Java

知

识

点

李浩洋专用

2019-12-06

学习计划：

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 技术点 | 截至 | 完成情况 |
| 1 | Jojo的SpringCloud微服务安全，学完后用大白话总结在本word | 12月15号 | 25% |
| 2 |  |  |  |
| 3 |  |  |  |
| 4 |  |  |  |
| 5 |  |  |  |
| 6 |  |  |  |
| 7 |  |  |  |
| 8 |  |  |  |
| 9 |  |  |  |
| 10 |  |  |  |

**目录**

# 一 javaSE

## 1.1面向对象

面向对象与内存解析

方法重载与内存解析

this关键字

super关键字

继承

重写

继承中的构造方法

Object的toString方法

Object的equals方法

hashCode解释

static关键字

对象转型

多态

抽象类

final关键字

Interface

## 自问自答：

谈谈面向对象吧？封装、继承、多态

一：抽象封装

答：嗯...（思考几秒）那我就举个例子吧：就用面向对象思维封装“老张开车去东北”吧。

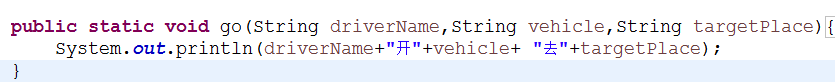
最原始的，就用一mian方法，打印：

|  |
| --- |
| System.***out***.println("老张开车去东北！"); |

下一步封装，比如，把人物“老张”，交通工具“车”，目的地“东北”，抽取出来，提取成变量：

|  |
| --- |
| String driverName = "老张";  String vehicle = "车";  String targetPlace = "东北";  System.***out***.println(driverName+"开"+vehicle+ "去"+targetPlace); |

进一步封装，可以封装成一个方法：



这还不够面向对象，面向对象拿到一个问题，首先要考虑问题域里面有哪些对象。考虑类：一般问题域里面的名词，就是对象，或者对象的属性（不可脱离具体的应用环境），动词是方法。比如这个问题里面：

隐藏封装（降低耦合度）：属性定义为private 私有的，对外提供get、set方法，也是一种隐藏和封装，比如不想让别人修改某个属性，把set方法去掉就可以。比如可以在get方法里添加判断，看你是否有权利get该属性。

司机Driver：

属性：name

方法：drive（开车）

车Car ：

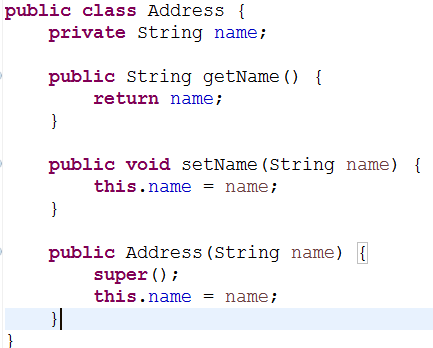
方法: go()

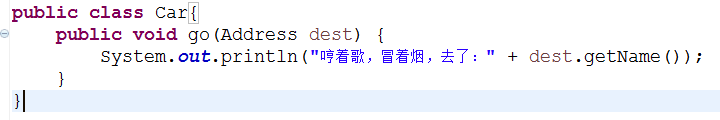
地址Address

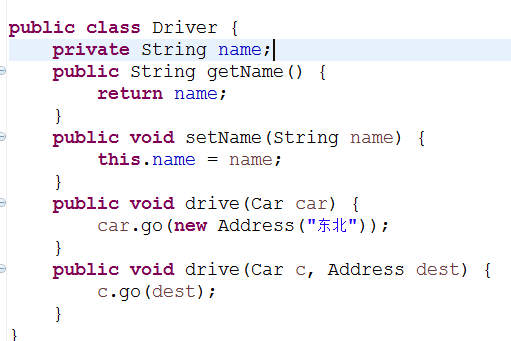
属性：name

要考虑 类和类之间的关系：司机和车之间是有关系的，司机可以开车，可以把车定义为司机的一个属性（这辆车就属于这个司机），也可以把车传个司机的开车方法drive(Car c)（司机可以开任何车）；

