物联网中间件

第4章 分布式面向对象组件-JAVA简介

参考材料

- 菜鸟编程
  - https://www.runoob.com/java/java-tutorial.html

- 如果需要深入学习JAVA
  - 《Thinking in Java》、《Core Java》

大家对面向对象了解多少？

- A 了解面向对象相关概念，不熟悉面向对象编程语言
- B 了解面向对象相关概念，自己学习过一门面向对象编程语言
- C 不了解面向对象相关概念，也没有学习过面向对象编程语言

编程语言排行榜

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

Apr 2020	Apr 2019	Change	Programming Language	Ratings	Change
1	1		Java	16.73%	+1.69%
2	2		C	16.72%	+2.64%
3	4	↑	Python	9.31%	+1.15%
4	3	↓	C++	6.78%	-2.06%
5	6	↑	C#	4.74%	+1.23%
6	5	↓	Visual Basic	4.72%	-1.07%
7	7		JavaScript	2.38%	-0.12%
8	9	↑	PHP	2.37%	+0.13%
9	8	↓	SQL	2.17%	-0.10%
10	16	⇈	R	1.54%	+0.35%

Java 简介

- Java一开始也被称为C++--
- Java是由Sun Microsystems公司于1995年5月推出的Java面向对象程序设计语言和Java平台的总称。由James Gosling和同事们共同研发，并在1995年正式推出。

- Java分为三个体系：
  - JavaSE（J2SE）（Java2 Platform Standard Edition，java平台标准版）
  - JavaEE (J2EE) (Java 2 Platform,Enterprise Edition，java平台企业版)
  - JavaME (J2ME) (Java 2 Platform Micro Edition，java平台微型版)。

发展历史

- 1995年5月23日，Java语言诞生
- 2005年6月，JavaOne大会召开，SUN公司公开Java SE 6
- 2014年3月18日，Oracle公司发表 Java SE 8。
- 2017年9月21日，Oracle公司发表 Java SE 9
- 2018年3月21日，Oracle公司发表 Java SE 10
- 2018年9月25日，Java SE 11 发布
- 2019年3月20日，Java SE 12 发布

Java主要特性

- **Java语言是简单的：**
- Java语言的语法与C语言和C++语言很接近，使得大多数程序员很容易学习和使用。
- Java丢弃了C++中很少使用的、很难理解的、令人迷惑的那些特性，如操作符重载、多继承、自动的强制类型转换。
- Java语言不使用指针，而是引用。并提供了自动的内存管理，使得程序员不必为内存管理而担忧。

Java主要特性

- **Java语言是面向对象的：**
- Java语言提供类、接口和继承等面向对象的特性，
- 为了简单起见，只支持类之间的单继承，
- 但支持接口之间的多继承，并支持类与接口之间的实现机制（关键字为implements）。
- 总之，Java语言是一个纯的面向对象程序设计语言。

Java主要特性

- **Java语言是分布式的：**
- Java语言支持Internet应用的开发
- 在基本的Java应用编程接口中有一个网络应用编程接口（java net），它提供了用于网络应用编程的类库，包括URL、URLConnection、Socket、ServerSocket等。
- Java的RMI（远程方法激活）机制也是开发分布式应用的重要手段。

Java主要特性

- **Java语言是健壮的：**Java的强类型机制、异常处理、垃圾的自动收集等是Java程序健壮性的重要保证。对指针的丢弃是Java的明智选择。Java的安全检查机制使得Java更具健壮性。

Java主要特性

- **Java语言是体系结构中立的：**
- Java程序（后缀为java的文件）在Java平台上被编译为体系结构中立的字节码格式（后缀为class的文件），
- 可以在实现这个Java平台的任何系统中运行（Java 虚拟机）。这种途径适合于异构的网络环境和软件的分发。

Java主要特性

- **Java语言是可移植的：**
- 这种可移植性来源于体系结构中立性，
- Java还严格规定了各个基本数据类型的长度。
- Java系统本身也具有很强的可移植性，Java编译器是用Java实现的，Java的运行环境是用ANSI C实现的。

Java主要特性

- **Java语言是解释型的：**
- Java程序在Java平台上被编译为字节码格式，然后可以在实现这个Java平台的任何系统中运行。
- 在运行时，Java平台中的Java解释器对这些字节码进行解释执行，执行过程中需要的类在联接阶段被载入到运行环境中。

主要特性

- **Java是高性能的：**与那些解释型的高级脚本语言相比，Java的确是高性能的。事实上，Java的运行速度随着JIT(Just-In-Time）编译器技术的发展越来越接近于C++。

- **Java语言是多线程的：**Java语言支持多个线程的同时执行，并提供多线程之间的同步机制

Java 开发环境配置

- 下载JDK，安装JDK
- 配置环境变量

- **流行JAVA开发工具**
  - **JetBrains** 的 IDEA， 现在很多人开始使用了， 功能很强大， 下载地址: https://www.jetbrains.com/idea/download/

基本语法

- **大小写敏感**：Java 是大小写敏感的，这就意味着标识符 Hello 与 hello 是不同的。

- **类名**：对于所有的类来说，类名的首字母应该大写。如果类名由若干单词组成，那么每个单词的首字母应该大写，例如 **MyFirstJavaClass** 。

- **方法名**：所有的方法名都应该以小写字母开头。如果方法名含有若干单词，则后面的每个单词首字母大写。

基本语法

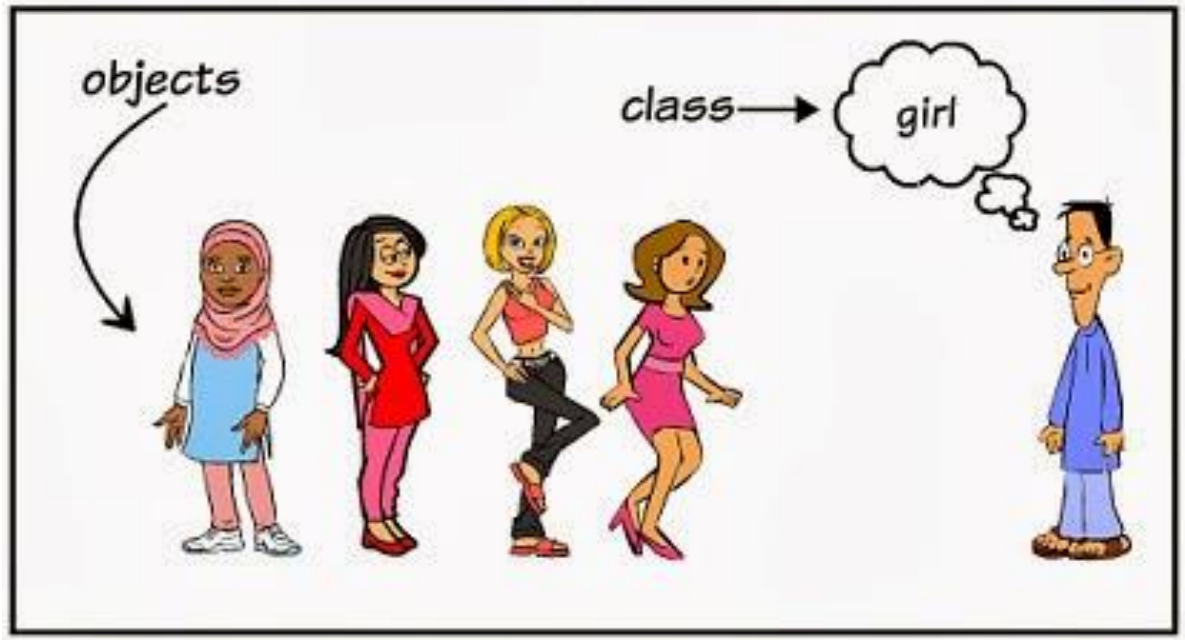
- **源文件名**：源文件名必须和类名相同。当保存文件的时候，你应该使用类名作为文件名保存（切记 Java 是大小写敏感的），文件名的后缀为 **.java**。（如果文件名和类名不相同则会导致编译错误）。

