Selenium WebDriver阶段三 - (高级Jave设计模式在webdriver中的实践)

WebDriver
(阶段三)

课程目的：掌握日常自动化项目开发的必备java基础知识。
培训结果：熟悉java的继承，多态，接口等知识并运用到webdriver中。
课程相关脚本：practicejava
作者：Terry

QQ:314768474

个人微博：http://weibo.com/alwaysterry

- ■类的定义
- ■声明对象
- ■构造函数
- ■this关键字
- ■方法重载
- ■访问控制
- ■Static 和Final
- ■String和StringBuffer

类定义
声明对象

```
Class classname{
   type instance-variable1;
    …
   type instance-variableN;

    type methodnames(parameter-list1){
       body of method
   }
}
```

一个简单Java类

```
public class HelloJava {
    public static void main(String[] args) {
       System.out.println("Hello world!");
    }
}
```

```
class Box{
   double width;
   double height;
   double depth;
}
class BoxDemo2{
 public static void main(String[] args) {
   Box mybox = new Box();
   double vol;
   mybox.width  = 10;
   mybox.height  = 20;
   mybox.depth  = 9;
   vol = mybox.width* mybox.height*mybox.depth;
   System.out.println("Volume is " + vol);
  }
}
```

为什么对整数或字符的简单变量不使用new 运算符。答案是java简单类型不是作为对象实现的。出于效率的考虑它们是作为常规常量实现的。new 运算符是在运行期间为对象分配内存的，这样做的好处是你的程序在运行期间可以创建它所需要的内存，但是内存是有限的当内存不足就抛出异常。

构造函数

构造函数在对象创建时初始化，它与它的类名相同，它的语法与方法类似

注意构造函数没有返回值

```
class Box{
        double width;
        double height;
        double depth;
        Box(String a){    //定义的构造函数不带参数
            width = 10;
            height = 10;
            depth = 10;
        }
        double volume(){
            return width*height*depth;
        }
    }

    class BoxDemo2{
     public static void main(String[] args) {
        Box mybox = new Box();
        System.out.println("Volume is " + mybox . volume());
      }
    }


    class Box{
       double width;
       double height;
       double depth;
       Box(double w,doublle h,double d){//带自变量的构造函数
           width = w;
           height = h;
           depth = d;
       }
       double volume(){
           return width*height*depth;
       }
    }
    class BoxDemo2{
     public static void main(String[] args) {
       Box mybox = new Box(10,10,10);
       System.out.println("Volume is " + mybox . volume());
      }
    }
```

this关键字

this关键字是在当前类中将当前类作为一个对象来使用，这样可以调用其内在的方法和成员变量。
注意：作用范围是当前类。

```
Box(double width,doublle h,double d){
        this.width =  width ;
        this.height = h;
        this.depth = d;
    }
```

this的第二个用法是定义2重名的局部变量在java中是不合法的，所以this还有隐藏的实例变量

方法重载overload

同一个类中的2个或2个以上的方法可以有同一个名字，只要他们的参数声明（类型或数量）不同即可，在这种情况下该方法被称为重载。

下面是一个说明方法重载的简单例子：

```
// Demonstrate method overloading.
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for one integer parameter.
  void test(int a) {
    System.out.println("a: " + a);
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }

  // overload test for a double parameter
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}

class Overload {
```

```
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    double result;

    // call all versions of test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Result of ob.test(123.25): " + result);
  }
}
```

访问控制

public
默认(friendly)
private
protected

访问控制符	同类	同包子类	同包其它类	不同包子类	不同包其它类
public	√	√	√	√	√
protected	√	√	√	√	×
默认	√	√	√	×	×
private	√	×	×	×	×

Static讲解

该修饰符可以修饰成员变量，成员常量和成员方法。使用该关键字修饰的内容，在面向对象中 static修饰的内容是隶属于类，而不是直接隶属于对象的，所以 static修饰的成员变量一般称作类变量，而 static修饰的方法一般称作类方法