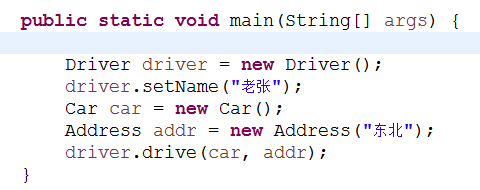
代码：



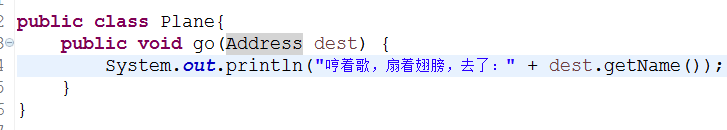




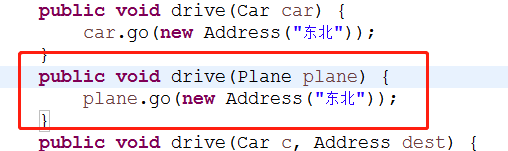
测试类：



现在如果不想开着车去，想开飞机，怎么办呢？再写个类：飞机Plane



司机再加个drive的重载方法：

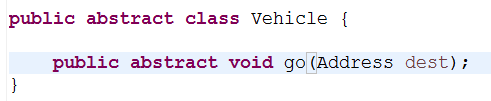


这样就可以了。这样的问题是可扩展性不好，随着交通工具的增多，司机类的drive方法需要继续重载，可不可以让司机只有一个drive方法，就能drive所有的交通工具呢？

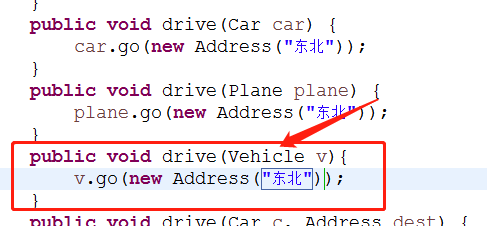
这就要用到继承了。

二：继承

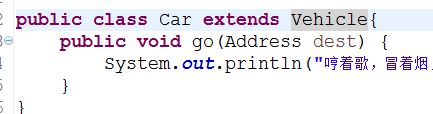
可以。写个交通工具类Vehicle，这个类是抽象的：

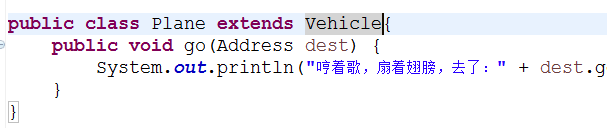


然后司机类里面，drive方法传入Vehicle交通工具，

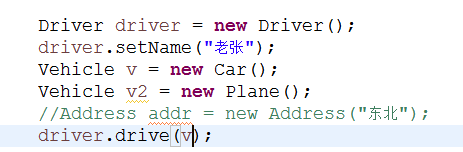


然后让Car和Plane都继承Vehicle，重写go方法：





测试类：



这样在用户使用的时候，传入是Car，就调用Car的go方法，传入的是Plane，就调用Plane的go方法，这就叫**多态**。

发生多态的三个条件：要有继承，要有重写，父类引用指向子类对象

多态的耦合性比较高，父类的修改会影响子类，需要慎重使用

多态的好处：**可扩展性高了，这就是面向对象的精华**

如果再添加其他交通工具，只需要添加类，然后继承Vehicle类就好了。

接口和抽象类的区别：

如果你脑子里有一个具体的事物，比如交通工具，就设计为抽象类。

如果你脑子里没有具体的事物，只有一类事物，或者几类事物的特征，比如说，会飞的、会跑的，就定义为接口，runnable、Moveable，比如上例中的Vehicle，就可以设计成Moveable，