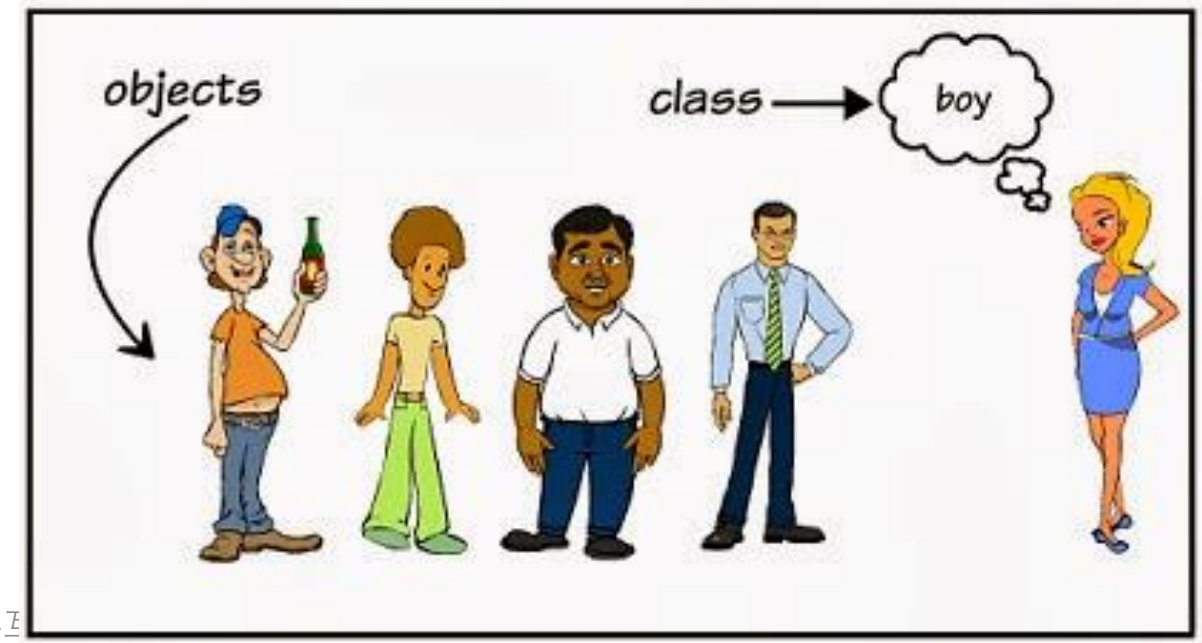
- **主方法入口**：所有的 Java 程序由 **public static void main(String []args)** 方法开始执行。

Java 对象和类

- **对象**：对象是类的一个实例，有状态和行为。

- **类**：类是一个模板，它描述一类对象的行为和状态。


objects
class
girl


objects
class
boy

Java中的对象

- 现在让我们深入了解什么是对象。看看周围真实的世界，会发现身边有很多对象，车，狗，人等等。所有这些对象都有自己的状态和行为。
- 拿一条狗来举例，它的状态有：名字、品种、颜色，行为有：叫、摇尾巴和跑。
- 对比现实对象和软件对象，它们之间十分相似。
- 软件对象也有状态和行为。软件对象的状态就是属性，行为通过方法体现。
- 在软件开发中，方法操作对象内部状态的改变，对象的相互调用也是通过方法来完成。

Java中的类

- 类可以看成是创建Java对象的模板。
- 通过下面一个简单的类来理解下Java中类的定义:

```
public class Dog{
   String breed;
   int age;
   String color;
   void barking(){
   }

   void hungry(){
   }

   void sleeping(){
   }
}
```

```java
public class Dog{
   String breed;
   int age;
   String color;
   void barking(){
   }

   void hungry(){
   }

   void sleeping(){
   }
}
```

```java
struct Dog{
   String breed;
   int age;
   String color;
};

 void barking(){
  }

 void hungry(){
  }

 void sleeping(){
 }
```

类型变量

- **局部变量**：在方法、构造方法或者语句块中定义的变量被称为局部变量。变量声明和初始化都是在方法中，方法结束后，变量就会自动销毁。
- **成员变量**：成员变量是定义在类中，方法体之外的变量。这种变量在创建对象的时候实例化。成员变量可以被类中方法、构造方法和特定类的语句块访问。
- **类变量**：类变量也声明在类中，方法体之外，但必须声明为static类型。

```
public class Variable{
    static int allClicks=0;    // 类变量

    String str="hello world";  // 实例变量

    public void method(){

        int i =0;  // 局部变量

    }
}
```

创建对象

- 对象是根据类创建的。在Java中，使用关键字new来创建一个新的对象。创建对象需要以下三步:
- **声明**：声明一个对象，包括对象名称和对象类型。
- **实例化**：使用关键字new来创建一个对象。
- **初始化**：使用new创建对象时，会调用构造方法初始化对象。

```
public class Puppy{
  public Puppy(String name){
    //这个构造器仅有一个参数：name
    System.out.println("小狗的名字是 : " + name );
  }
  public static void main(String[] args){
    // 下面的语句将创建一个Puppy对象
    Puppy myPuppy = new Puppy( "tommy" );
  }
}
```

编译并运行上面的程序，会打印出下面的结果：

小狗的名字是 : tommy

访问类中的实例变量和方法

```
/* 实例化对象 */
Object referenceVariable = new Constructor();
/* 访问类中的变量 */
referenceVariable.variableName;
/* 访问类中的方法 */
referenceVariable.methodName();
```

内置数据类型

Java语言提供了八种基本类型。六种数字类型（四个整数型，两个浮点型），一种字符类型，还有一种布尔型。

byte

- byte 数据类型是8位、有符号的，以二进制补码表示的整数;
- 最小值是 **-128（-2^7）**；
- 最大值是 **127（2^7-1）**；
- 默认值是 **0**;
- byte 类型用在大型数组中节约空间，主要代替整数，因为 byte 变量占用的空间只有 int 类型的四分之一;
- 例子：byte a = 100，byte b = -50。

short

- short 数据类型是 16 位、有符号的以二进制补码表示的整数
- 最小值是 **-32768（-2^15）**；
- 最大值是 **32767（2^15 - 1）**；
- Short 数据类型也可以像 byte 那样节省空间。一个short变量是int型变量所占空间的二分之一；
- 默认值是 **0**;
- 例子：short s = 1000，short r = -20000。

int

- int 数据类型是32位、有符号的以二进制补码表示的整数;
- 最小值是 **-2,147,483,648（-2^31）**；
- 最大值是 **2,147,483,647（2^31 - 1）**；
- 一般地整型变量默认为 int 类型;
- 默认值是 **0**；
- 例子：int a = 100000, int b = -200000。

long

- long 数据类型是 64 位、有符号的以二进制补码表示的整数;
- 最小值是 **-9,223,372,036,854,775,808（-2^63）**;
- 最大值是 **9,223,372,036,854,775,807（2^63 -1）**;
- 这种类型主要使用在需要比较大整数的系统上;
- 默认值是 **0L**;
- 例子： long a = 100000L，Long b = -200000L。
  "L"理论上不分大小写，但是若写成"l"容易与数字"1"混淆，不容易分辨。所以最好大写。

float

- float 数据类型是单精度、32位、符合IEEE 754标准的浮点数;
- float 在储存大型浮点数组的时候可节省内存空间;
- 默认值是 **0.0f**;
- 浮点数不能用来表示精确的值，如货币;
- 例子：float f1 = 234.5f。

浮点数的表示方法

- 国际标准IEEE 754规定，任意一个二进制浮点数V都可以表示成下列形式：

$$V = (-1)^s * M * 2^E$$

- (-1)^s 表示符号位，当s=0，V为整数；s=1，V为负数；
- M 表示有效数字，1≤M<2；
- 2^E 表示指数位

浮点数的表示方法

IEEE 单精度浮点数

符号 Sign	指数 Exponent	尾数 Mantissa
1 bit	8 bits	23 bits

IEEE 双精度浮点数

符号 Sign	指数 Exponent	尾数 Mantissa
1 bit	11 bits	52 bits

浮点数的表示方法

0.6根本无法用2进制在有限位数内精确表示，实际上，$0.6 = 0.\dot{1}00\dot{1}_2$ ，是个无限循环小数。

那么，0.6 这样的数在计算机里怎么表示呢?

很简单，舍入保留有限位数。相对于十进制的“四舍五入”，二进制的“0舍1入”更简单。

所以，如果保留12个二进制位，$0.6 \approx 0.100110011010_2$ 。但是如果再把它转化成十进制，$0.100110011010_2 = 0.600098$ 。

误差就是从这里产生的。

double

- double 数据类型是双精度、64 位、符合IEEE 754标准的浮点数;
- 浮点数的默认类型为double类型;
- double类型同样不能表示精确的值，如货币;
- 默认值是 **0.0d**;
- 例子：double d1 = 123.4。

boolean

- boolean数据类型表示一位的信息；
- 只有两个取值：true 和 false；
- 这种类型只作为一种标志来记录 true/false 情况；
- 默认值是 **false**；
- 例子：boolean one = true。

char

- char类型是一个单一的 16 位 Unicode 字符;
- 最小值是 **\u0000**（即为0）；
- 最大值是 **\uffff**（即为65,535）；
- char 数据类型可以储存任何字符;
- 例子： char letter = 'A';。

类型默认值

数据类型	默认值
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'u0000'
String (or any object)	null
boolean	false

- DefaultValue.java

Java 常量

- 常量在程序运行时是不能被修改的。
- 在 Java 中使用 final 关键字来修饰常量，声明方式和变量类似:
- final double PI = 3.1415927;

Java 变量

- 在Java语言中，所有的变量在使用前必须声明。声明变量的基本格式如下：
- int a, b, c;　　// 声明三个int型整数：a、 b、c
- int d = 3, e = 4, f = 5; // 声明三个整数并赋予初值
- byte z = 22;　　// 声明并初始化 z
- String s = "runoob"; // 声明并初始化字符串 s
- double pi = 3.14159; // 声明了双精度浮点型变量 pi
- char x = 'x';　　// 声明变量 x 的值是字符 'x'。