静态变量
静态方法
静态代码块

注意：static不能修饰普通类，但是对于内部类是可以修饰的。
1)static修饰的变量称作静态变量，在该类的所有对象之间共享值，方便访问变量。

```
class StaticDemo{ static int m; int n; char c;
public static  void main(String[] args){
    System.out.println(m);
}

}
public class TestStaticVar{
   public static void main(String[] args)
      {  StaticDemo sv1 = new StaticDemo();
         sv1.m = 2;
        StaticDemo sv2 = new StaticDemo();
```

```
        System.out.println(sv2.m);                    //结果值为2;
    }
}
```

我们也可以直接用ClassName.静态成员；
如上面的例子：
StaticDemo.m

2)静态方法
使用静态方法时，需要特别注意的是静态方法内部使用该类的非静态成员变量，否则将出现语法错误。

3)静态代码块
静态代码块在该类第一次被使用时执行一次，以后再也不执行,这里需要考虑的是它的执行顺序，静态代码块的执行顺序是系统启动时就立即执行。

```
public class StaticBlock{
   static{ System.out.println("静态代码块！")};
   }
}
```

final

final类不能被继承，没有子类，final 类中的方法都是默认是final;
final方法不能被子类覆盖，但是可以被继承；
final成员变量表示常量，只能被赋值一次，赋值后值不再改变；
final不能修饰构造方法；

String和 StringBuffer（线程安全的），StringBuilder

虽然都代表字符串，但是由于两个类内部实现的区别，所以一般把 String看成不可变字符串，而把 StringBuffer看成可变字符串，对于 String的每次改变(例如字符串连接等)都会生成一个新的字符串，比较浪费内存，而 StringBuffer每次都改变自身，不生成新的对象，比较节约内存。

String对象的初始化
由于 String对象特别常用，所以在对 String对象进行初始化时，Java提供了一种简化的特殊语法，格式如下：
String s = "abc"; s = "Java语言"; 其实按照面向对象的标准语法，其格式应该为：
String s = new String("abc"); s = new String("Java语言 ");
只是按照面向对象的标准语法，在内存使用上存在比较大的浪费。例如 String s = newString(“abc”);实际上创建了两个 String对象，一个是” abc”对象，存储在常量空间中，一个是使用 new关键字为对象 s申请的空间。

StringBuffer类
StringBuffer类和 String一样，也用来代表字符串，只是由于StringBuffer的内部实现方式和 String不同，所以 StringBuffer在进行字符串处理时，不生成新的对象，在内存使用上要优于 String类。

所以在实际使用时，如果经常需要对一个字符串进行修改，例如插入、删除等操作，使用 StringBuffer要更加适合一些。
StringBuffer对象的初始化

StringBuffer对象的初始化不像 String类的初始化一样，Java提供的有特殊的语法，而通常情况下一般使用构造方法进行初始化。
例如：
StringBuffers = new StringBuffer(); 这样初始化出的 StringBuffer对象是一个空的对象。
如果需要创建带有内容的 StringBuffer对象，则可以使用：
StringBuffer s = new StringBuffer("abc"); 这样初始化出的 StringBuffer对象的内容就是字符串"abc"。
需要注意的是，StringBuffer和 String属于不同的类型，也不能直接进行强制类型转换，下面的代码都是错误的：
StringBuffers ="abc"; //赋值类型不匹配
StringBuffers =(StringBuffer)"abc"; //不存在继承关系，无法进行强转 StringBuffer对象和 String对象之间的互转的代码如下：

```
String s = "abc";
StringBuffer sb1 = new StringBuffer("123");
StringBuffer sb2 = new StringBuffer(s); //String转换为 StringBuffer
String s1 = sb1.toString(); //StringBuffer转换为 String
```

如果我们的程序是在单线程下运行，或者是不必考虑到线程同步问题，我们应该优先使用StringBuilder类；当然，如果要保证线程安全，自然非StringBuffer莫属了。

