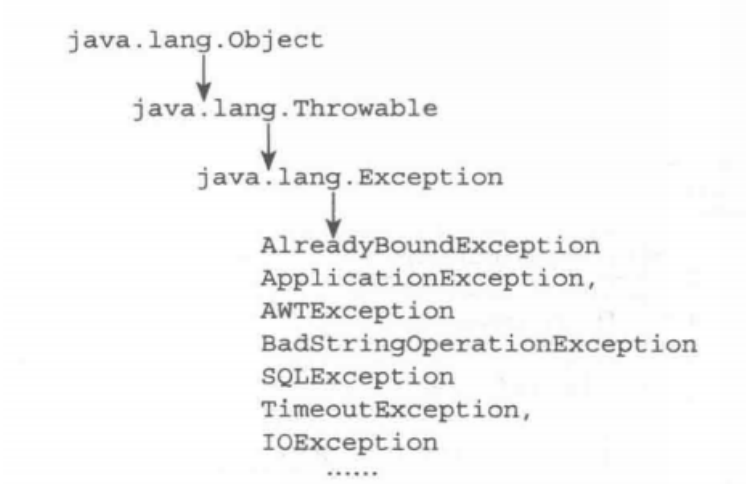


Java 的异常类总体上可以分为两类:非检查型异常(non-checked exception) 和检查型异常(checked exception)。

非检查型异常继承自RuntimeException运行时异常，这类异常是不需要检测的，由系统自动检测出异常，自动报错，提供默认的异常处理程序。程序中可以选择捕获处理，也可以不处理。这类异常一般是由程序逻辑错误引起的，应该从逻辑角度尽可能避免这类异常的
发生，常见的如数组越界、除零等。常用的非检查型异常继承关系如下:



检查型异常要求用户程序必须处理，属于**Exception类及其子类**，需要手动标示在什么位置可能出现异常、如何捕获异常以及如何处理。常用的检查型异常继承关系如下:



2）异常的声明与处理

Java的异常处理机制是如何工作的呢?无论是检查型异常还是非检查型异常，针对可能出现的异常，用户程序必须从以下两种处理方式中做出选择:
1) 在异常出现的方法中主动用try...catch句型处理。
2) 如果该方法中没有做出任何处理，就把异常抛给调用该方法的上一层方法，如果上一层方法由try-catch句型处理了，则到此为止，否则继续顺着方法调用的层次逐级向上抛出，沿着被调用的顺序往前寻找，直到找到符合该异常种类的处理程序，交给这部分程序处理，如
果抛到了程序调用顶层，main() 还没有被处理，则程序执行停止，main() 方法发出错误信息。

非检查型异常处理

非检查型异常属于系统定义的运行异常，由系统自动检测并做出默认处理，用户程序不必做任何事情

举例：处理数组越界出现的异常

```
public class TestException_1 {
    //处理数组越界出现的异常
    public static void main(String[] args) {
        int a[]=new int[5];
        a[6]=5;
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 6 out
of bounds for length 5
    at com.xjtu.chapter06.TestException_1.main(TestException_1.java:12)
```

检查型异常处理

对于检查型异常，Java要求用户程序必须进行处理，根据具体情况判断在代码段何处处理异常，处理方式包括两种：**捕获异常和声明抛出异常。**对于前者，使用try-catch语句，捕获到发生的异常，并进行相应的处理；对于后者，不在当前方法内处理异常，而是把异常抛出到调用方法中由上层方法处理

1 捕获异常

这种处理方式通常是在发生异常的方法内捕获该异常，并立即进行处理，语法格式如下:

```
try{
    statement()  //可能产生异常的代码块
}catch(exceptiontype objectname){
    statement()  //处理代码块
}
finally{
    statement()  //必须执行的代码
}
```

将可能产生异常的代码块放在try{中，每个try语句必须伴随一个或多个catch语句，用于捕获try 代码块产生的异常并做相应的处理。catch语句可以接受一个参数，即异常类型的对象，exceptiontype必须是一个从Throwable类派生出来的异常类的类型。

有时，需要执行finally语句,finally语句用于执行收尾工作，为异常处理提供一个统一的出口。无论程序是否抛出异常，也无论catch捕获到的异常是否正确，finally 指定的代码都要被执行，并且这个语句总是在方法返回前执行，目的是给程序一个补救的机会。**通常在finally语句中可以清除除了内存以外的其他资源，如关闭打开的文件、删除临时文件等。**

> 注意事项: try、catch、finally三个语句块必须组合起来使用，三者可以组成try...catch...finally、try ...catch和 try..finally三种结构，catch语句可以有一个或多个，**finally语句最多一个。**异常处理的查找遵循类型匹配原则，一个异常在匹配到其中一种异常类型后接着执行catch 块代码，一旦第一个匹配的异常处理被执行，则不会再执行后面的catch块。异常处理执行完毕，程序接着最后一个catch代码段后的语句继续执行。

2 声明抛出异常

在一个方法中生成了异常，但是该方法并不采用try-catch语句处理产生的异常，而是沿着调用层次向上传递，交给调用它的上一层方法来处理，这就是**声明抛出异常。**

声明抛出异常的方法是在产生异常的方法名后面加上关键字throws，后面接上所有可能产生异常的异常类型，语法格式如下:

```
void func() throws ExceptionA,ExceptionB,ExceptionC{
......
}
```

3）自定义异常

原则上，异常处理的过程应该分为三步:首先，将产生异常的代码段放在try{}里，然后抛出(throw)异常，最后捕获(catch) 异常。前面提到的try-catch方式，实际上省略了其中的抛出步骤，try-catch 方式处理的异常通常由Java JVM产生，或者由Java类库中的某些异常类产生，然后隐式地在程序里被抛出，JVM已经替程序完成了抛出异常的操作，而程序中只需执行try和catch两步即可。

然而，有些情形下三个步骤是缺一不可的，例如程序中仅仅使用Java类库提供的异常类不能够满足要求时，**需要自己定义异常类**，当然这些异常JVM是不可能识别的，只能由用户程序手动引发，通过new生成自定义的异常对象，然后将它抛出来(注意:这里是throw而不是throws)。throw 语法格式如下:

```
throw new ThrowableObject();
```