Java包（package）

- 包主要用来对类和接口进行分类。当开发Java程序时，可能编写成百上千的类，因此很有必要对类和接口进行分类。

- 包 的作用就是防止名字相同的类产生冲突。

- Java 编译器在编译时，直接根据 package 指定的信息，将 class 文件生成到对应目录下。如 **package aaa.bbb.ccc;** 编译器就将该 .java 文件下的各个类生成到 **./aaa/bbb/ccc/** 这个目录。

包的作用

- 1、把功能相似或相关的类或接口组织在同一个包中，方便类的查找和使用。
- 2、如同文件夹一样，包也采用了树形目录的存储方式。同一个包中的类名字是不同的，不同的包中的类的名字是可以相同的，当同时调用两个不同包中相同类名的类时，应该加上包名加以区别。因此，包可以避免名字冲突。
- 3、包也限定了访问权限，拥有包访问权限的类才能访问某个包中的类

创建包

- 创建包的时候，你需要为这个包取一个合适的名字。之后，如果其他的一个源文件包含了这个包提供的类、接口、枚举或者注释类型的时候，都必须将这个包的声明放在这个源文件的开头。

- 包声明应该在源文件的第一行，每个源文件只能有一个包声明

- 如果一个源文件中没有使用包声明，那么其中的类，函数，枚举，注释等将被放在一个无名的包（unnamed package）中。

package 的目录结构

- 类放在包中会有两种主要的结果:
- 包名成为类名的一部分，正如我们前面讨论的一样。
- 包名必须与相应的字节码所在的目录结构相吻合。

package 的目录结构

- 例如,一个Something.java 文件它的内容
- package net.java.util;
- public class Something{
- ...
- }
- 那么它的路径应该是 net/java/util/Something.java 这样保存的。package(包) 的作用是把不同的 java 程序分类保存，更方便的被其他 java 程序调用。

Import语句

- 在Java中，如果给出一个完整的限定名，包括包名、类名，那么Java编译器就可以很容易地定位到源代码或者类。Import语句就是用来提供一个合理的路径，使得编译器可以找到某个类。

- 例如，下面的命令行将会命令编译器载入java_installation/java/io路径下的所有类
- import java.io.*;

Import语句

- import 是为了简化使用 package 之后的实例化的代码。

- 假设 **./aaa/bbb/ccc/** 下的 A 类，假如没有 import，实例化A类为：**new aaa.bbb.ccc.A()**，使用 **import aaa.bbb.ccc.A** 后，就可以直接使用 **new A()** 了，也就是编译器匹配并扩展了 **aaa.bbb.ccc.** 这串字符串。

Java 修饰符

- Java语言提供了很多修饰符，主要分为以下两类：
- 访问修饰符
- 非访问修饰符

访问控制修饰符

- Java中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java 支持 4 种不同的访问权限。
- **default** (即默认，什么也不写）: 在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法。
- **private** : 在同一类内可见。使用对象：变量、方法。 **注意：不能修饰类（外部类）**
- **public** : 对所有类可见。使用对象：类、接口、变量、方法
- **protected** : 对同一包内的类和所有子类可见。使用对象：变量、方法。 **注意：不能修饰类（外部类）**。

私有访问修饰符-private

- 私有访问修饰符是最严格的访问级别，所以被声明为 **private** 的方法、变量和构造方法只能被所属类访问，并且类和接口不能声明为 **private**。

- 声明为私有访问类型的变量只能通过类中公共的 getter 方法被外部类访问。

- Private 访问修饰符的使用主要用来隐藏类的实现细节和保护类的数据。

Java 封装

- 在面向对象程式设计方法中，封装（英语：Encapsulation）是指一种将抽象性函式接口的实现细节部分包装、隐藏起来的方法。

- 封装可以被认为是一个保护屏障，防止该类的代码和数据被外部类定义的代码随机访问。

Java 封装

- 要访问该类的代码和数据，必须通过严格的接口控制。
- 封装最主要的功能在于我们能修改自己的实现代码，而不用修改那些调用我们代码的程序片段。
- 适当的封装可以让程式码更容易理解与维护，也加强了程式码的安全性。

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. 隐藏信息以及实现细节。

EncapTest.java

RunEncap.java

默认访问修饰符-不使用任何关键字

- 使用默认访问修饰符声明的变量和方法，对同一个包内的类是可见的。
- 接口里的变量都隐式声明为 **public static final**, 而接口里的方法默认情况下访问权限为 **public**。
- 如下例所示，变量和方法的声明可以不使用任何修饰符。

- String version = "1.5.1";
- boolean processOrder() {
- return true;
- }

公有访问修饰符-public

- 被声明为 public 的类、方法、构造方法和接口能够被任何其他类访问。

- 如果几个相互访问的 public 类分布在不同的包中，则需要导入相应 public 类所在的包。由于类的继承性，类所有的公有方法和变量都能被其子类继承。

- Java 程序的 main() 方法必须设置成公有的，否则，Java 解释器将不能运行该类。

受保护的访问修饰符-protected

- protected 需要从以下两个点来分析说明:
- **子类与基类在同一包中**：被声明为 protected 的变量、方法和构造器能被同一个包中的任何其他类访问;
- **子类与基类不在同一包中**：那么在子类中，子类实例可以访问其从基类继承而来的 protected 方法，而不能访问基类实例的protected方法。

访问控制和继承

- 请注意以下方法继承的规则:
- 父类中声明为 public 的方法在子类中也必须为 public。
- 父类中声明为 protected 的方法在子类中要么声明为 protected，要么声明为 public，不能声明为 private。
- 父类中声明为 private 的方法，不能够被继承

非访问修饰符

- 为了实现一些其他的功能，Java 也提供了许多非访问修饰符。
- static 修饰符，用来修饰类方法和类变量。
- final 修饰符，用来修饰类、方法和变量，final 修饰的类不能够被继承，修饰的方法不能被继承类重新定义，修饰的变量为常量，是不可修改的。
- abstract 修饰符，用来创建抽象类和抽象方法。
- synchronized 和 volatile 修饰符，主要用于线程的编程。

static 修饰符

- **静态变量：**
- static 关键字用来声明独立于对象的静态变量，无论一个类实例化多少对象，它的静态变量只有一份拷贝。 静态变量也被称为类变量。局部变量不能被声明为 static 变量。
- **静态方法：**
- static 关键字用来声明独立于对象的静态方法。静态方法不能使用类的非静态变量。静态方法从参数列表得到数据，然后计算这些数据。

static 修饰符

- 静态方法可以直接通过类名.静态方法名来调用;
- 实例方法则必须通过实例对象来调用实例方法

static 修饰符

- InstanceCounter.java

final 修饰符

- **final 变量：**
- final 表示"最后的、最终的"含义，变量一旦赋值后，不能被重新赋值。被 final 修饰的实例变量必须显式指定初始值。

- final 修饰符通常和 static 修饰符一起使用来创建类常量。

```
public class Test{
  final int value = 10;
  // 下面是声明常量的实例
  public static final int BOXWIDTH = 6;
  static final String TITLE = "Manager";

  public void changeValue(){
     value = 12; //将输出一个错误
  }
}
```

final 方法

- 父类中的 final 方法可以被子类继承，但是不能被子类重写。
- 声明 final 方法的主要目的是防止该方法的内容被修改。
- 如下所示，使用 final 修饰符声明方法。

```
public class Test{
    public final void changeName(){
        // 方法体
    }
}
```

final 类

- final 类不能被继承，没有类能够继承 final 类的任何特性。

- public final class Test {
- // 类体
- }

abstract 修饰符

- **抽象类：**
- 抽象类不能用来实例化对象，声明抽象类的唯一目的是为了将来对该类进行扩充。
- 一个类不能同时被 abstract 和 final 修饰。如果一个类包含抽象方法，那么该类一定要声明为抽象类，否则将出现编译错误。
- 抽象类可以包含抽象方法和非抽象方法。

```
abstract class Caravan{
    private double price;
    private String model;
    private String year;
    public abstract void goFast(); //抽象方法
    public abstract void changeColor();
}
```

抽象方法

- 抽象方法是一种没有任何实现的方法，该方法的的具体实现由子类提供。
- 抽象方法不能被声明成 final 和 static。
- 任何继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类。
- 如果一个类包含若干个抽象方法，那么该类必须声明为抽象类。抽象类可以不包含抽象方法。
- 抽象方法的声明以分号结尾，例如：**public abstract sample();**。

```
public abstract class SuperClass{
    abstract void m(); //抽象方法
}

class SubClass extends SuperClass{
    //实现抽象方法
     void m(){
          .....
     }
}
```

synchronized 修饰符

- synchronized 关键字声明的方法同一时间只能被一个线程访问。synchronized 修饰符可以应用于四个访问修饰符。

transient 修饰符

- 序列化的对象包含被 transient 修饰的实例变量时，java 虚拟机(JVM)跳过该特定的变量。
- 该修饰符包含在定义变量的语句中，用来预处理类和变量的数据类型。
- public transient int limit = 55; // 不会序列化
- public int b; //序列化

volatile 修饰符

- volatile 修饰的成员变量在每次被线程访问时，都强制从共享内存中重新读取该成员变量的值。而且，当成员变量发生变化时，会强制线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