Java 继承

```
//Monster.java
public class Monster
{
  public void move(){//移动功能}
}
//Boss.java
public class Boss extends Monster
{
  public void move(){//Boss类的移动规则}
}
//NormalMonster.java
public class NormalMonster extends Monster
{
  public void move(){// NormalMonster类的移动规则}
}
```

这样在 Monster的每个子类内部都重新书写了 move方法的功能，这种在子类内部重新父类中的方法的语法现象，称作方法覆盖(override)。

在使用子类的对象时，子类内部的方法将覆盖从父类继承过来的方法，也就是说子类的对象调用的是子类的功能方法，而不是父类的方法。在进行方法覆盖时，子类内部的方法和父类的方法声明相同，而且子类方法的限制不能比父类的方法严格。

继承需要注意的问题：

属性覆盖没有必要

方法覆盖可以重写对应的功能，在实际继承时在语法上也支持属性覆盖(在子类内部声明和父类属性名相同的属性)，但是在实际使用时修改属性的类型将导致类结构的混乱，所以在继承时不能使用属性覆盖。

子类构造方法的书写

该项是继承时书写子类最需要注意的问题。在子类的构造方法内部必须调用父类的构造方法，为了方便程序员进行开发，如果在子类内部不书写调用父类构造方法的代码时，则子类构造方法将自动调用父类的默认构造方法。而如果父类不存在默认构造方法时，则必须在子类内部使用 super关键字手动调用，关于 super关键字的使用将在后续进行详细的介绍。

说明：子类构造方法的参数列表和父类构造方法的参数列表不必完全相同。

子类的构造过程

在构造子类时由于需要父类的构造方法，所以实际构造子类的过程就显得比较复杂了。其实在实际执行时，子类的构造过程遵循：首先构造父类的结构，其次构造子类的结构，无论构造父类还是子类的结构，都是首先初始化属性，其次执行构造方法。则子类的构造过程具体如下：

如果类A是类B的父类，则类 B的对象构造的顺序如下：

(a)	类A的属性初始化
(b)	类A的构造方法
(c)	类B的属性
(d)	类B的构造方法

多态

多态性是面向对象技术中最灵活的特性，主要是增强项目的可扩展性，提高代码的可维护性。

多态性依赖继承特性，可以把多态理解为继承性的扩展或者深入。

在这里把多态性分为两方面来进行介绍，对象类型的多态和对象方法的多态。为了方便后续的讲解，首先给出一个继承结构的示例。

```
//文件名: SuperClass.java
public class SuperClass{
  public void test(){
    System.out.println("SuperClass");
  }
}
//文件名: SubbClass1.java
public class SubbClass1 extends SuperClass{
  public void test(){
    System.out.println("SubbClass1");
  }
}
//文件名: SubbClass2.java
public class SubbClass2 extends SuperClass{
  public void test(){
    System.out.println("SubbClass2");
  }
}
```

对象类型的多态是指声明对象的类型不是对象的真正类型，而对象的真正类型由创建对象时调用的构造方法进行决定。例外，按照继承性的说明，子类的对象也是父类类型的对象，可以进行直接赋值。

例如如下代码：

SuperClass sc = new SubbClass1();

这里声明了一个 SuperClass类型的对象 sc，然后使用 SuperClass的子类SubbClass1的构造方法进行创建，因为子类类型的对象也是父类类型的对象，所以创建出来的对象可以直接赋值给父类类型的对象 sc。除了对象的赋值以外，另外一个更重要的知识是 sc对象虽然使用 SuperClass声明的类型，但是内部存储的却是 SubbClass1类型的对象。这个可以Java语言的中 instanceof运算符进行判断。

instanceof是一个运算符，其作用是判断一个对象是否是某个类类型的对象，如果成立则表达式的值为true，否则为false。语法格式如下：

对象名 instanceof类名

需要注意的是：这里的类名必须和声明对象时的类之间存储继承关系，否则将出现语法错误。

```
/**
*测试对象类型的多态 */
public class TestObjectType{
  public static void main(String[] args){
    SuperClass sc = new SubbClass1();
    boolean b = sc instanceof SuperClass;
    boolean b1 = sc instanceof SubbClass1;
    System.out.println(b);
    System.out.println(b1);
  }
}
```