或者先自定义一个异常类，然后抛出其对象:

```
myException e = new myException();
throw e;
```

抛出的异常必须是Throwable或其子类的对象，throw语句常用于异常产生语句块中，与try-catch语句配合使用。

throws与throw仅一个字母的差别，却是两种完全不同的概念。throws写在方法的后面,抛出异常交给上级方法或类，即抛给调用它的方法进一步处理; 而throw多用来抛出自定义的异常类对象，这类异常必须是Throwable类的子类，需要用户自己手工进行捕获。

首先看看程序中如果显式地将Java类库提供的异常类对象通过throw抛出，这有助于理解异常处理的三个步骤。

举例：使用try，throw，catch处理三种情形：无异常、除数为零、数组越界可能产生的异常

```
public class Process {
    void Proc(int sel){
        System.out.println("******in case "+sel+" ******");
        if(sel==0){
            //没有异常
            System.out.println("no exception caught");
            return;
        }else if(sel==1){
            try{
                int i=0;
                int j=5/i;//除数为零
                throw new ArithmeticException();//显式地抛出ArithmeticException异常
对象
            }catch (ArithmeticException e){
                System.out.println(e.toString());
            }
        }else if(sel==2){
            try{
                int array[] = new int[4];
                array[5]=5;
                throw new ArrayIndexOutOfBoundsException();//显式地抛出
ArrayIndexOutOfBoundsExceptionn异常对象
            }catch (ArrayIndexOutOfBoundsException e){
                System.out.println(e.toString());
            }
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Process process = new Process();
        try{
            process.Proc(0);//调用Proc
            process.Proc(1);
            process.Proc(2);
        }catch (ArithmeticException e){
            System.out.println("catch: "+ e+";Reason: "+e.getMessage());
        }catch (ArrayIndexOutOfBoundsException e){
            System.out.println("catch: "+ e+";Reason: "+e.getMessage());
        }catch (Exception e){
            System.out.println("Will not be executed");
        }finally {
            System.out.println("must go inside finally");
        }
    }
```

```
}
```

运行结果：

```
******in case 0 ******
no exception caught
******in case 1 ******
java.lang.ArithmeticException: / by zero
******in case 2 ******
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 4
must go inside finally
```

> 此例中一旦有异常产生，就创建异常类ArithmeticException或者ArrayIndexOutOfBoundsException对象，并执行throw new ArithmeticException()语句抛出一个ArithmeticException类异常对象，或者执行throw new ArrayIndexOutOfBoundsException(语句抛出ArrayIndexOutOfBoundsException类异常对象。一 般而言，当抛出Java类库定义的异常时,JVM会自动识别，程序里往往可以省略throw步骤。

如何抛出一个自定义的异常类对象？Java允许用户自行设计异常类，以便处理运行中可能出现的逻辑错误。比如学生年龄为20是-一个整数，如果不小心给学生年龄赋值为200，编译时不会有语法错误，但不符合常识，这时，可以自定义一个AgeException异常类来表示这种异常。在软件开发中，需要自定义异常类的情形有很多,尤其是大型项目开发，开发人员通常预定义好一些异常类，这些异常类可以如同Java类库的异常类一样使用,方便项目组同事共享，在有异常隐患的程序中调用，以提高项目的整体稳定性及协调性。声明自己的异常时，所有异常类都必须是Exception的子类，声明格式如下:

```
class MyException extends SuperclassOfException{
    ...
}
```

其中SuperclassOfException可以为Exception、ArithmeticException、 IOException……

举例：以电子产品商店为例，假设产品价格少于100元则不合理，自定义一个异常类如下：

```
public class PriceException extends Exception {
    public PriceException(){
        System.out.println("the price is less than 100 yuan ,too low!");
    }
}
```

测试类如下：

```
public class TestPriceException {
    public static void main(String[] args) {
        Product product = new Product();
        product.productPrice = 20;
        try {
            if(product.productPrice<100) throw new PriceException();
        } catch (PriceException e) {
            e.getMessage();
        }

    }
}
```

运行结果如下：

```
the price is less than 100 yuan ,too low!
```

在main()中创建了一个product对象，其价格为20，在try块中判断ProductPrice的价格，小于100，则抛出自定义的PriceException对象，紧跟着的catch语句捕获到对象e后，调用构造方法输出错误信息。

> 编程提示:程序里尽可能多地使用异常处理机制，在后面的学习中，每一项新内容都应学会使用类库提供的异常处理方法，比如学习JDBC则应学会使用SQLException。

4、类类型

(1) 类和对象的创建

1）类的定义

类（class）是Java提供的一种抽象数据类型，以Bird类为例如下：

```
class Bird{             //类名
    String breed;
    String color;       //数据成员
    int numChirps;

    void chrip(){       //成员方法
        numChrips++;
    }
}
```

在类中，属性称为类的数据成员，方法称为类的成员方法

2）对象的创建

定义好的类只是一个抽象的概念，通常需要创建实例，**类的实例也称为对象**

通过对象调用类的数据成员，能够展现类的特征；调用类的方法，使之具有一定的行为

调用示例如下：

```
Bird red = new bird();  //创建一个red对象
red.chrip();            //调用方法
```

凡是对象的创建都采用new关键字实现，这里的red就是Bird类的一个对象，创建对象格式如下

```
Classname objctname = new Classname();
```

- 此语句在内存中为对象分配了内存空间
- 对象调用方法的格式如下：`对象名.方法名`

3）定义main方法

一个完整的Java程序除了对象和类以外，还需要一个main()方法作为程序的入口

```
public static void main(String args[]){
    代码
}
```

(2) 构造器的使用

构造方法(constructor),专门用于对象的初始化，负责为每个属性指定初值。构造方法是一种以类名来命名的特殊方法，没有返回值，可以带参数或不带参数，一个类可以有多个构造方法，构造方法在类中的定义格式如下:

```
public class Classname{
    Classname( ) {}              //无参构造方法
    Classname( larguments] ) {} //有参构造方法
}
```

构造方法的调用方式与普通成员方法的调用方式不同，不必通过对象来调用，而是在创建对象时自动调用，格式如下:

```
Classname obj1 =  new Classname();
Classname obj2 = new Classname [arguments];
```

最好加上默认构造方法

构造方法的特点总结如下:
1) 一种和类同名的特殊方法，一个类中可以有多个构造方法。
2) 用来完成对象的初始化工作。
3) 无返回类型、无修饰符void,通常被声明为**公有的(public)**。
4) 一个构造方法可以有任意多个参数。
5) 不能在程序中显式调用，在生成一个对象时，系统会自动调用该类的构造方法。
6) 如果没有写构造方法，系统会自动提供一个默认构造方法`Classname(){}`。
7) 一旦类中已有带参数的构造方法，系统则不会再提供默认构造方法。