然后里面有个方法，run(), 然后Car、Plane都实现这个接口即可。

## 1.2 异常

## 1.3数组

数组的创建和使用

## 1.4常用类

String

StringBuffer

Integer缓冲

枚举类

## 1.5容器

容器API\_Collection

迭代器Iterator

List

Set

Compareable

Map

自动装箱&拆箱

泛型

## 1.6 IO

### 1.6.1概念

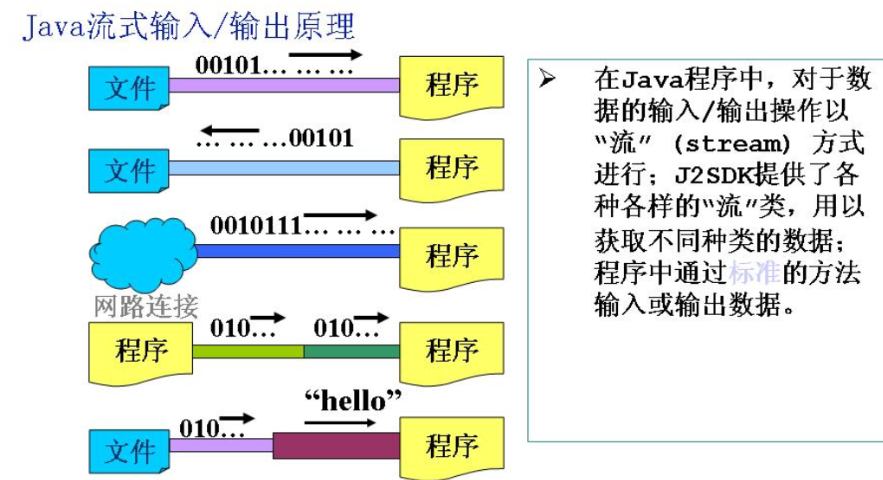
流 水流的流 流氓的流，，流 英文叫做stream，溪流

### 1.6.2 分类



### 1.6.3 Java流输入输出原理

流是用来读写数据的。



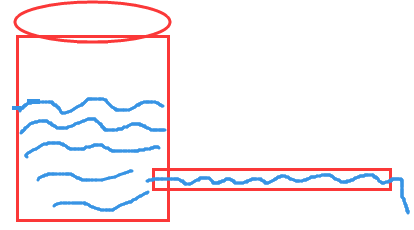
流就像水流一样，File类封装的是文件的名字，它是内存里头的一个对象，OK，但是真正的文件是在硬盘上的一块空间，在这文件里头，存着各种各样的数据，那么我们想读这数据怎么办呢？是这样来读，是通过一个流的方式来读。

**从文件读数据：**

如下图，比方说这是我们文件，咱们要想从程序读数据，大家都知道，对于计算机来说，你读任何的数据都是什么样的形式，是不是都是010101 这些形式，对吧 。



怎么样吧这些文件里边的内容个读出来，你就把这个文件想象成一个小桶。文件就是个桶，你怎么从桶里边取水呢？或者说怎么从文件里边取数据呢，常见的取法是这样的，我们弄一根管道怼到桶上，然后把这根管道，哐唧，一怼，怼到桶上边，这边打开水龙头，就开始出水，这就叫流，什么叫流，这根管道就叫流，水流哗啦哗啦哗啦流出来了，这就叫流，java里头的输入输出，跟这个水流一模一样，当你要从文件取数据的时候，一根管道咣唧怼上去，然后就顺着这根管道，你开始往外读数据。

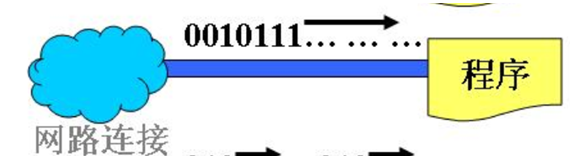


**往文件写数据：**

反过来，咣唧一根管道怼上去，从管道这一头开始咵唧咵唧往里头灌水。

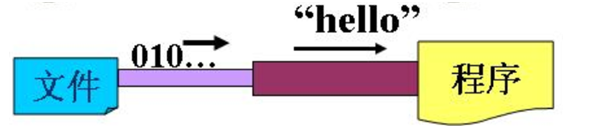


**从网络上读数据：**



比如这根管道咵唧从我机器上怼到你机器上了，我说一句话，你那边就收着了。你说一句话我这边就收着了

**处理流**，包装在其他流之上的流：



 有的时候一根管道不够使，比方说这根管道读过来的水，里有杂质，外边再包一层管道， 比方说你读过来的数据，最原始的数据肯定是010101，一个字节一个字节的往外读，当你这样读的时候你觉得这样的方法不合适，没关系。外边再包一层强大一点的管道，这个管道可以吧010101转换成字符串，哐当读出来，这个管道就包在了原始管道的外边， 这就是流式的输入输出，这些管道就是java的流。