由程序运行结果可以看出，sc既是 SuperClass类型的对象，也是SubbClass1类型的对象，而 SubbClass1的类型被隐藏起来了，这就是对象的多态。其实 sc对象不仅仅在类型上是 SubbClass1类型的，其存储的内容也是SubbClass1的内容，具体参看后面介绍的对象方法的多态。

Java的多态机制是通过方法的重写和方法的重载实现的

1)override-继承了父类的同名函数；

2)overwrite-当前类同名方法；

抽象类和接口

在实际的项目中，整个项目的代码一般可以分为结构代码和逻辑的代码。就像建造房屋时，需要首先搭建整个房屋的结构，然后再细化房屋相关的其它的结构，也像制造汽车时，需要首先制作汽车的框架，然后才是安装配件以及美化等工作。程序项目的实现也遵循同样的道理。

在项目设计时，一个基本的原则就是——“设计和实现相分离”。也就是说结构代码和逻辑代码的分离，就像设计汽车时只需要关注汽车的相关参数，而不必过于关心如何实现这些要求的制作。程序设计时也是首先设计项目的结构，而不用过多的关系每个逻辑的代码如何进行实现。

前面介绍的流程控制知识，主要解决的是逻辑的代码的编写，而类和对象的知识，则主要解决结构代码的编写。那么还有一个主要的问题：如何设计结构代码呢？这就需要使用下面介绍的抽象类和接口的知识了

抽象类

抽象类(AbstractClass)是指使用 abstract关键字修饰的类，也就是在声明一个类时加入了 abstract关键字。抽象类是一种特殊的类，其它未使用abstract关键字修饰的类一般称作实体类。

```
public abstract class A{
  public A(){}
}
```

抽象类和实体类相比，主要有以下两点不同：抽象类不能使用自身的构造方法创建对象(语法不允许)
例如下面的语法是错误的：

```
A a = new A();
```

但是抽象类可以声明对象，例如下面的代码是正确的：

```
A a;
A a1,a2;
```

说明：抽象类可以使用子类的构造方法创建对象。
抽象类内部可以包含任意个(0个、1个或多个)抽象方法
抽象类内部可以包含抽象方法，也可以不包含抽象方法，对于包含的个数没有限制。而实体类内部不能包含抽象方法。
在抽象类内部，可以和实体类一样，包含构造方法、属性和实体方法，这点和一般的类一样。

抽象类的用途

严禁直接创建该类的对象
如果一个类内部包含的所有方法都是 static方法，那么为了避免其它程序员误用，则可以将该类声明为 abstract，这样其它程序员只能使用类名.方法名调用对应方法，而不能使用对象名.方法名进行调用。这样的类例如 API中的 Math类
强制子类覆盖抽象方法

接口

接口就是一个纯粹用来设计的数据类型，在接口这种数据类型中，只能书写两类声明的结构
所有的常量数据都是 public static的。如果声明时不书写则系统将自动添加这两个修饰符。

接口中的所有方法都只在逻辑上规定该方法的作用，而不能书写方法体。所有接口中的方法都是 public abstract的，如果声明时不书写则系统将自动添加这两个修饰符。

```
访问控制符 interface接口名 [extends父接口名 1,父接口名 2……]{
  常量声明
  方法声明
}
```

和类的声明一样，访问控制符只能使用 public和默认的。声明时使用interface关键字声明接口，接口可以继承其它的接口，使用 extends关键字进行继承，多个接口名之间使用逗号进行分隔。和类的集成一样，子接口继承父接口中所有的常量数据和方法，子接口的对象也是父接口的对象。
注意：和抽象类一样，接口只能声明对象，而不能创建对象。