(3) static数据与static方法

1) static数据

Java类中有一种特殊的数据成员，它不属于某个对象，不能通过某个对象来引用。在声明前加上关键字static，static数据也称为类数据，**属于类范围**。

static变量生命周期从创建开始到程序运行结束，可通过类名访问，格式为:

```
类名.staticVarible
```

2) static方法

类里同样可以定义一个static方法，称为类方法，可直接通过类名来访问。Java主程序的main()方法就是一个static方法

static方法不属于类的某个对象，所以它们只能引用static方法或其他的static方法，非静态的方法可以调用静态的方法，反之则不行

static方法的调用和static数据的调用类似，通过类名来调用，格式如下:

```
类名.staticMethod
```

(3) 终态final

final修饰符可以用在数据成员、方法、对象、类之前，这意味着是一种终结状态，即给定数值后就不能再做任何更改，例如:

```
final static int mynumber- 36; // 定义一个final变量
final Time today = new Time(12, 21,12); //定义一个final对象
final int dd = 42;//定义一个final变量
```

final修饰符放在类、方法、变量前表示的意义不同:

> final 在类之前:表示该类是终结类，不能再被继承。
>
> final在方法之前:表示该方法是终结方法，该方法不能被任何派生的子类覆盖。
>
> final在变量之前:表示变量的值在初始化之后就不能再改变，相当于定义了一个常量。
>
> 对于final,需要牢记以上放在不同实体前表示的意义，在程序设计时不至于造成不必要的麻烦

(4) super与this

1）this

this 是自身的一个对象，代表对象本身，可以理解为：**指向对象本身的一个指针。**

this 的用法在 Java 中大体可以分为3种：

1 普通的直接引用

这种就不用讲了，this 相当于是指向当前对象本身。

2 形参与成员名字重名，用 this 来区分：

示例：

```
class Person {
    private int age = 10;
    public Person(){
    System.out.println("初始化年龄："+age);
}
    public int GetAge(int age){
        this.age = age;
        return this.age;
    }
}

public class test1 {
    public static void main(String[] args) {
        Person Harry = new Person();
        System.out.println("Harry's age is "+Harry.GetAge(12));
    }
}
```

运行结果：

```
初始化年龄：10
Harry's age is 12
```

可以看到，这里 age 是 GetAge 成员方法的形参，this.age 是 Person 类的成员变量。

[3] 引用构造函数

这个和 super 放在一起讲，见下面。

2）super

super 可以理解为是指向自己超（父）类对象的一个指针，而这个超类指的是离自己最近的一个父类。

super 也有三种用法：

[1] 普通的直接引用

与 this 类似，super 相当于是指向当前对象的父类，这样就可以用 **super.xxx** 来引用父类的成员。

[2] 子类中的成员变量或方法与父类中的成员变量或方法同名

示例：

```
class Country {
    String name;
    void value() {
       name = "China";
    }
}

class City extends Country {
    String name;
    void value() {
    name = "Shanghai";
    super.value();      //调用父类的方法
    System.out.println(name);
    System.out.println(super.name);
    }

    public static void main(String[] args) {
       City c=new City();
       c.value();
       }
}
```

运行结果：

```
Shanghai
China
```

可以看到，这里既调用了父类的方法，也调用了父类的变量。若不调用父类方法 value()，只调用父类变量 name 的话，则父类 name值为默认值 null。

[3] 引用构造函数

- **super(参数)**：调用父类中的某一个构造函数（应该为构造函数中的**第一条语句**）。
- **this(参数)**：调用本类中另一种形式的构造函数（应该为构造函数中的**第一条语句**）。

示例：

```
class Person {
```

```
    public static void prt(String s) {
        System.out.println(s);
    }

    Person() {
        prt("父类·无参数构造方法:  "+"A Person.");
    }//构造方法(1)

    Person(String name) {
        prt("父类·含一个参数的构造方法:  "+"A person's name is " + name);
    }//构造方法(2)
}

public class Chinese extends Person {
    Chinese() {
        super(); // 调用父类构造方法（1）
        prt("子类·调用父类"无参数构造方法":  "+"A chinese coder.");
    }

    Chinese(String name) {
        super(name);// 调用父类具有相同形参的构造方法（2）
        prt("子类·调用父类"含一个参数的构造方法":  "+"his name is " + name);
    }

    Chinese(String name, int age) {
        this(name);// 调用具有相同形参的构造方法（3）
        prt("子类：调用子类具有相同形参的构造方法：his age is " + age);
    }

    public static void main(String[] args) {
        Chinese cn = new Chinese();
        cn = new Chinese("codersai");
        cn = new Chinese("codersai", 18);
    }
}
```

运行结果：

```
父类·无参数构造方法:  A Person.
子类·调用父类”无参数构造方法“:  A chinese coder.
父类·含一个参数的构造方法:  A person's name is codersai
子类·调用父类”含一个参数的构造方法“:  his name is codersai
父类·含一个参数的构造方法:  A person's name is codersai
子类·调用父类”含一个参数的构造方法“:  his name is codersai
子类：调用子类具有相同形参的构造方法：his age is 18
```

从本例可以看到，可以用 super 和 this 分别调用父类的构造方法和本类中其他形式的构造方法。

例子中 Chinese 类第三种构造方法调用的是本类中第二种构造方法，而第二种构造方法是调用父类的，因此也要先调用父类的构造方法，再调用本类中第二种，最后是重写第三种构造方法。

3）super 和 this的异同

[1] super(参数)：调用基类中的某一个构造函数（应该为构造函数中的第一条语句）

[2] this(参数)：调用本类中另一种形成的构造函数（应该为构造函数中的第一条语句）

[3] super： 它引用当前对象的直接父类中的成员（用来访问直接父类中被隐藏的父类中成员数据或函数，基类与派生类中有相同成员定义时如：super.变量名 super.成员函数据名（实参）

[4] this：它代表当前对象名（在程序中易产生二义性之处，应使用 this 来指明当前对象；如果函数的形参与类中的成员数据同名，这时需用 this 来指明成员变量名）

[5] **调用super()必须写在子类构造方法的第一行，否则编译不通过。每个子类构造方法的第一条语句，都是隐含地调用 super()，如果父类没有这种形式的构造函数，那么在编译的时候就会报错。**