Java 运算符

- 计算机的最基本用途之一就是执行数学运算，作为一门计算机语言，Java也提供了一套丰富的运算符来操纵变量。我们可以把运算符分成以下几组:
- 算术运算符
- 关系运算符
- 位运算符
- 逻辑运算符
- 赋值运算符
- 其他运算符

Java 循环结构 - for, while 及 do...while

- 顺序结构的程序语句只能被执行一次。如果您想要同样的操作执行多次,，就需要使用循环结构。
- Java中有三种主要的循环结构:
- **while** 循环
- **do…while** 循环
- **for** 循环

Java 增强 for 循环

- Java5 引入了一种主要用于数组的增强型 for 循环。

- Java 增强 for 循环语法格式如下
- for(声明语句 : 表达式)
- {
- 　//代码句子
- }

Java 增强 for 循环

- **声明语句：**声明新的局部变量，该变量的类型必须和数组元素的类型匹配。其作用域限定在循环语句块，其值与此时数组元素的值相等。
- **表达式：**表达式是要访问的数组名，或者是返回值为数组的方法。

- **ForTest.java**

Java中的数据类别

- 目前Java中的数据类别分为两种，一种是primitive（原生类型），另一种就是object（对象类型）。

- 那就是原生类型在作为参数传递到方法中时使用的是“值传递”的方式，而对象作为参数传递到方法中时使用的是“引用传递”的方式

Wrapping Class

- Wrapping Class，意思是指原生类型的对象类，这里我们简称包装类。
- 那么什么是包装类呢？很简单，JAVA内部对上述的8种原生类型提供了8个对象类，而这8个对象类就是包装类，它们的对应关系如下：

基本数据类型	包装类
byte	Byte
boolean	Boolean
boolean	Boolean
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

为什么要设计出包装类

- 第一：如果你想在方法体内更新primitive类型的值，必须要使用primitive对应的object，因为前者使用的值传递，后者使用的是引用传递。
- 第二：java.util内操作的都是对象，如果没有包装类，会让程序员在使用这些工具类操作primitive类型时编写额外的代码。
- 第三：Java提供的集合框架中的数据结构，比如ArrayList和Vector，也是只能操作对象，理由跟第二点相似。
- 第四：多线程中也必须使用对象来完成各种同步操作。

装箱（boxing）拆箱（unboxing）

- 既然有了基本类型和包装类型，肯定有些时候要在它们之间进行转换。把基本类型转换成包装类型的过程叫做装箱（boxing）。反之，把包装类型转换成基本类型的过程叫做拆箱（unboxing）
- 在 Java SE5 之前，开发人员要手动进行装拆箱，比如说：

- Integer chenmo = new Integer(10); // 手动装箱
- int wanger = chenmo.intValue(); // 手动拆箱

自动装箱与自动拆箱

- Java SE5 为了减少开发人员的工作，提供了自动装箱与自动拆箱的功能。

- Integer chenmo = 10; // 自动装箱
- int wanger = chenmo; // 自动拆箱

Java String 类

- 字符串广泛应用 在 Java 编程中，在 Java 中字符串属于对象
- Java 提供了 String 类来创建和操作字符串。
  - String className = “Middleware";

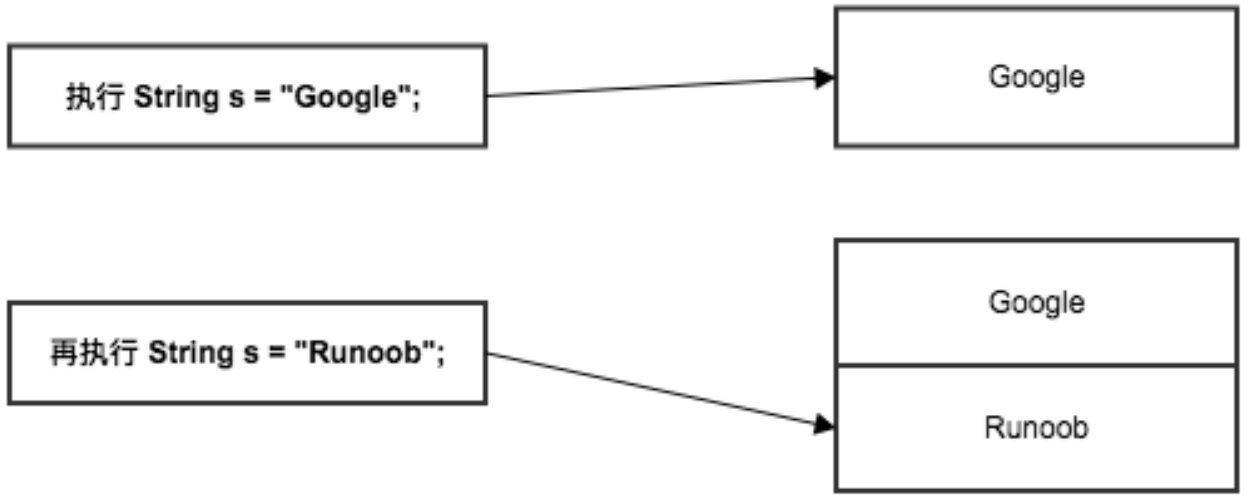
- String 类是不可改变的，所以你一旦创建了 String 对象，那它的值就无法改变了。
- 如果需要对字符串做很多修改，那么应该选择使用 StringBuffer & StringBuilder 类。

String 类是不可改变的解析

- String s = "Google";
- System.out.println("s = " + s);
- s = "Runoob";
- System.out.println(“s = ” + s);

- 输出结果为:
- Google
- Runoob

原因在于实例中的 s 只是一个 **String** 对象的引用，并不是对象本身，当执行 **s = "Runoob";** 创建了一个新的对象 "Runoob"，而原来的 "Google" 还存在于内存中。