抽象类和接口的区别(不同点)

a)抽象类是类，而接口是接口。因为抽象类是一个类，所以类内部可以包含的内容(构造方法、方法和属性等)在抽象类内部都可以存在，当然抽象类也受到类的单重继承的限制。而接口是接口类型，所以接口内部只能包含常量属性和抽象方法，但是一个类可以实现多个接口，所以接口不受类的单重继承的限制。

b)抽象类内部可以包含实体方法，而接口不能抽象类是一个类，所以在抽象类内部既可以包含抽象方法也可以包含实体方法，而接口内部的每个方法都必须是抽象方法。

c)抽象类可以继承类，而接口不能 抽象类是一个类，所以在设计时可以使抽象类继承其它的类，在已有类的基础上进行设计，但是接口不能继承类。

抽象类和接口的联系(相同点)

a)抽象类和接口都可以声明对象，但是都只能使用子类的构造方法进行创建。

b)抽象类和接口内部都可以包含抽象方法。 按照Java语言的语法，子类在继承抽象类或实现接口时，都必须覆盖这些抽象方法，否

则必须声明为抽象类。

c)抽象类和接口都可以代表一种类型，从而可以统一子类对象的类型，获得良好的可扩展性。

内部类

内部类是 Java语言中的一种特殊的语法，简单的来说，就是在一个类的内部再声明一个类，这些声明在类内部的类就被称作内部类。在实际声明时，内部类可以声明在类的内部、类的方法内部，也可以声明在类的构造方法内部，内部类声明的语法格式和一般类的声明一样，只是内部类声明时可以使用static修饰符进行修饰

```
/**
*内部类基本使用示例代码 */
public class OutClass{
  int i = 0;
  public class InnerClass{
    public void test(){
      i++;
    }
  }
}
```

内部类可以访问外部类的所有属性和方法，避免了参数传递

在内部类中，可以访问外部类中的所有属性和方法，private访问控制符修饰的属性和方法也可以被内部类访问，这样将方便内部类的编写，避免了参数传递，也减少了外部类需要向其它类开放的属性和方法的数量。

通过内部类可以实现多继承。

课间习题时间：

程序一

```
1  public class Main {
2      public static final A Aobj=null;
3      public static void main(String[] args) {
4          Aobj=new A();
5      }
6  }
7  class A{
8  }
```

程序二

```
01  public class Test {
02      public static void main(String[] args) {
03          MyClass obj=new MyClass();
04          obj.Info+=",World!";
05          System.out.println(obj.Info);
06      }
07  }
08  class MyClass  {
09          public String Info="Hello";
10          public MyClass(String Info)
11           {
12               this.Info=Info;
13           }
14  }
```

程序三

```
01  abstract class Animal  {
02  public abstract  void saySomething()    {
03       System.out.println("你想说什么就说吧！");
04       }
05  }
06  class Dog extends Animal {
07  public void saySomething() {
08       System.out.println("我现在有了第二职业：捉
耗子！");
09       }
10  }
11  public class Test {
12      public static void main (String[] args)
{
13             Animal ani=new Dog();
14             ani.saySomething();
15  }
16  }
```