[6] super() 和 this() 类似,区别是，super() 从子类中调用父类的构造方法，this() 在同一类内调用其它方法。

[7] super() 和 this() 均需放在构造方法内第一行。

[8] 尽管可以用this调用一个构造器，但却不能调用两个。

[9]**this 和 super 不能同时出现在一个构造函数里面**，因为this必然会调用其它的构造函数，其它的构造函数必然也会有 super 语句的存在，所以在同一个构造函数里面有相同的语句，就失去了语句的意义，编译器也不会通过。

[10] this() 和 super() 都指的是对象，所以，**均不可以在 static 环境中使用。**包括：static 变量,static 方法，static 语句块。

[11]从本质上讲，this 是一个指向本对象的指针, 然而 super 是一个 Java 关键字。

(5) 类的访问修饰

访问控制权限分为不同等级，从最大权限到最小权限依次为:
public→protected→包访问权限(没有关键字)→private

对于类的访问控制只提供了public(公共类)及包(默认类)两种权限，对于类成员的访问控制权限有以下几种:

[1] **公有(public):**可以被其他任何对象访问(前提是对类成员所在的类有可访向权限)。

[2] **保护(protected):**只可被**同包、同一类及其子类**的对象访问。

[3] **包访问权限:**不加任何修饰符，默认访问权限，仅允许同一个包内的成员访问。

[4] **私有(private):** 只能被这个类本身方法访问，在类外不可见。

对于同类、同包及其子类情形下，访问权限修饰符表示的封装程度如表所示

修饰符	同类	同包	子类	不同包之间的通用性
public	是	是	是	是
protected	是	是	是	否
default	是	是	否	否
private	是	否	否	否

5、重载、覆盖和面向对象

面向对象三大特征：封装、继承、多态

(1) 面向对象三大特征的基本概念

1）封装

封装就是将对象的数据和基于数据的操作封装成一个独立性很强的模块

将对象不需要对外提供的私有数据和私有操作隐藏起来

外界要访问封装的数据—>依靠对象将某些**操作公有化**，在类中定义为对外的公共接口

2）继承

继承是在当前类的基础上创建新类，并在其中添加新的属性和功能

继承是实现代码复用的主要方式

3）多态

多态即在一个继承关系的程序中允许同名的不同方法共存

多态是以继承为前提的

多态的引入使程序能够在继承的基础上进一步实现变异，增强其可扩展性

(2) 封装的实现

1）封装的优点

[1] 良好的封装能够减少耦合。

[2] 类内部的结构可以自由修改。

[3] 可以对成员变量进行更精确的控制。

[4] 隐藏信息，实现细节。

2）封装实现的步骤

[1] 修改属性的可见性来限制对属性的访问（一般限制为private），例如：

```
private  String u_id;
private  String u_pwd;
```

这段代码中，将 **u_id** 和 **u_pwd** 属性设置为私有的，只能本类才能访问，其他类都访问不了，如此就对信息进行了隐藏。

[2] 对每个值属性提供对外的公共方法访问，也就是创建一对赋取值方法，用于对私有属性的访问，例如：

```
    public String getU_id() {
         return u_id;
     }

     public void setU_id(String u_id) {
         this.u_id = u_id;
```

```
    }

    public String getU_pwd() {
        return u_pwd;
    }

    public void setU_pwd(String u_pwd) {
        this.u_pwd = u_pwd;
    }
```

采用 **this** 关键字是为了解决实例变量和局部变量之间发生的同名的冲突。

(3) 继承的实现

继承是在一个类(父类)的基础上扩展新的功能而实现的，父类定义了公共的属性和方法，而其子类自动拥有了父类的所有功能，在基础上，又可以增添自己特有的**新的属性和方法进行扩展。**

在Java创建一个新类时，总是在继承，除非指明继承于一个指定类，否则都是隐式地从Java的根类Object中派生出来的子类，即**Object类是所有类的"祖先"**，Java 中的类一律继承了Object类的方法，这是Java的一.大特色。

需要注意的是，Java **只支持类的单继承**，每个子类只能有一个直接父类，不允许有两个以上的父类，这样使Java的继承方式很直接，代码简洁，结构清晰。

1）父类与子类

父类（base class）:被直接或间接继承的类

子类（derived class）：子类将继承所有祖先的状态和行为，可以增加新的变量和方法，也可以覆盖（override）所继承的方法，赋予新的功能

在继承关系下，子类和父类之间是一种**is-a (或is kind of)**的关系，父类与子类之间必然是有共同点的，子类可看作是父类的一种特例，比如Computer类是Product 类的一种特例。

2）继承的语法

在类的定义中，通过关键字extends来表示子类对父类的继承，继承的语法格式为：

```
class childClass extends parentClass{}
```

子类是从父类中派生出来的，继承了父类的非私有数据成员和方法。

3）子类的数据成员

一个对象从其父类中继承数据成员和方法，但是不能直接访问从父类中继承的私有数据和方法，必须通过公有（或者保护）方法进行访问

如果子类中定义有和父类中相同的成员变量名，那么从父类中继承而来的同名变量将会**被隐藏**，例如以下代码有Parent类派生出Child类

```
class Parent{
    String name;
}
class Child{
    String name;
}
```

Child中的name将会覆盖从Parent继承而来的name

4）子类的方法

与数据成员的继承方式相同，子类也只继承父类中非private的成员方法，当子类中定义有和父类同名的成员方法时，从父类中继承而来的成员方法会被子类中的同名成员方法覆盖(Override)，方法覆盖即在子类中重新定义(重写)父类中同名的方法

5）super与this

(4) 接口与多态

接口的用途之一是实现多重继承，一个类通过接口使用其他多个类的资源。类也以接口方式将其他不相关的类整合到自己类中，并在此基础上进-一步扩展， 集合而成一个新的、可操作的新系统，实现了多重继承。

接口的用途之二在于它是一种规范，它规定了一组类在实现某些功能时必须拥有的统一规则，它屏蔽了相关功能的实施细节，以一种标准模式即接口方式提供给外部类或系统使用。实现了功能与系统的分离

1）接口的声明与实现

接口的声明格式如下：

```
[public] interface 接口名称{
    返回类型 方法名（参数列表）;
    ......
    类型 常量名=值
    ......
}
```