### 1.6.4 流的分类

根据不同的角度分成不同的类型

输入流or输出流的划分：都是站在程序打的角度来说的，相对于程序来说的。

#### 字节流or字符流：

字节流：最原始的一个流， 读出来的数据就是010101，只不过是按照字节来读的，一个字节是8位。

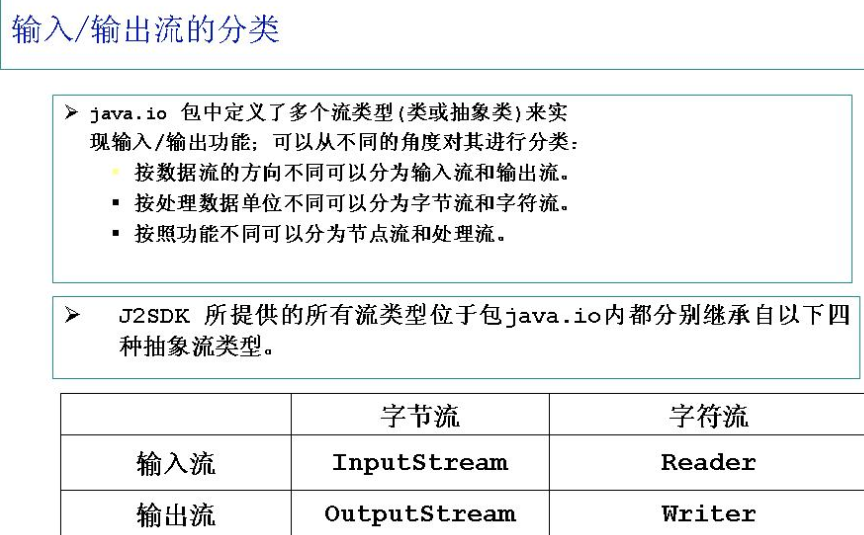
字符流：一个字符一个字符往外读，字符是2个字节。Java里边是Unicode字符串一个字符是2个字节。 Utf16  ok