Overload和Override的区别

接口是否可继承接口? 抽象类是否可实现(implements)接口? 抽象类是否 可继承实体类

STRING与STRINGBUFFER的区别

抽象类和接口的区别

在java中一个类被声明为final类型，表示了什么意思

设计模式

==========

工厂方法模式：负责实例化同一接口的多个类。工厂方法模式的意义是定义一个创建产品对象的工厂类，由该工厂统一创建继承了同一个接口的多个产品对象。

分为个3类型：

工厂方法模式

多个工厂方法模式

简单工厂模式

1）工厂方法模式

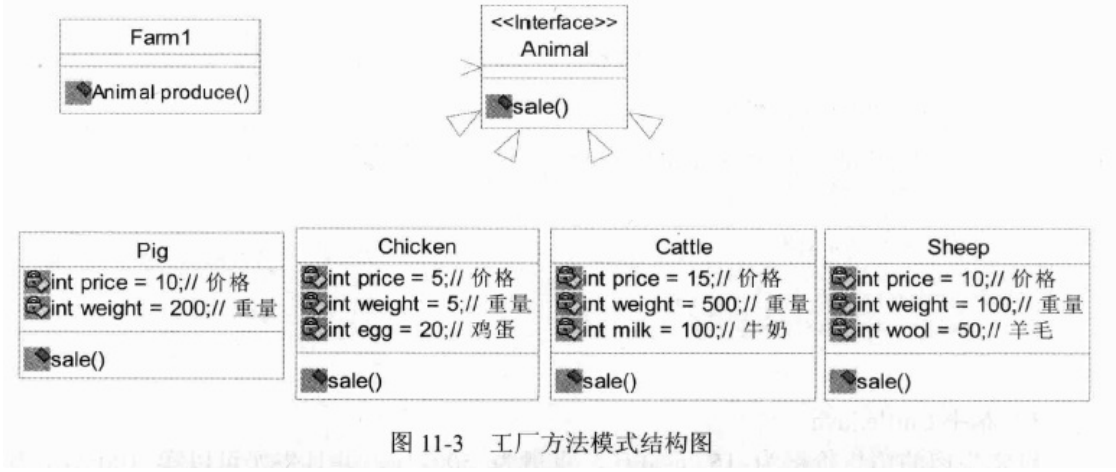


图 11-3　工厂方法模式结构图

```
package creation.factorymethod;

/**
 * @author liuzhongbing
 * 工厂方法模式
 */
public class Farm1 {
    public Animal produce(String type) {
        if (type == "pig") {
            return new Pig();
        } else if (type == "chicken") {
            return new Chicken();
        } else if (type == "cattle") {
            return new Cattle();
        } else if (type == "sheep") {
            return new Sheep();
        } else {
            return new Chicken();
        }
    }
}
```

Terry:通过type决定生产哪个animal。

2）多个工厂方法模式

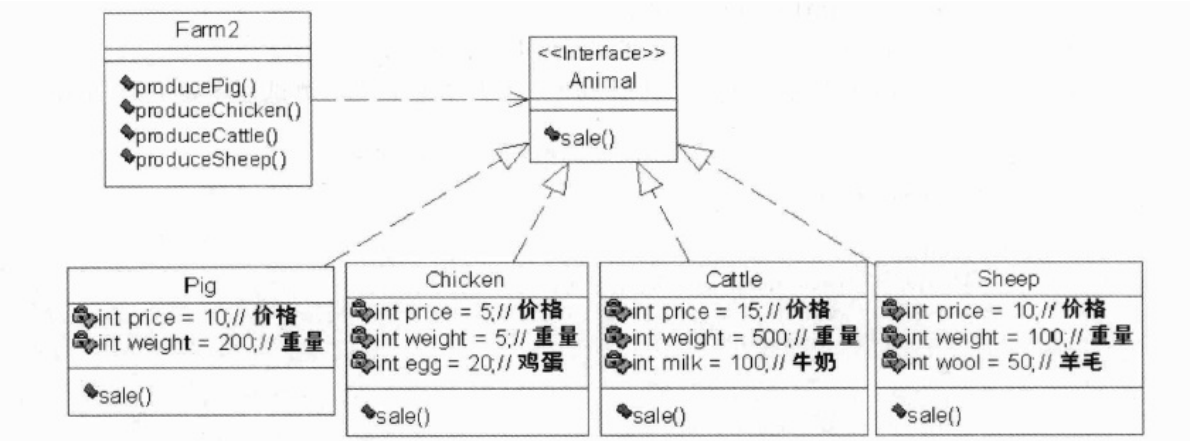


图 11-4　多个工厂方法模式结构图

```
package creation.factorymethod;

/**
 * @author liuzhongbing
 * 多个工厂方法模式
 */
public class Farm2 {
    public Animal producePig() {
        return new Pig();
    }

    public Animal produceChicken() {
        return new Chicken();
    }

    public Animal produceCattle() {
        return new Cattle();
    }

    public Animal produceSheep() {
        return new Sheep();
    }
}
```