接口通过定义具体类实现，不能采用new运算符创建对象方式生成，而是在类里使用**implements**关键字实现接口。语法如下：

```
[public] class 类名称 implements 接口名1，接口名2{
    .....
    /*接口的方法体实现部分*/
    /*类本身的数据和方法*/
    ......
}
```

在具体类实现的接口方法必须显式地定义为public，因为接口方法声明默认为public abstarct的，否则无法编译通过，而且实现一个接口，必须实现接口中的所有抽象方法

接口如同类一样具有继承的功能，派生出子接口，与类不同的是，接口允许多重继承，语法如下：

```
interface 子接口名 extends 父接口名1,父接口名2,...{
    ......
}
```

当具体类实现一个子接口时，需要实现子接口连同其所有父接口中的所有抽象方法

2）接口与抽象类的比较

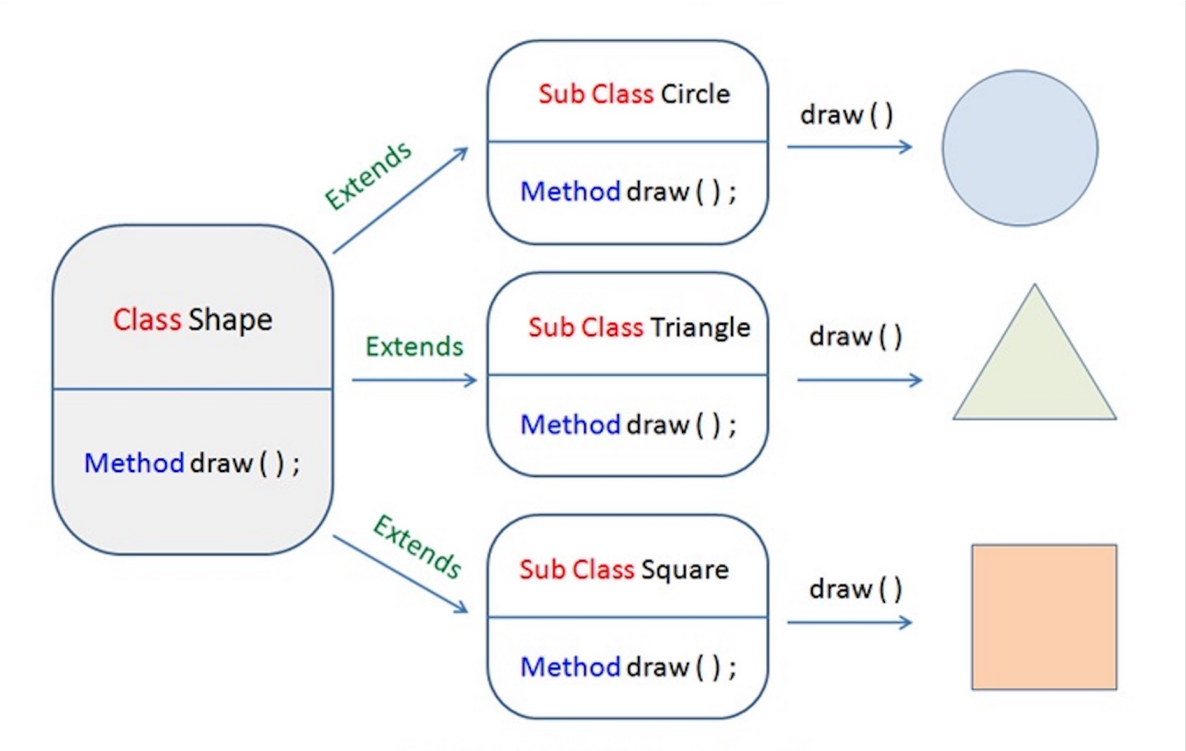
抽象类和接口在定义方面有一定的相似性，都具有抽象方法，抽象方法的实现都放在其他类中实现。二者的最大区别在于抽象类除了有抽象方法外还可以有一般方法的实现，而接口除了允许静态和默认方法外。只能是"纯抽象方法"

抽象类	接口
抽象类必须被继承，是一种继承关系	接口具有多重继承共享，一个类可以实现多个接口
抽象方法和成员变量可以是 public、protected 和默认 package 权限	接口方法和常量是 public 的

3）多态

多态存在的三个必要条件

- 继承
- 重写
- 父类引用指向子类对象：**Parent p = new Child();**



当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，再去调用子类的同名方法。

多态的好处：可以使程序有良好的扩展，并可以对所有类的对象进行通用处理。

示例代码如下：

```
public class Test {
    public static void main(String[] args) {
      show(new Cat());  // 以 Cat 对象调用 show 方法
      show(new Dog());  // 以 Dog 对象调用 show 方法

      Animal a = new Cat();  // 向上转型
      a.eat();               // 调用的是 Cat 的 eat
      Cat c = (Cat)a;        // 向下转型
      c.work();        // 调用的是 Cat 的 work
```

```
  }

    public static void show(Animal a)  {
      a.eat();
        // 类型判断
        if (a instanceof Cat)  {  // 猫做的事情
            Cat c = (Cat)a;
            c.work();
        } else if (a instanceof Dog) { // 狗做的事情
            Dog c = (Dog)a;
            c.work();
        }
    }
}

abstract class Animal {
    abstract void eat();
}

class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
    public void work() {
        System.out.println("抓老鼠");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
    public void work() {
        System.out.println("看家");
    }
}
```

执行以上程序，输出结果为：

```
吃鱼
抓老鼠
吃骨头
看家
吃鱼
抓老鼠
```

(5) 重写(覆盖)与重载

1）重写(Override)

重写是子类对父类的允许访问的方法的实现过程进行重新编写, 返回值和形参都不能改变。**即外壳不变，核心重写！**

重写的好处在于子类可以根据需要，定义特定于自己的行为。 也就是说子类能够根据需要实现父类的方法。

重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。例如： 父类的一个方法申明了一个检查异常 IOException，但是在重写这个方法的时候不能抛出 Exception 异常，因为 Exception 是 IOException 的父类，只能抛出 IOException 的子类异常。

在面向对象原则里，重写意味着可以重写任何现有方法。实例如下：

```
class Animal{
   public void move(){
      System.out.println("动物可以移动");
   }
}

class Dog extends Animal{
   public void move(){
      System.out.println("狗可以跑和走");
   }
}

public class TestDog{
   public static void main(String args[]){
      Animal a = new Animal(); // Animal 对象
      Animal b = new Dog(); // Dog 对象

      a.move();// 执行 Animal 类的方法

      b.move();//执行 Dog 类的方法
   }
}
```