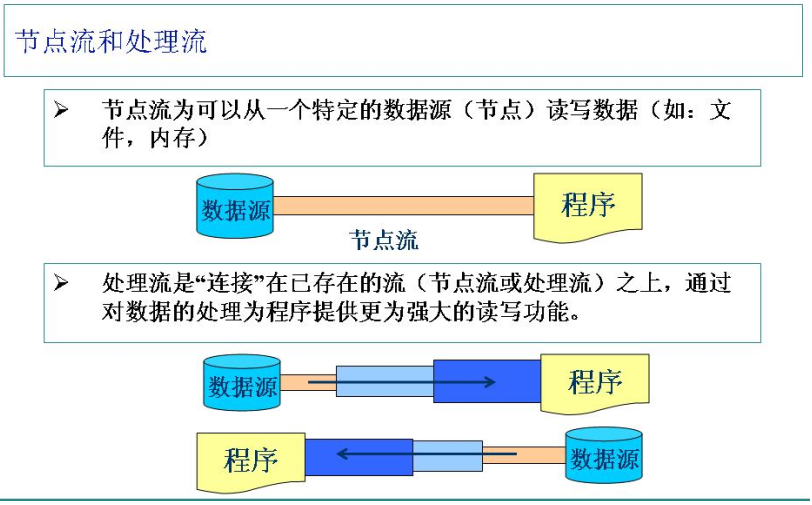
#### 节点流or处理流：

节点流：这根管道直接怼到数据源上的流。

处理流：套在其他管道之上的流。

凡是以Stream结尾的都是字节流，最原始的读取数据的方式，一个字节一个字节的读。





FileInputStream&FileOutputStream

FIleReader &FileWriter

缓冲流

转换流

DataIO&ByteArrayIO

打印流

对象流

对象序列化&反序列化

## 1.8 网络编程

网络基础

IP

TCP

UDP

TCP\_SOCKET

DUP

专题：线程池等-------@see多线程与高并发专题

# 二 Spring

# 三 SpringMVC

# 四Mybatis

# 五 互联网架构基础必备

## 多线程与高并发

## java内存模型JMM

## JVM调优案例实战

## 经常被问的操作系统问题

## 网络原理解读

## 设计模式&设计原则

## ~~UML~~

# 六 分布式必备基础技能

## 互联网架构演变

## 高负载高并发架构设计指导思想

## LVS

## Keepalived

## 流量介入层高性能服务

## Zookeeper

## 消息中间件

## ELK

## 分布式事务

## 单点登录

## 分布式任务调度

## 缓存

# 七 Redis & Netty

# 八 微服务

## RPC构建分布式

## SpringBoot

## SpringCloud

## Docker

## ~~Kebernetes（暂可不学）~~

## ~~Service Mesh（暂可不学）~~

## ~~微服务的学与思（面试就要喷这些玩意！）~~

# 九 分布式存储

## Mysql优化

## FastDFS

## OpenResty

# 十 工具

## Maven

## Git

## Jenkins

# 十一 源码解析

## JDK源码解析

## Spring源码解析

## MyBatis源码解析

## Dubbo源码解析

## SpringMVC源码解析（阅读SpringMVC源码解析书籍）

## ~~Netty源码解析~~

。

# JavaSE

# 第五章 多线程与高并发

## 5.1，线程基础概念

### 5.1.1基本概念

#### **线程**

一个程序里边不同的执行路径

#### 线程和进程的区别

一个QQ.exe是一个程序是静态的概念，一运行，就有一个进程，是一个动态的概念，线程是一个进程内不同的执行路径

每个进程都有独立的代码和数据空间（进程上下文），进程间切换会有较大的开销。

线程可以看成是轻量级的进程，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器，线程切换开销小。

多进程：在操作系统中能同时运行多个任务（程序）

多线程：在同一应用程序中有多个顺序流同时执行

我们的windows同时运行着好多个线程，所以说机器是支持多线程的。当然也支持多进程。windows、linux、unix都是多进程、多线程的，dos是只支持单进程的，同一个时间点只能执行一个进程执行。

|  |
| --- |
| 进程：进程是一个静态的概念，机器上的一个class文件，一个exe文件。  程序的执行过程，要把程序的代码放到内存里，放到代码区里，一个进程准备开始，进程已经产生了，但还没有开始执行，这叫进程。是一个静态的概念。平时说的进程的执行说的是进程里面主线程开始执行了。在机器里面运行的实际上都是线程。 |
|  |
| CPU计算速度快，把自己的时间分成一个个时间片，这个时间片执行你一会，下一个时间片执行它一会，轮着来。虽然有几十个线程，轮着执行由于速度很快，在我们看来就像是多个线程同时执行的。但实际上，一个时间点上，CUP只有一个线程在运行。（如果机器是多核，有多个CPU，才是真正意义上的多线程） |

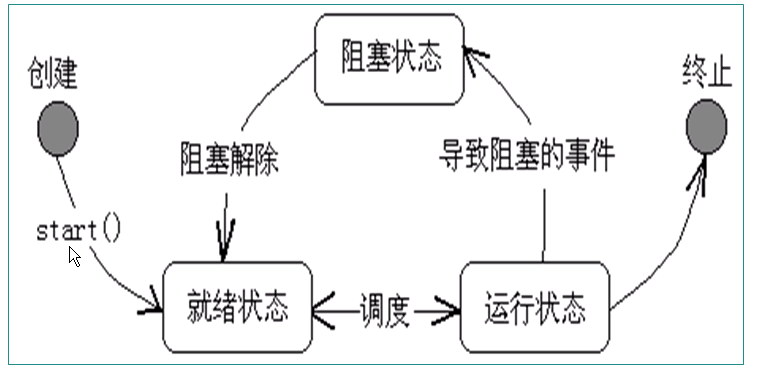
### 5.1.2启动线程

1继承thread，

2实现runnable，

#3吹毛求疵：lamba，是实现runnable的变形；通过线程池，也是1、2中的一种

### 5.1.3 线程的状态（初级版）



|  |
| --- |
| 一个线程类，new出来只是内存里的一个对象。调用start方法，不是立即执行，而是就绪状态，排队等候，因为CPU可能正在执行其他的线程。CPU高兴了，把对象调到CPU里才得以执行。执行的过程中，可能你的时间片到头了，还回到就绪状态去排队去，等候下次轮到你，一直到执行完了，就终止了。 |

### 5.1.4常用方法



#### 5.4.1 Thread.sleep()

线程睡眠指定的毫秒数，睡完后进入就绪状态

#### 5.4.2 Join

合并某个线程，等待某个线程执行结束，再恢复当前线程的运行。

在t1线程调用t2.join 等待t2线程执行结束。

问： 启动三个线程怎么让他们按顺序执行？

在t1里调用t2.join，在t2里调t3.join



例子程序：

|  |
| --- |
| **public class** Test {   **public static void** main(String[] args) {  MyThread2 t1 = **new** MyThread2(**"abcde"**);  t1.start();  **try** {  */\*\*  \* 把t1线程合并到主线程，即等待t1线程执行完，再执行主线程  \* 相当于方法调用  \*/* t1.join();  } **catch** (InterruptedException e) { }  *//等t1执行完了，才有机会执行这* **for**(**int** i=1;i<=10;i++){  System.***out***.println(**"i am main thread"**);  }  } }  **class** MyThread2 **extends** Thread {  MyThread2(String s){*//给线程起名字* **super**(s);  }   @Override  **public void** run(){  **for**(**int** i =1;i<=10;i++){  System.***out***.println(**"i am "**+getName());  **try** {  *sleep*(1000);  } **catch** (InterruptedException e) {  **return**;  }  }  } } |