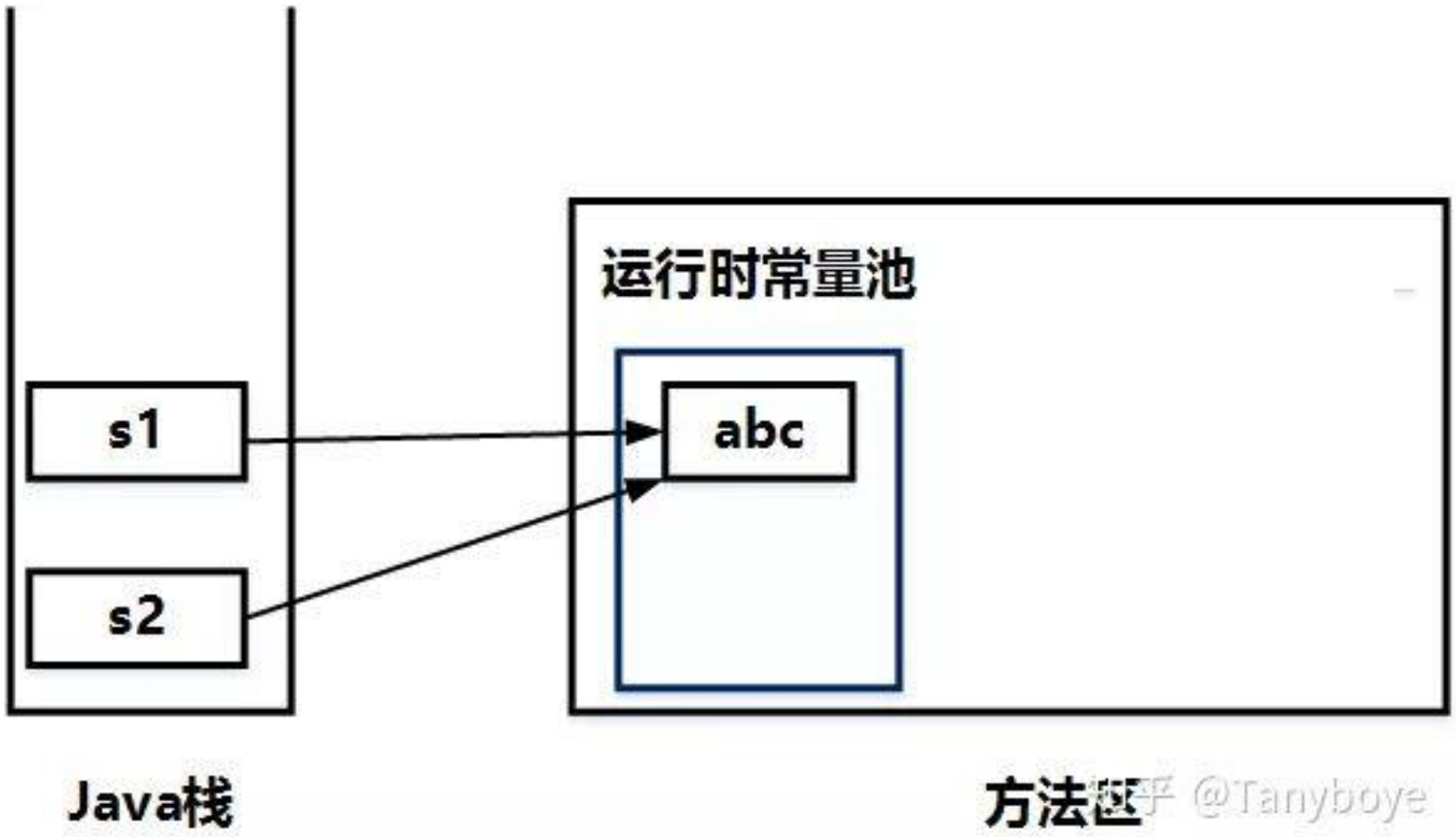
String 类是不可改变的解析

- String s = "abc";
- 执行上述代码时，JVM首先在运行时常量池中查看是否存在String对象“abc”，如果已存在该对象，则不用创建新的String对象“abc”，而是将引用s直接指向运行时常量池中已存在的String对象“abc”；如果不存在该对象，则先在运行时常量池中创建一个新的String对象“abc”，然后将引用s指向运行时常量池中创建的新String对象。

String s1 = "abc";

String s2 = "abc";

执行上述代码时，在运行时常量池中只会创建一个String对象"abc"，这样就节省了内存空间


运行时常量池
s1
s2
abc
Java栈
方法区
知乎 @Tanyboye

Java StringBuffer 和 StringBuilder 类

- 当对字符串进行修改的时候，需要使用 StringBuffer 和 StringBuilder 类。
- 和 String 类不同的是，StringBuffer 和 StringBuilder 类的对象能够被多次的修改，并且不产生新的未使用对象。
- StringBuilder 类在 Java 5 中被提出，它和 StringBuffer 之间的最大不同在于 StringBuilder 的方法不是线程安全的（不能同步访问）。
- 由于 StringBuilder 相较于 StringBuffer 有速度优势，所以多数情况下建议使用 StringBuilder 类。然而在应用程序要求线程安全的情况下，则必须使用 StringBuffer 类

- TestStringBuffer.java

Java 方法

- 在前面几个章节中我们经常使用到 **System.out.println()**，那么它是什么呢?
- println() 是一个方法。
- System 是系统类。
- out 是标准输出对象。
- 这句话的用法是调用系统类 System 中的标准输出对象 out 中的方法 println()。

那么什么是方法呢？

- Java方法是语句的集合，它们在一起执行一个功能。
- 方法是解决一类问题的步骤的有序组合
- 方法包含于类或对象中
- 方法在程序中被创建，在其他地方被引用

方法的命名规则

- 1.方法的名字的第一个单词应以小写字母作为开头，后面的单词则用大写字母开头写，不使用连接符。例如：**addPerson**。

Java 流(Stream)、文件(File)和I/O

- 一个流可以理解为一个数据的序列。输入流表示从一个源读取数据，输出流表示向一个目标写数据。

- Java 为 I/O 提供了强大的而灵活的支持，使其更广泛地应用到文件传输和网络编程中。

Java 流(Stream)、文件(File)和IO

- Java.io 包几乎包含了所有操作输入、输出需要的类。所有这些流类代表了输入源和输出目标。

- Java.io 包中的流支持很多种格式，比如：基本类型、对象、本地化字符集等等。

读取控制台输入

- Java 的控制台输入由 System.in 完成。
- 为了获得一个绑定到控制台的字符流，你可以把 System.in 包装在一个 BufferedReader 对象中来创建一个字符流。
- 下面是创建 BufferedReader 的基本语法：
- BufferedReader br = new BufferedReader(new InputStreamReader (System.in));
- BufferedReader 对象创建后，我们便可以使用 read() 方法从控制台读取一个字符，或者用 readLine() 方法读取一个字符串。

InputStreamReader

- 字符流InputStreamReader是Reader的子类；是字节流通向字符流的桥梁,也就是可以把字节流转化为字符流。

- InputStreamReader 构造方法：
  - InputStreamReader(Inputstream in)
  - 创建一个使用默认字符集的 InputStreamReader。

InputStreamReader

- 我们之所以要用到字符流，是因为用字符流处理中文比较方便；字节流处理8位数据，而字符流则是用于处理16位数据；
- 每次调用 InputStreamReader 中的一个 read() 方法都会导致从底层输入流读取一个或多个字节。也就是要实现一次从字节流转化为字符流的过程；

- 在实际使用中，为了提高效率，我们一般用，BufferedReader

BufferedReader：缓冲字符流

BufferedReader处理字符流是比较方便的，它可以处理一行数据;直接从文件中读取一行字符串;

- BufferedReader一共有两个构造方法:
- 一、创建一个使用默认大小输入缓冲区的缓冲字符输入流。
  - BufferedReader o=new BufferedReader(reader);

BufferedReader：缓冲字符流

二、创建一个使用指定大小输入缓冲区的缓冲字符输入流。

BufferedReader i=new BufferedReader(reader, sz);

传入一个Read类（用于读取字符流的抽象类）的对象；并且传入一个int型，指定输入缓冲区的大小，没有指定的话是使用默认的大小。大多数情况下，默认值就足够大了。

- BRRead.java
- BRReadLine.java

FileInputStream

- 该流用于从文件读取数据，它的对象可以用关键字 new 来创建。
- 有多种构造方法可用来创建对象。
- 可以使用字符串类型的文件名来创建一个输入流对象来读取文件
- InputStream f = new FileInputStream("C:/java/hello");

FileInputStream

- 也可以使用一个文件对象来创建一个输入流对象来读取文件。我们首先得使用 File() 方法来创建一个文件对象：

- File f = new File("C:/java/hello");
- InputStream out = new FileInputStream(f);

FileOutputStream

- 该类用来创建一个文件并向文件中写数据。
- 如果该流在打开文件进行输出前，目标文件不存在，那么该流会创建该文件。
- 有两个构造方法可以用来创建 FileOutputStream 对象。

FileOutputStream

- 使用字符串类型的文件名来创建一个输出流对象:
- OutputStream f = new FileOutputStream("C:/java/hello")
- 也可以使用一个文件对象来创建一个输出流来写文件。我们首先得使用File()方法来创建一个文件对象:
- File f = new File("C:/java/hello");
- OutputStream f = new FileOutputStream(f);

- FileStreamTest.java

Java 异常处理

- 异常是程序中的一些错误，但并不是所有的错误都是异常，并且错误有时候是可以避免的。

- 比如说，你的代码少了一个分号，那么运行出来结果是提示是错误 java.lang.Error；如果你用System.out.println(11/0)，那么你是因为你用0做了除数，会抛出 java.lang.ArithmeticException 的异常。

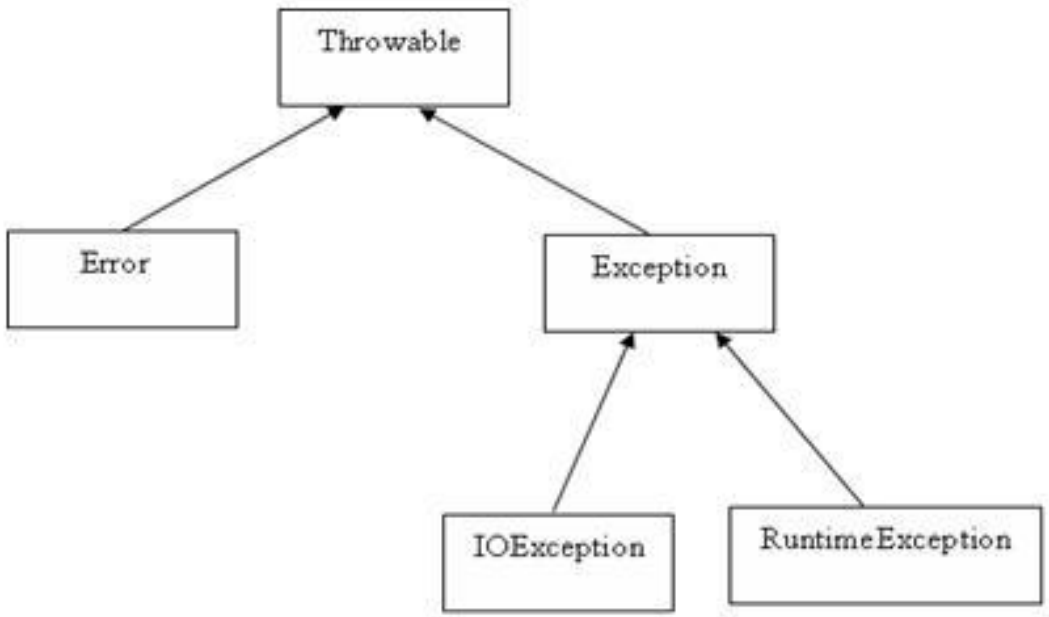
Java 异常处理

- 异常发生的原因有很多，通常包含以下几大类:

- 用户输入了非法数据。
- 要打开的文件不存在。
- 网络通信时连接中断，或者JVM内存溢出。

Exception 类的层次

- 所有的异常类是从 java.lang.Exception 类继承的子类。
- 异常类有两个主要的子类：IOException 类和 RuntimeException 类。


Throwable
Error
Exception
IOException
RuntimeException

捕获异常

- 使用 try 和 catch 关键字可以捕获异常。try/catch 代码块放在异常可能发生的地方。

- try/catch代码块中的代码称为保护代码，使用 try/catch 的语法如下：

捕获异常

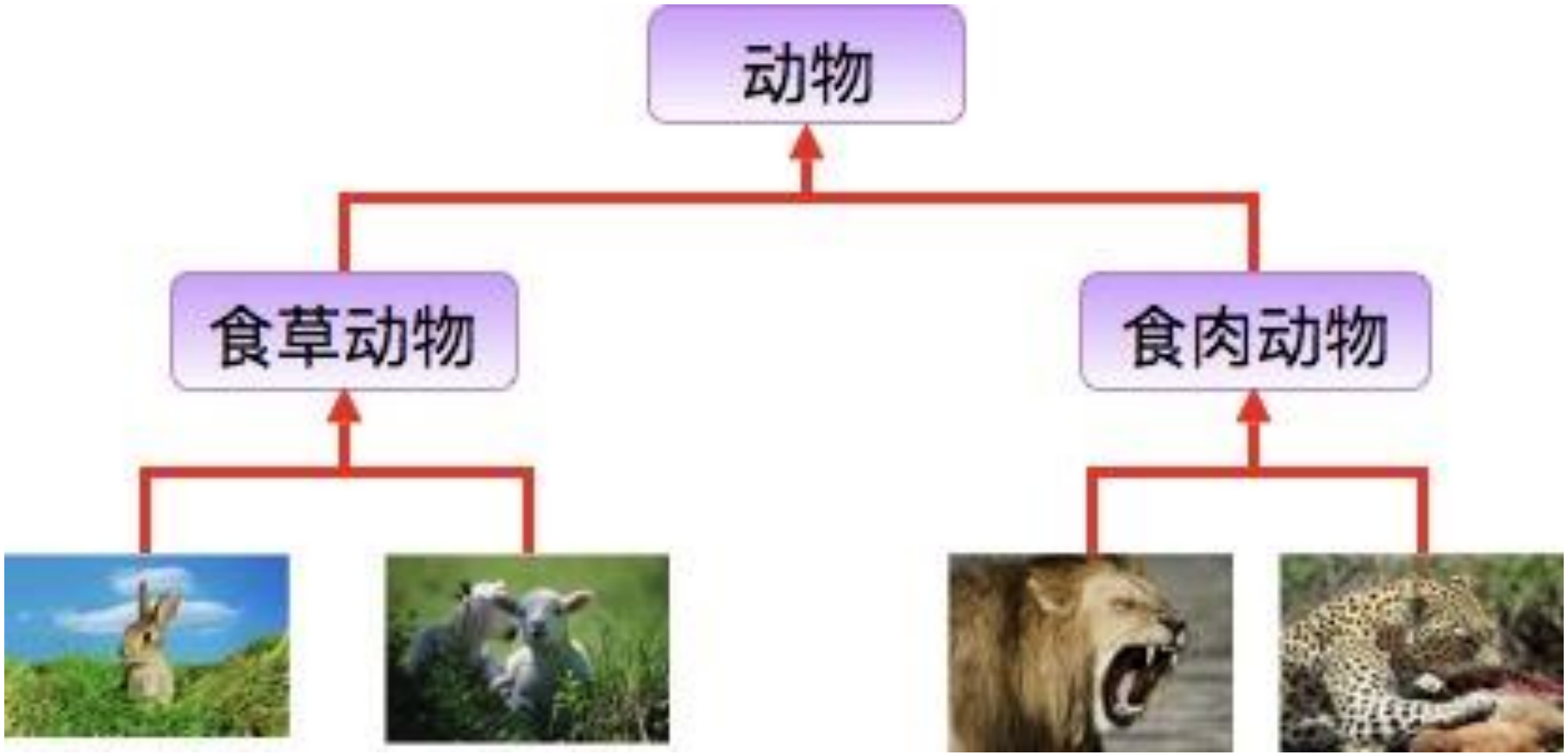
```
try
{
    // 程序代码
}catch(ExceptionName e1)
{
    //Catch 块
}
```

- ExcepTest.java

Java 继承

- 继承的概念
- 继承是java面向对象编程技术的一块基石，因为它允许创建分等级层次的类。

- 继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为


动物
食草动物
食肉动物

类的继承格式

- 在 Java 中通过 extends 关键字可以申明一个类是从另外一个类继承而来的。
- 类的继承格式
- class 父类 {
- }
-
- class 子类 extends 父类 {
- }

继承的特性

- 子类拥有父类非 private 的属性、方法。
- 子类可以拥有自己的属性和方法，即子类可以对父类进行扩展。
- 子类可以用自己的方式实现父类的方法。

继承的特性

- Java 的继承是单继承，但是可以多重继承，单继承就是一个子类只能继承一个父类，多重继承就是，例如 A 类继承 B 类，B 类继承 C 类，所以按照关系就是 C 类是 B 类的父类，B 类是 A 类的父类，这是 Java 继承区别于 C++ 继承的一个特性。

- 提高了类之间的耦合性（继承的缺点，耦合度高就会造成代码之间的联系越紧密，代码独立性越差）。

继承关键字

- 继承可以使用 extends 和 implements 这两个关键字来实现继承，而且所有的类都是继承于 java.lang.Object，当一个类没有继承的两个关键字，则默认继承object（这个类在 **java.lang** 包中，所以不需要 **import**）祖先类。

extends关键字

- 在 Java 中，类的继承是单一继承，也就是说，一个子类只能拥有一个父类，所以 extends 只能继承一个类。

- **extendsExample.java**
- Mouse.Java
- Penguin.Java

implements关键字

- 使用 implements 关键字可以变相的使java具有多继承的特性，使用范围为类继承接口的情况，可以同时继承多个接口（接口跟接口之间采用逗号分隔）。

```
public interface A {
    public void eat();
    public void sleep();
}

public interface B {
    public void show();
}

public class C implements A,B {
}
```

super 与 this 关键字

- super关键字：我们可以通过super关键字来实现对父类成员的访问，用来引用当前对象的父类。

- this关键字：指向自己的引用。

- SuperThisTest.java

Java 抽象类

- 在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

- 抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

抽象方法

- 如果你想设计这样一个类，该类包含一个特别的成员方法，该方法的具体实现由它的子类确定，那么你可以在父类中声明该方法为抽象方法。

- Abstract 关键字同样可以用来声明抽象方法，抽象方法只包含一个方法名，而没有方法体。

- 声明抽象方法会造成以下两个结果:
- 如果一个类包含抽象方法，那么该类必须是抽象类。
- 任何子类必须重写父类的抽象方法，或者声明自身为抽象类。

- 继承抽象方法的子类必须重写该方法。否则，该子类也必须声明为抽象类。最终，必须有子类实现该抽象方法，否则，从最初的父类到最终的子类都不能用来实例化对象。

- Employee.java
- Salary.java
- AbstractDemo.java

Java 接口

- 接口（英文：Interface），在JAVA编程语言中是一个抽象类型，是抽象方法的集合，接口通常以interface来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

- 接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法

Java 接口

- 除非实现接口的类是抽象类，否则该类要定义接口中的所有方法

- 接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。

接口与类相似点：

- 一个接口可以有多个方法。
- 接口文件保存在 .java 结尾的文件中，文件名使用接口名。
- 接口的字节码文件保存在 .class 结尾的文件中。
- 接口相应的字节码文件必须在与包名称相匹配的目录结构中。

接口与类的区别:

- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口中所有的方法必须是抽象方法。
- 接口不能包含成员变量，除了 static 和 final 变量。
- 接口不是被类继承了，而是要被类实现。
- 接口支持多继承。

接口特性

- 接口中每一个方法也是隐式抽象的,接口中的方法会被隐式的指定为 public abstract（只能是 public abstract，其他修饰符都会报错）。
- 接口中可以含有变量，但是接口中的变量会被隐式的指定为 public static final 变量（并且只能是 public，用 private 修饰会报编译错误）。
- 接口中的方法是不能在接口中实现的，只能由实现接口的类来实现接口中的方法。

抽象类和接口的区别

- 1. 抽象类中的方法可以有方法体，就是能实现方法的具体功能，但是接口中的方法不行。
- 2. 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 public static final 类型的。
- 3. 接口中不能含有静态代码块以及静态方法(用 static 修饰的方法)，而抽象类是可以有静态代码块和静态方法。
- 4. 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

接口的声明

- 接口的声明语法格式如下:

- [可见度] interface 接口名称 [extends 其他的接口名] {
- // 声明变量
- // 抽象方法
- }

接口有以下特性:

- 接口是隐式抽象的，当声明一个接口的时候，不必使用**abstract**关键字。
- 接口中每一个方法也是隐式抽象的，声明时同样不需要**abstract**关键字。
- 接口中的方法都是公有的。

- MammalInt.java

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。

- Override.java

方法的重写规则

- 参数列表必须完全与被重写方法的相同。
- 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）。
- 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。

方法的重写规则

- 父类的成员方法只能被它的子类重写。
- 声明为 final 的方法不能被重写。
- 声明为 static 的方法不能被重写，但是能够被再次声明。
- 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。
- 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。

方法的重写规则

- 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。

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。

- Overloading.java

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。

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）

重写与重载之间的区别

- 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。

Overriding 重写

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

方法名与参数都一样

Overloading 重载

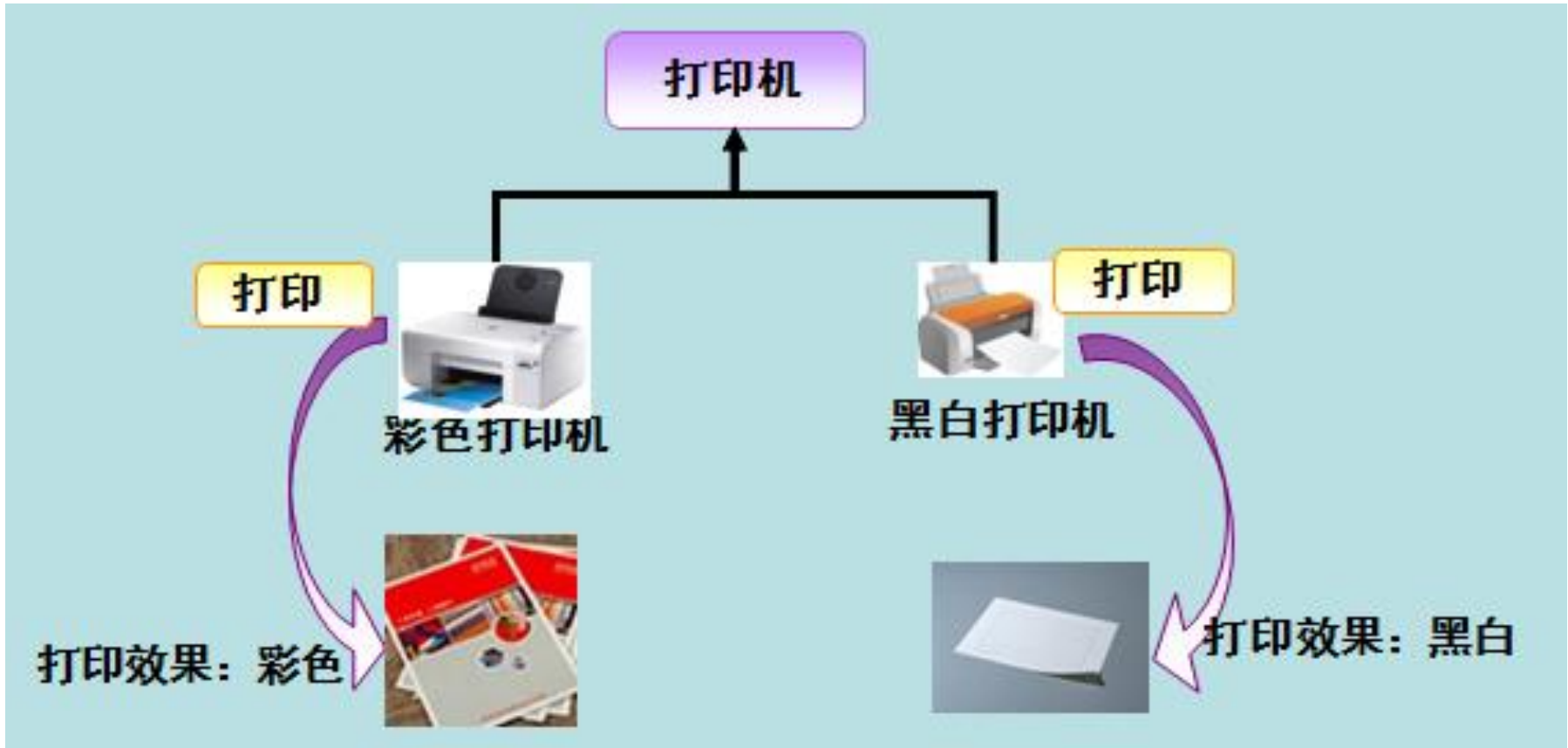
```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

方法名相同，参数不同

Java 多态

- 多态是同一个行为具有多个不同表现形式或形态的能力。
- 多态就是同一个接口，使用不同的实例而执行不同操作，如图所示：


打印机
打印
彩色打印机
打印
黑白打印机
打印效果：彩色
打印效果：黑白

多态的实现方式

- 方式一：重写
- 方式二：接口
- 方式三：抽象类和抽象方法

多态的优点

- 1. 消除类型之间的耦合关系
- 2. 可替换性
- 3. 可扩充性
- 4. 接口性
- 5. 灵活性
- 6. 简化性

源文件声明规则

- 一个源文件中只能有一个public类
- 一个源文件可以有多个非public类
- 源文件的名称应该和public类的类名保持一致。例如：源文件中public类的类名是Employee，那么源文件应该命名为Employee.java

- 如果一个类定义在某个包中，那么package语句应该在源文件的首行。

源文件声明规则

- 如果源文件包含import语句，那么应该放在package语句和类定义之间。如果没有package语句，那么import语句应该在源文件中最前面。

- import语句和package语句对源文件中定义的所有类都有效。在同一源文件中，不能给不同的类不同的包声明

static 修饰符

- **静态变量：**
- static 关键字用来声明独立于对象的静态变量，无论一个类实例化多少对象，它的静态变量只有一份拷贝。 静态变量也被称为类变量。局部变量不能被声明为 static 变量。
- **静态方法：**
- static 关键字用来声明独立于对象的静态方法。静态方法不能使用类的非静态变量。静态方法从参数列表得到数据，然后计算这些数据。

Static 详解

- 方便在没有创建对象的情况下来进行调用。
  - 也就是说：被static关键字修饰的不需要创建对象去调用，直接根据类名就可以去访问

访问类中的实例变量和方法

```
/* 实例化对象 */
Object referenceVariable = new Constructor();
/* 访问类中的变量 */
referenceVariable.variableName;
/* 访问类中的方法 */
referenceVariable.methodName();
```

static关键字修饰方法

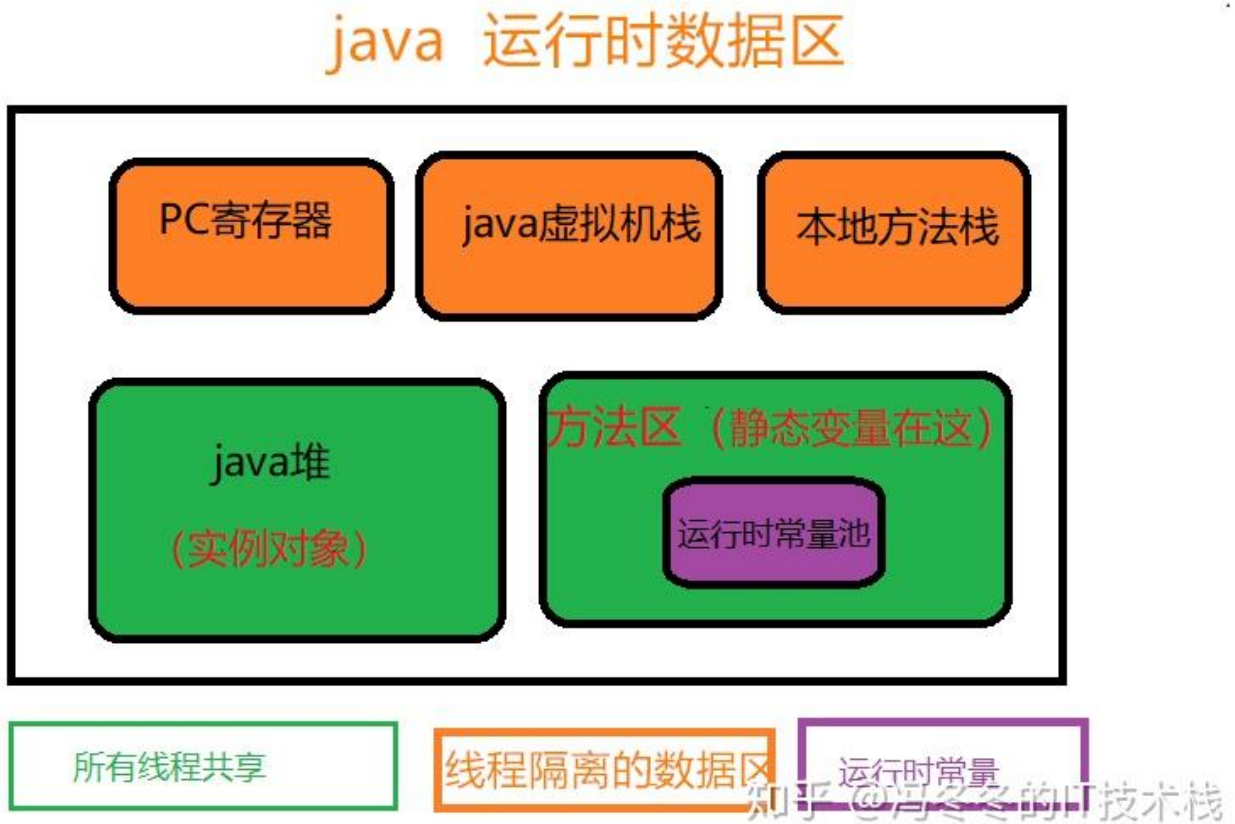
```
public class StaticMethod {
    public static void test() {
        System.out.println("============= 静态方法=============");
    };
    public static void main(String[] args) {
        //方式一：直接通过类名
    StaticMethod.test();
        //方式二：
    StaticMethod fdd=new StaticMethod();
        fdd.test();
    }
}
```

static关键字修饰变量

- 被static修饰的成员变量叫做静态变量，也叫做类变量，说明这个变量是属于这个类的，而不是属于是对象，没有被static修饰的成员变量叫做实例变量，说明这个变量是属于某个具体的对象的。

