Lambda表达式：list.forEach(element->{xxxx});

多参数：（student1, student2）->{ } 有多个参数时，必须使用小括号包裹

无参数：( ) ->{ } 没有参数的时候，必须使用小括号

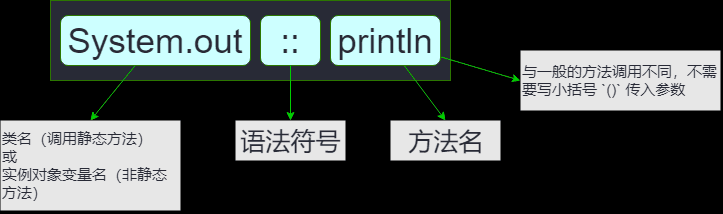
排序 Collections.sort(ar, (a, b) ->{ //a表示后一个元素，b表达前一个元素

Return b.getAge( ) – a.getAge( ); //b-a表示降序 a-b表示升序

})

双冒号（::）操作符 names.forEach(System.out::println);

System.out::println等同于n->{System.out.println(n);}



调用静态方法： LambdaTest::print 代替 f->LambdaTest.print( f )

非静态方法： LambdaTest lt = new LambdaTest( ); fruits.forEach(lt::print);

简写为fruits.forEach(new LambdsTest( )::print);

Stream<xxxx> stream = Stream.of( xxxx数组 );

Stream<xxxx> stream = xxxxlist.stream( );

**数据流过滤：.filter( xxxx -> xxxxx )**

Pupils.stream( ).filter(pupil ->pupil.getAverageScore() >= 80&& pupil.getViolationCount() <1)

.forEach(pupil->{ System.out.println(pupil.getName( ));});

**数据流映射：.map( xxx ->{ return xxxx; })**

Numbers.stream().map(num->{return num\*num;}).forEach(System.out::println);

**数据流排序：sorted((x1,x2)->{return xxxx;}) //x1指后一个元素，x2指前一个元素**

Students.stream().sorted((stu1,stu2)->{return stu1.getRoolNo()-stu2.getRollNo();})

数据流摘取：**.limit(x) //返回流的前x个元素**

**流合并：.reduce((a,b)->a+b)**

Int sum = numbers.stream().reduce((a,b)->a+b).get();

a:第一次运行的第一个元素/上次运行的缓存结果

b:运行的第二个元素

**reduce返回对象**

Student result=students.stream()

.reduce(new Student(“”,0)(a,b)->{

a.setMidtermScore(a.getMidtermScore()+b.getMidtermScore());

return a;});

此时new对象作为缓存角色的对象，a表示第一个参数对象，b表示第二个参数对象

且new对象作为返回对象，无需调用.get( )方法。

**forEach（）方法和reduce（）方法都是流的终点。**

**数据流收集：.collect( ) ——流的终点**

.collect( Collectors.toList( ) ); Collectors.toList( )是一个静态方法，作为参数告诉collect（）方法存入一个List集合。

**Stream（）——串行 parallelStream（）——并行**

将代码中的.stream( )改成parallelStream（）

**单例模式**

一个类只允许创建一个实例对象

将构造函数设为私有，除了自己任何类都不能实例化该类对象

Public class ClassMaster{

private xxxx属性;