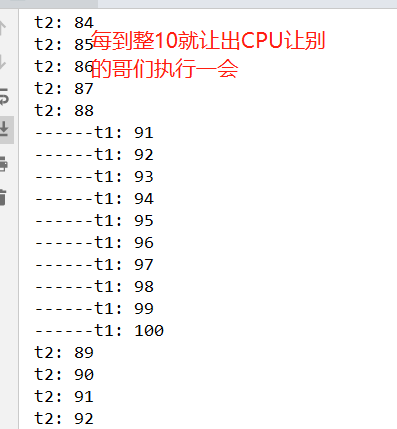
执行结果：



#### 5.4.3 Yield()

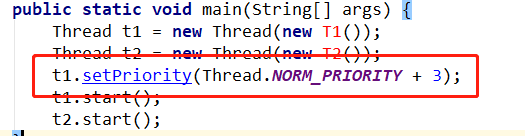
高风亮节，让出一下CPU，进到等待队列，给其他线程执行的机会，然后再回来。

|  |
| --- |
| **public class** Test {   **public static void** main(String[] args) {  MyThread3 t1 = **new** MyThread3(**"------t1"**);  MyThread3 t2 = **new** MyThread3(**"t2"**);  t1.start();  t2.start();  } } **class** MyThread3 **extends** Thread {  MyThread3(String s){**super**(s);}  **public void** run(){  **for**(**int** i =1;i<=100;i++){  System.***out***.println(getName()+**": "**+i);  **if**(i%10==0){*//能被10整除，让出时间片，不明显  yield*();  }  }  } } |