2）静态工厂方法模式

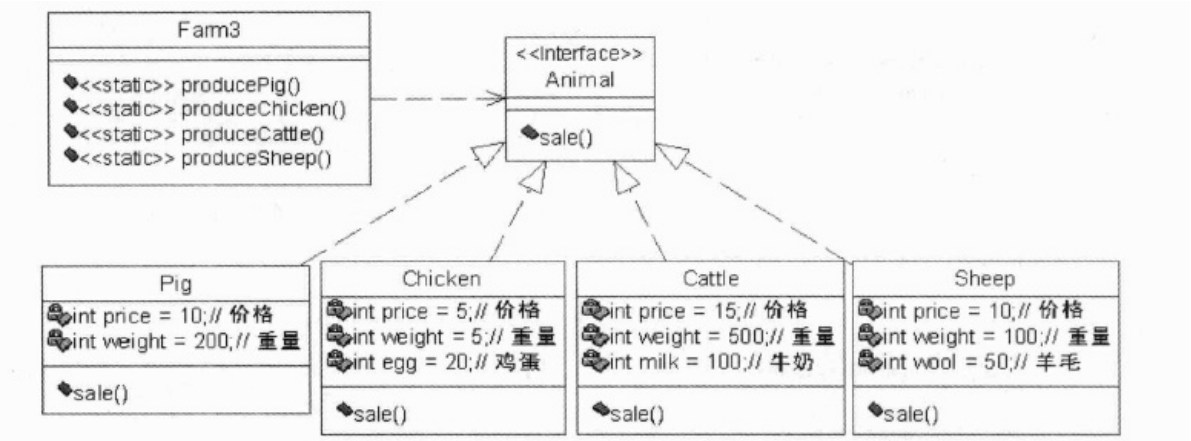


图 11-5　静态工厂方法模式结构图

```
package creation.factorymethod;

/**
 * @author liuzhongbing
 * 简单工厂模式（静态工厂方法模式）
 */
public class Farm3 {
```

```
    public static Animal producePig() {
        return new Pig();
    }

    public static Animal produceChicken() {
        return new Chicken();
    }

    public static Animal produceCattle() {
        return new Cattle();
    }

    public static Animal produceSheep() {
        return new Sheep();
    }
}
```