private static ClassMaster instance = new ClassMaster( );//唯一实例，注意必须使用static

private ClassMaster( ){ }

public static ClassMatser getInstance( ){ return instance;}//外部类访问途径

}

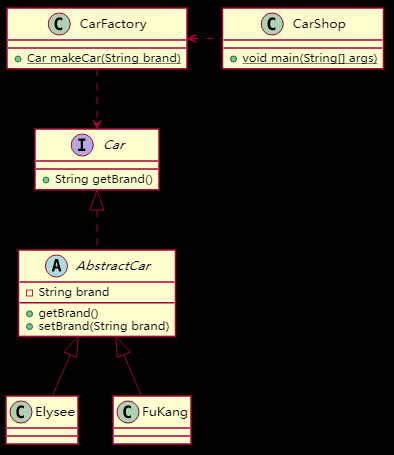
**简单工厂模式**

**（一种产品多个类别，类似于工厂）**

1.从具体的产品类抽象出接口，连接到产品接口类

2.把生产实例对象的过程，收拢到工厂类中实现

3.应用抽象类减少代码重复：抽象类存放公共属性和方法，每个实现类存放各种特有的属性和方法



**Extends 和 implements**

Extends是继承某个类，继承之后可以使用父类的方法，也可以重写父类的方法

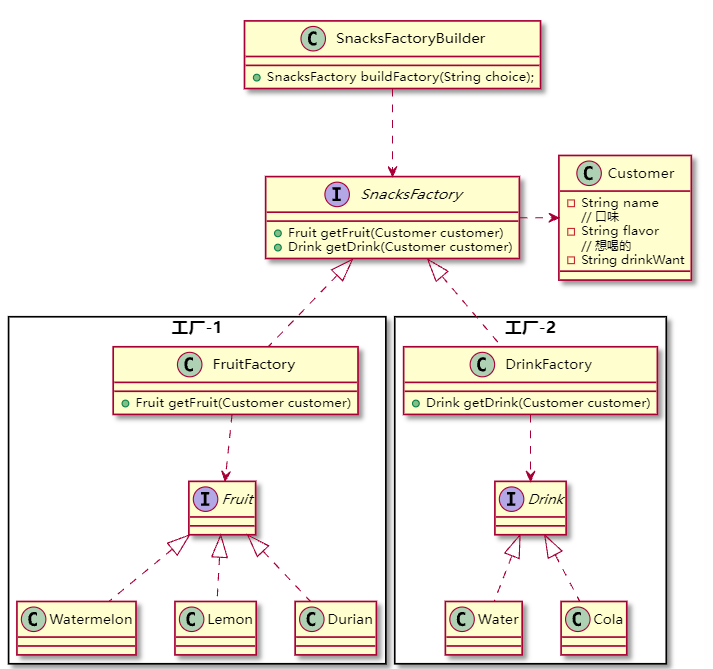
（只要那个类不是声明为final或者那个类定义为abstract的就能继承）

Implements是实现多个接口，接口的方法一般为空，必须重写才能使用

**抽象工厂模式**

**（一个系列，多种产品，类似于超市）**

简单工厂的主要作用是把多个产品抽象，使用一个工厂统一创建；那么抽象工厂的主要作用是把多个工厂也进一步抽象。



1.工厂接口：包含所有工厂的方法

2.工厂的工厂：简单工厂模式

**Spring结合工厂**

在工厂类上添加 @Component

在任何需要工厂的类内添加@Autowired自动注入

**观察者模式**

被观察类：public class WeatherData extends Observable{

Xxxx

发生变化时：super.setChanged( );

super.notifyObservers(“xxxx”);（括号里可以是信息也可以是**任意对象**）

}

观察者类：public class WeatherObserver implements Observer{

Xxxx

Public void update(Observable o,object arg){

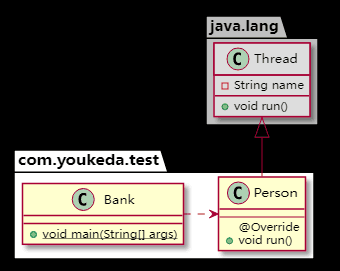
第一个参数是被观察者对象，即被观察类

第二个参数是被观察类中notifyObservers内发送的信息

}

}

**继承Thread类（只允许继承一个类）**



Person类extends（继承）Thread类，并重写run（）方法【public void run()】，

Thread父类中有name属性（private），只能通过setName（）和getName（）调用

Person thread1 = new Person(); System.out.println(getName()+”开始取钱”);

Thread1.setName(“张三”);

**实现Runnable接口**

Public class Person implements Runnable {

Xxxx

Public void run() { xxxx }

}

Runnable接口中只有一个带实现的run（）方法，需要自己补充属性，

Thread.sleep（）方法在Runnable中仍然可用

实现Runnable接口的线程类，需要包装在Thread类的实例中：Thread thread1 = new Thread（person1）； **thread1.start（）；**

**正在运行的线程的实例对象Thread.currentThread( )**

**在线程函数中添加synchronized解决余量错误（可见性）**

Public **synchronized** void sell( ){ xxxx }

Synchronized也叫线程**同步锁**，表示该方法同一时刻只能由一个线程执行。

（synchronized可能由于线程等待导致性能问题）

**原子性**：方法全部执行并且执行的过程不会被任何因素打断。

**可见性**：当多个变量访问同一个变量时，一个线程修改了这个变量的值，其他线程能立即看得到修改得值。

**AtomicInteger实现余量自增自减（原子性）**

Private AtomicInteger count = new AtomicInteger（30）；

Int newCount = count.decrementAndGet（）；

**CompletableFuture**

.supplyAsync()

.thenApply()

.thenAccept()