#### 5.4.4 Priority

线程优先级



### 5.1.5 线程同步

### 5.1.6 Synchronized 锁升级

### 5.1.7 Synchronized 同步方法与非同步方法

### 5.1.8 Synchronized 锁重入

### 5.1.9 异常与锁

### 5.1.10 volatile 关键字

### 5.1.11 AtomicXXX 原子类

### 5.1.12 wait notify （面试高频）

线程同步

生产者消费者

## 5.2 JUC 同步工具

### 5.2.1 CAS自旋原理

### 5.2.2 ReentrantLock 可重入锁

### 5.2.3 Condition 条件等待与通知

### 5.2.4 LATCH门闩

# 十二 设计模式

## 12.1 单例

## 12.2 工厂

### 12.1.1 简单工厂

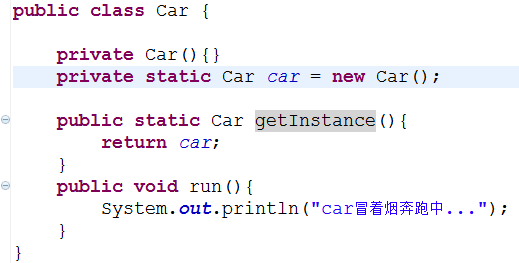
#### 自问自答

**问：设计模式了解吗？来介绍一下工厂吧！**

答：嗯，，，（思考），举个例子吧，“老张开车去东北”比如要说----交通工具，

**问题一：只给司机一辆车？**

单例：私有化构造方法，getInstance

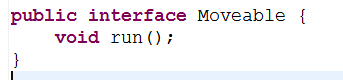


多例：



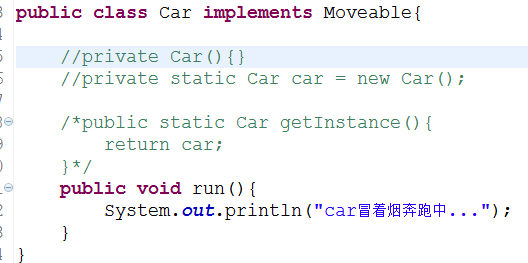
**问题二：可以任意定制交通工具的类型 && 和 生产过程**

抽象Moveable接口

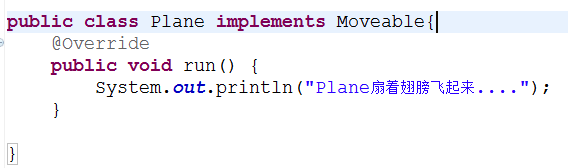


所有交通工具都实现这个接口：（多态）

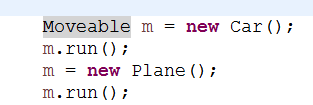
汽车实现：



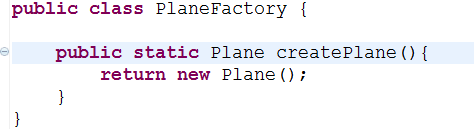
飞机实现：



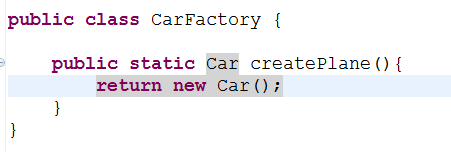
调用：父类引用指向子类对象。多态



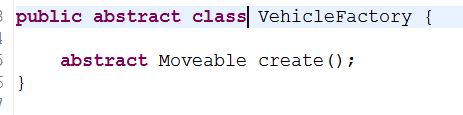
飞机工厂：来生产飞机



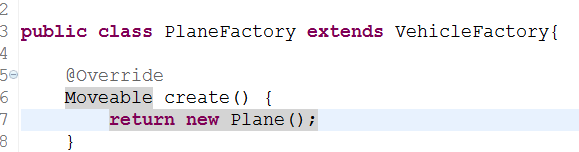
如果要生产汽车，就要再来个汽车工厂：



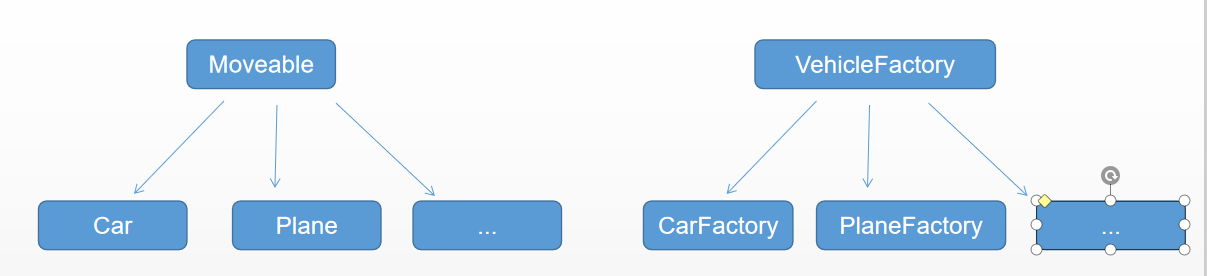
定义交通工具抽象工厂：



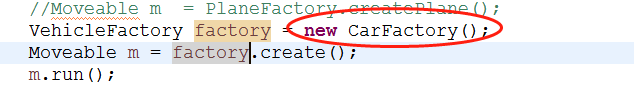
飞机工厂和汽车工厂继承交通工具工厂：



此时是这样的：



测试类只需要换工厂的实现：



这种模式，被称为简单工厂

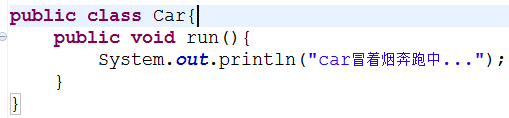
### 12.1.2 抽象工厂

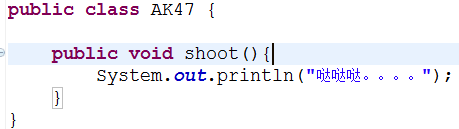
#### 自问自答

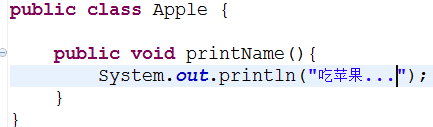
老张开车去东北

系列产品（车、武器、食品补给.....）一次性替换一系列产品

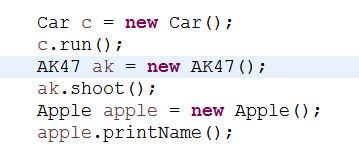
可以设计汽车类Car、武器类AK47、食品类Apple







测试类：

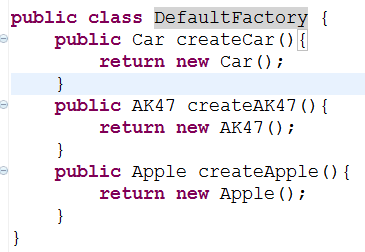


car冒着烟奔跑中...

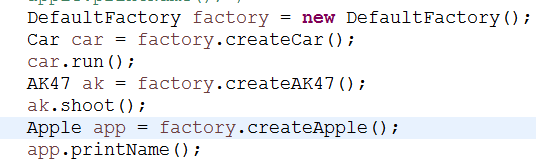
哒哒哒。。。。

吃苹果...