以上实例编译运行结果如下：

```
动物可以移动
狗可以跑和走
```

> 在上面的例子中可以看到，尽管 b 属于 Animal 类型，但是它运行的是 Dog 类的 move方法。这是由于**在编译阶段，只是检查参数的引用类型**。然而在运行时，Java 虚拟机(JVM)指定对象的类型并且运行该对象的方法。因此在上面的例子中，之所以能编译成功，是因为 Animal 类中存在 move 方法，然而运行时，运行的是特定对象的方法。

思考以下例子：

```
class Animal{
   public void move(){
      System.out.println("动物可以移动");
   }
}

class Dog extends Animal{
   public void move(){
      System.out.println("狗可以跑和走");
   }
   public void bark(){
      System.out.println("狗可以吠叫");
   }
}
```

```
public class TestDog{
   public static void main(String args[]){
      Animal a = new Animal(); // Animal 对象
      Animal b = new Dog(); // Dog 对象

      a.move();// 执行 Animal 类的方法
      b.move();//执行 Dog 类的方法
      b.bark();
   }
}
```

以上实例编译运行结果如下：

```
TestDog.java:30: cannot find symbol
symbol  : method bark()
location: class Animal
                b.bark();
                 ^
```

该程序将抛出一个编译错误，因为b的引用类型Animal没有bark方法。

2）方法的重写规则

[1] 参数列表与被重写方法的参数列表必须完全相同。

[2] 返回类型与被重写方法的返回类型可以不相同，但是必须是**父类返回值的派生类**（java5 及更早版本返回类型要一样，java7 及更高版本可以不同）。

[3] 访问权限**不能比父类中被重写的方法的访问权限更低**。例如：如果父类的一个方法被声明为 public，那么在子类中重写该方法就不能声明为 protected。

[4] 父类的成员方法只能被它的子类重写。

[5] 声明为 final 的方法不能被重写。

[6] 声明为 static 的方法不能被重写，但是**能够被再次声**明。

[7] 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为 private 和 final 的方法。

[8] 子类和父类不在同一个包中，那么子类只能够重写父类的声明为 **public 和 protected 的非 final 方法。**

[9] 重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。

构造方法不能被重写。

如果不能继承一个类，则不能重写该类的方法。

3）Super 关键字的使用

当需要在子类中调用父类的被重写方法时，要使用 super 关键字。

```
class Animal{
   public void move(){
      System.out.println("动物可以移动");
```

```
    }
}

class Dog extends Animal{
    public void move(){
        super.move(); // 应用super类的方法
        System.out.println("狗可以跑和走");
    }
}

public class TestDog{
    public static void main(String args[]){

        Animal b = new Dog(); // Dog 对象
        b.move(); //执行 Dog类的方法

    }
}
```

以上实例编译运行结果如下：

```
动物可以移动
狗可以跑和走
```

4）重载(Overload)

重载(overloading) 是**在一个类里面**，方法名字相同，而参数不同。返回类型可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

最常用的地方就是构造器的重载。

重载规则:

> **被重载的方法必须改变参数列表(参数个数或类型不一样);**
>
> **被重载的方法可以改变返回类型;**
>
> **被重载的方法可以改变访问修饰符;**
>
> **被重载的方法可以声明新的或更广的检查异常;**
>
> **方法能够在同一个类中或者在一个子类中被重载。**
>
> **无法以返回值类型作为重载函数的区分标准。**

示例说明：

```
public class Overloading {
    public int test(){
        System.out.println("test1");
        return 1;
    }

    public void test(int a){
        System.out.println("test2");
    }
```

```
    //以下两个参数类型顺序不同
    public String test(int a,String s){
        System.out.println("test3");
        return "returntest3";
    }

    public String test(String s,int a){
        System.out.println("test4");
        return "returntest4";
    }

    public static void main(String[] args){
        Overloading o = new Overloading();
        System.out.println(o.test());
        o.test(1);
        System.out.println(o.test(1,"test3"));
        System.out.println(o.test("test4",1));
    }
}
```

运行结果：

```
test1
1
test2
test3
returntest3
test4
returntest4
```

5）重写与重载之间的区别

区别点	重载方法	重写方法
参数列表	必须修改	一定不能修改
返回类型	可以修改	一定不能修改
异常	可以修改	可以减少或删除，一定不能抛出新的或者更广的异常
访问	可以修改	一定不能做更严格的限制（可以降低限制）

总结

方法的重写(Overriding)和重载(Overloading)是java多态性的不同表现，**重写是父类与子类之间多态性的一种表现，重载可以理解成多态的具体表现形式。**