```java
public class StaticVar {
    private static String name="java的架构师技术栈";
  public static void main(String[] args) {
        //直接通过类名
    StaticVar.name;
    }
}
```

深入分析static关键字



Static 详解

- **堆区:** 1、存储的全部是对象，每个对象都包含一个与之对应的class的信息。(class的目的是得到操作指令) 2、jvm只有一个堆区(heap)被所有线程共享，堆中不存放基本类型和对象引用，只存放对象本身
- **栈区:** 1.每个线程包含一个栈区，栈中只保存基础数据类型的对象和自定义对象的引用(不是对象)，对象都存放在堆区中 2、每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问。 3、栈分为3个部分：基本类型变量区、执行环境上下文、操作指令区(存放操作指令)。

Static 详解

- **方法区:** 1、又叫静态区，跟堆一样，被所有的线程共享。方法区包含所有的class和static变量。

- 方法区中包含的都是在整个程序中永远唯一的元素，如class，static变量。

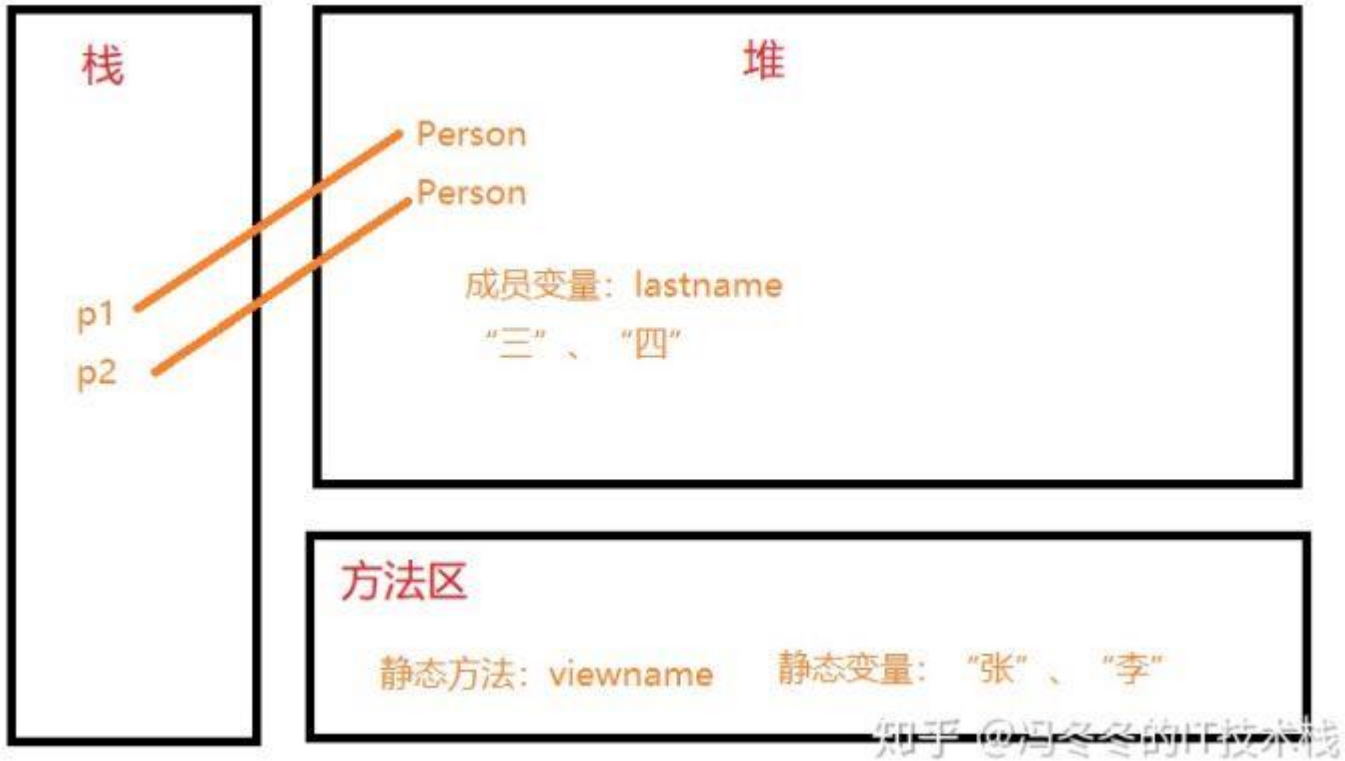
```java
public class Person {
    //静态变量
  static String lastName;
    String firstName;
    public void showName(){
        System.out.println(lastName+firstName);
    }
    //静态方法
  public static void viewName(){
        System.out.println(lastName);
    }

    public static void main(String[] args) {
        Person p =new Person();
        Person.lastName = "张";
        p.firstName="三";
        p.showName();

        Person p2 =new Person();
        Person.lastName="李";
        p2.firstName="四";
        p2.showName();
//                          p.showName();
    }
```

- p.showName()的输出是？
- A 张三
- B 李四
- C 李三
- D 张四

内存


栈
堆
Person
Person
成员变量：lastname
“三”、“四”
p1
p2
方法区
静态方法：viewname
静态变量：“张”、“李”

Static 详解

- 从上面可以看到，静态方法在调用的时候，是从方法区调用的，但是堆内存不一样，堆内存中的成员变量lastname是随着对象的产生而产生。随着对象的消失而消失。静态变量是所有线程共享的，所以不会消失。这也就能解释上面static关键字的真正原因。

static关键字小结

- 1、static是一个修饰符，用于修饰成员。（成员变量，成员函数）static修饰的成员变量 称之为静态变量或类变量。
- 2、static修饰的成员被所有的对象共享。
- 3、static优先于对象存在，因为static的成员随着类的加载就已经存在。
- 4、static修饰的成员多了一种调用方式，可以直接被类名所调用，（类名.静态成员）。
- 5、static修饰的数据是共享数据，对象中的存储的是特有的数据

成员变量和静态变量的区别

- 1. 生命周期的不同:

- 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。

成员变量和静态变量的区别

- 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，也可以用类名调用。（推荐用类名调用）

成员变量和静态变量的区别

- 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，也可以用类名调用。（推荐用类名调用）

成员变量和静态变量的区别

- 3、别名不同:
- 成员变量也称为实例变量。
- 静态变量称为类变量。

成员变量和静态变量的区别

- 4、数据存储位置不同:
- 成员变量数据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。

静态方法只能访问静态成员

- 当类的字节码被加载到内存的时候，静态变量和静态方法就已经被分配了相应的内存空间，但是实例方法和实例成员变量还没有被分配内存空间（只有在创建了新的对象的时候才分配），所以不能去访问一个可能不存在的东西。

- 非静态既可以访问静态，又可以访问非静态

主函数是静态的

- public static void main(String... args)
- 1、正因为 main 方法是静态的，JVM 调用这个方法就不需要创建任何包含这个 main 方法的实例。
- 2、因为 C 和 C++ 同样有类似的 main 方法作为程序执行的入口。
- 3、如果 main 方法不声明为静态的，JVM 就必须创建 main 类的实例，因为构造器可以被重载，JVM 就没法确定调用哪个 main 方法。

总结

一个 Java 程序可以认为是一系列对象的集合，而这些对象通过调用彼此的方法来协同工作。

- **实例变量**：每个对象都有独特的实例变量，对象的状态由这些实例变量的值决定。

- **方法**：方法就是行为，一个类可以有很多方法。逻辑运算、数据修改以及所有动作都是在方法中完成的。

答疑