此时可以来一个工厂，来生产这一系列产品：



客户类：



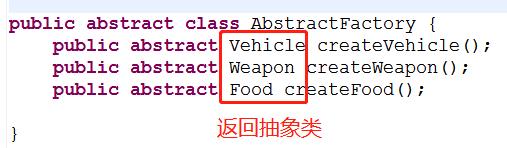
car冒着烟奔跑中...

哒哒哒。。。。

吃苹果...

这时，如果要把这一系列（汽车、AK47、苹果）都给换了，就需要从新定义一个工厂，返回其他的比如（飞机、极光武器、瓜子）一系列物资。

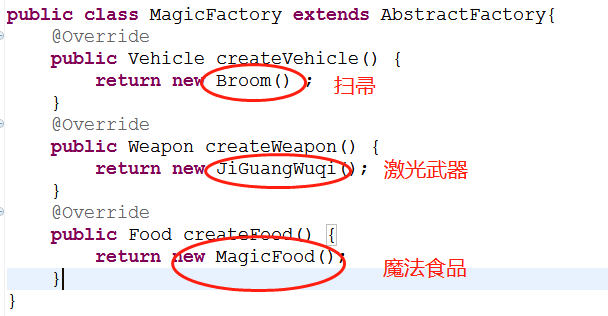
此时可以定义一个抽象工厂：



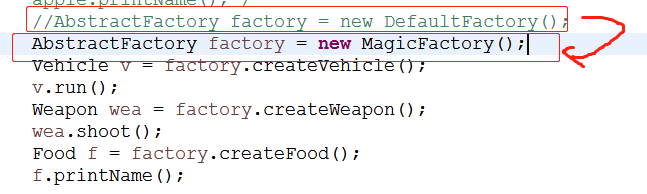
然后Car、Ak47、Apple都继承这些抽象类。

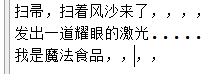
下面再定义其他工厂，都继承这个抽象工厂，比如说：魔法工厂：

MagicFactory

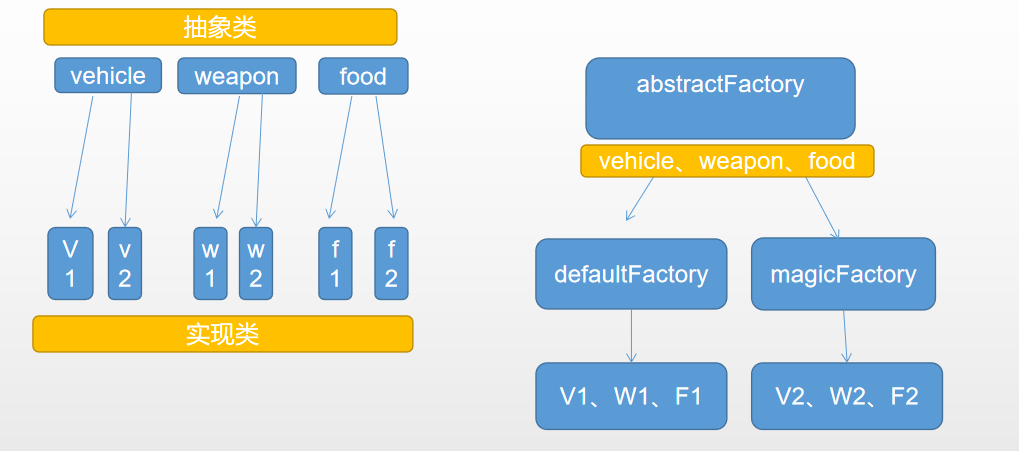


此时客户类，只需要吧具体工厂的实现换了，就能一下换了所有物品。其他代码不用动





现在的好处是，你对一系列产品不满意，就可以自己定义自己的工厂继承抽象工厂，定义自己的产品，继承抽象产品。（换肤）



抽象工厂：生成一系列的产品，如果想一次性换掉这一系列的产品。或者说是做扩展。就用抽象工厂

普通工厂和抽象工厂的区别：

**普通工厂**：方便从产品维度进行扩展。添加产品、添加产品工厂。但是随着产品增多，想增加一系列产品，会导致【工厂泛滥】

**抽象工厂：**方便添加一系列产品。但是如果想添加一个产品品种，需要先在抽象工厂添加该产品的创建方法，添加该产品的抽象类，所有已经实现的工厂，需要都实现新添加的方法。麻烦