(1)方法重载是一个类中定义了多个方法名相同,而他们的参数的数量不同或数量相同而类型和次序不同,则称为方法的重载(Overloading)。

(2)方法重写是在子类存在方法与父类的方法的名字相同,而且参数的个数与类型一样,返回值也一样的方法,就称为重写(Overriding)。

(3)方法重载是一个类的多态性表现,而方法重写是子类与父类的一种多态性表现。

6、其他

(1) 绘制变量在内存中的布局图

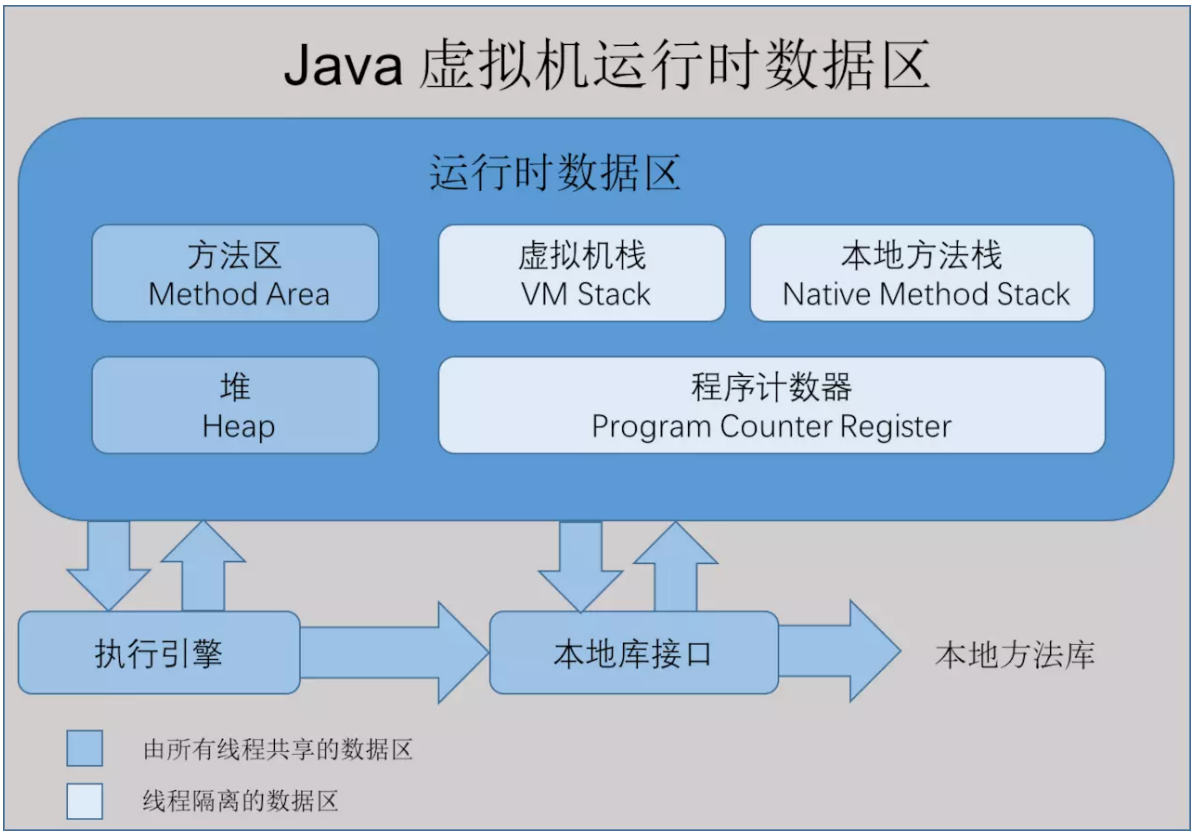
Java语言支持的变量类型有：

[1] 类变量：独立于方法之外的变量，用 static 修饰。

[2] 实例变量：独立于方法之外的变量，不过没有 static 修饰。

[3] 局部变量：类的方法中的变量

1）内存解析



[1] **堆（Heap）**，此内存区域的唯一目的就是存放对象实例，即new出来的结构，几乎所有的对象实例都在这里分配内存。这一点在Java虚拟机规范中的描述是：**所有的对象实例以及数组都要在堆上分配。**

[2] **通常所说的栈（Stack），是指虚拟机栈。**虚拟机栈用于存储局部变量等。 局部变量表存放了编译期可知长度的 各种基本数据类型（boolean、byte、char 、 short 、 int 、 float 、 long 、double）、对象引用（reference类型， 它不等同于对象本身，是**对象在堆内 存的首地址**）。 方法执行完，自动释放。

[3] 方法区（Method Area），用于存储已被虚拟机加载的类信息、常量、**静态变量**、即时编译器编译后的代码等数据。

2）对象的内存解析

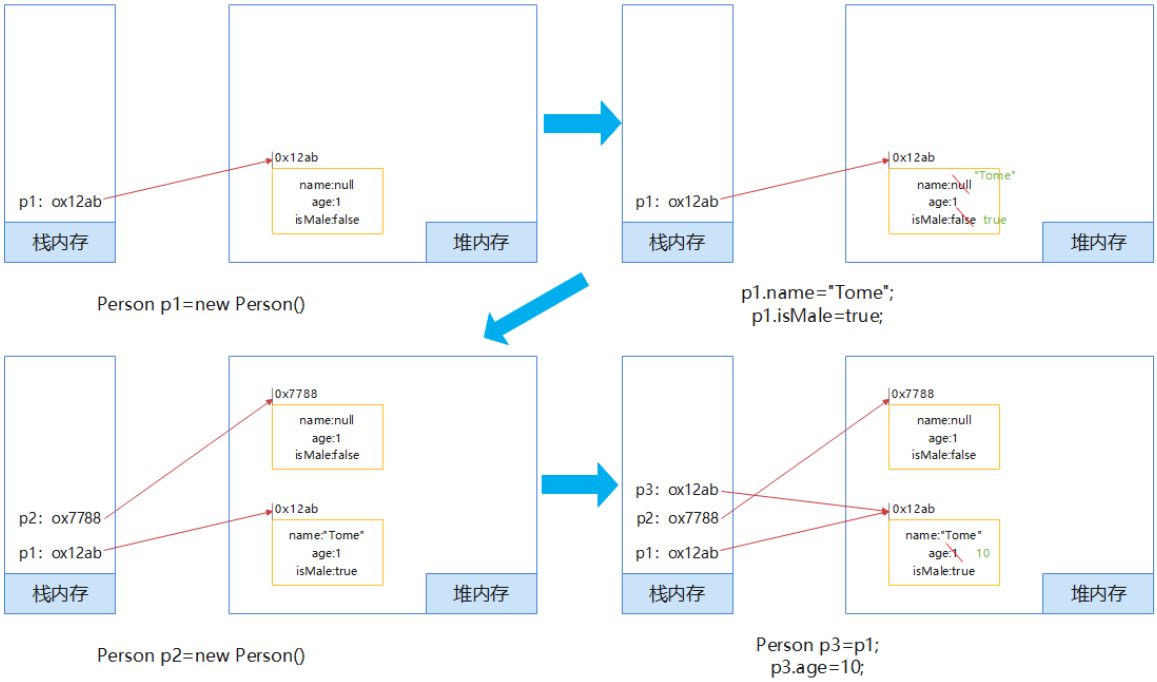
```
public class Person {
    String name;//姓名
    int age = 1;//年龄
    boolean isMale;//是否为男性

    public static void main(String[] args){
        Person p1=new Person();
        p1.name="Tome";
```

```
        p1.isMale=true;
        Person p2=new Person();
        System.out.println(p2.name);
        Person p3 = p1;
        p3.age=10;

    }
}
```

对以上代码的内存布局图分析如下：



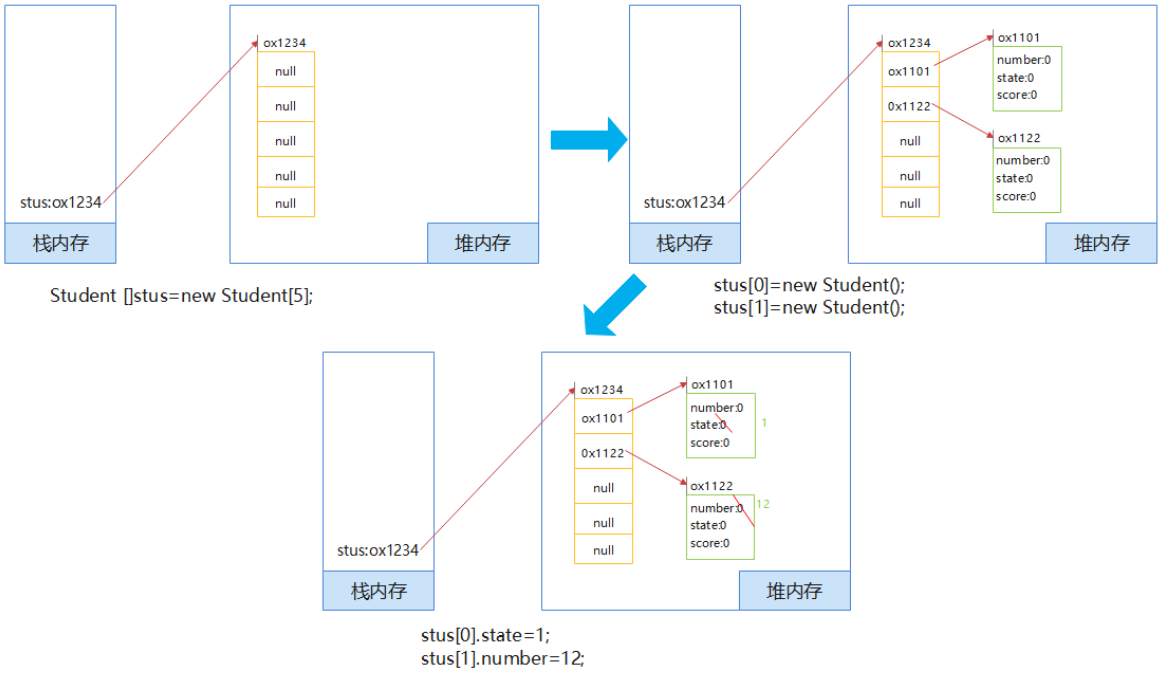
3）对象数组的内存解析

引用类型的对象不是null就是地址

```
public class Student {
    int number;
    int state;
    int score;

    public static void main(String[] args) {
        Student []stus=new Student[5];
        stus[0]=new Student();
        stus[1]=new Student();
        stus[0].state=1;
        stus[1].number=12;
        System.out.println(stus[0].state);
        System.out.println(stus[1].number);
    }
}
```

对以上代码的内存布局图分析如下：



(2) UML图绘制类之间的继承关系与组合关系

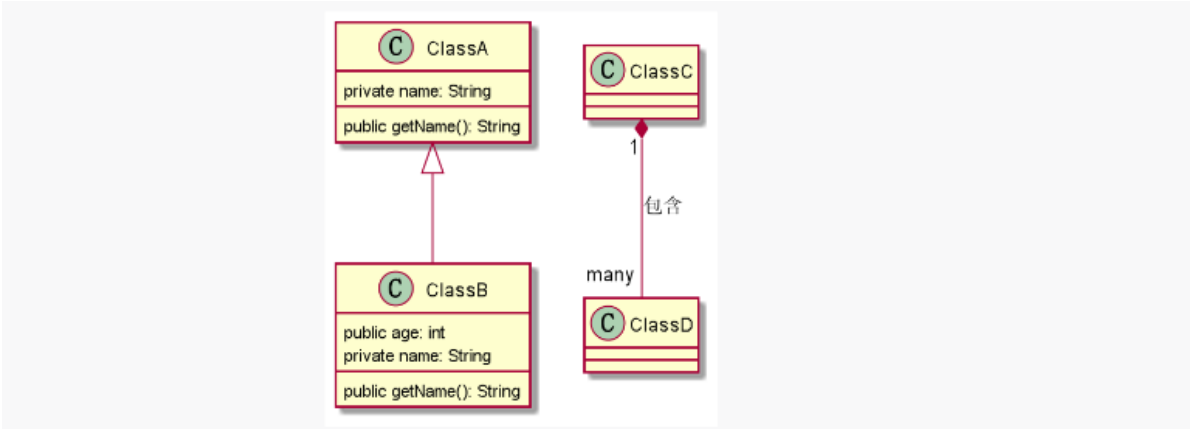
1）泛化与组合的概念

泛化关系就是继承，即找出当前一个类的属性和方法，或者若干类之间共同的属性和方法，构造出一个一般类，凡是具有这个一般类特征并且还有自身一些特殊特征的类为特殊类。一般类和特殊类之间的关系就是继承，一般类是父类，特殊类都是它的子类，这种系就是通常所说的　　关系。

组合是一种特殊的聚合（$has-a$），在这种关系中，作为整体类和作为局部类的生命周期是相辅相成的，一旦整体类的生命周期结束，局部类的生命周期也就结束了，它们之间是$contains-a$的关系。

2）泛化与组合的UML图表示

泛化是用一个空心三角箭头连接，组合是用一个实心菱形箭头连接



如上图所示，ClassA和ClassB是继承关系，ClassB继承自ClassA，ClassC和ClassD是组合关系，ClassC中包含ClassD，而且是一对多的关系。

3）泛化与组合的代码表示

泛化比较简单，如果一个类是另一个类的父类或者子类，即包含关键字extends，就说明这两个类具有泛化关系。

组合需要和聚合区别开来，其中一个重要的区分点就是，聚合中一个类可以离开另一个类而独立存在，组合中则不行，整体类和局部类是同生共死的。

代码表示如下：

```
class A{}
class B extends A{}//A和B具有泛化关系
class C{
    A a;
    C(A a){//以实例作为构造方法的参数
        this.a=a;
    }
}//A和C具有聚合关系
class D{
    A a;
    D(){
        A x = new A();//在构造方法内实例化
        this.a = x;
    }
}//A和D具有组合关系
```

参考资料

Java语言程序设计 吴倩主编

尚硅谷 30天搞定Java核心技术

CSDN——Java程序的运行过程

菜鸟教程—— Java 中 this 和 super 的用法总结

菜鸟教程——Java封装

菜鸟教程——java重写与重载

菜鸟教程——java多态