抽象工厂方法模式：如果产品的树形结构需要扩展，就必须在工程类中为新增的产品增加创建功能，这显然违背了开闭原则-在扩展时不能够修改原有的代码。

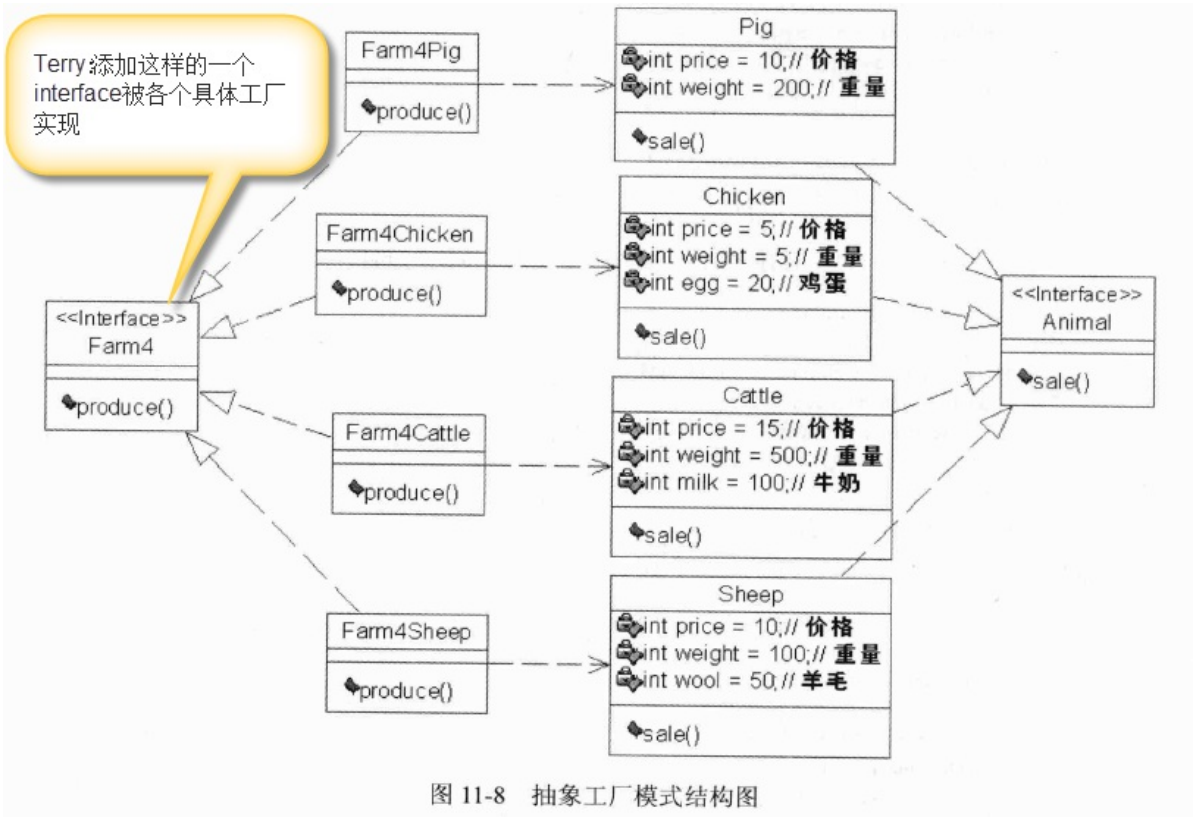


图 11-8　抽象工厂模式结构图

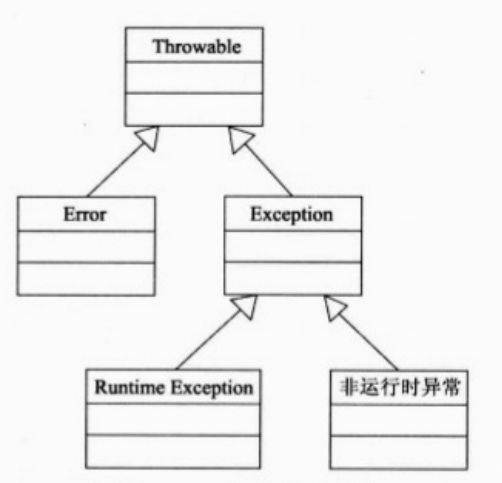
```
package creation.abstractfactory;

/**
 * @author liuzhongbing
 * 抽象工厂模式
 */
public interface Farm4 {
    public Animal produce();
}
```

```
package creation.abstractfactory;

public class Farm4Pig implements Farm4 {
    public Animal produce() {
        return new Pig();
    }
}
```

。。。。。。



异常处理

在实际的项目中，程序执行时经常会出现一些意外的情况，这些意外的情况会导致程序出错或者崩溃，从而影响程序的正常执行，如果不能很好的处理这些意外情况，将使项目的稳定性不强

对于这些程序执行时出现的意外情况，在 Java语言中被称作异常(Exception)，出现异常时相关的处理则称之为异常处理。

我们看一下下面的例子：

```
/**
*异常出现示例
*/
public class ExceptionDemo{
  public static void main(String[] args){
    String s = null;
    int len = s.length();
  }
}
```

在运行该程序时，在控制台的输出结果如下：

```
Exception in thread "main" java.lang.NullPointerException at ExceptionDemo.main(Exc
eptionDemo.java:7)
```

Java API中专门设计了java.lang.Throwable类，只有该类子类的对象才可以在系统的异常传递体系中进行。该类的两个子类分别是：

Error类

该类代表错误，指程序无法恢复的异常情况。对于所有错误类型以及其子类，都不要求程序进行处理。常见的 Error类例如内存溢出StackOverflowError等。

Exception类

该类代表异常，指程序有可能恢复的异常情况。该类就是整个 Java语言异常类体系中的父类。使用该类，可以代表所有异常的情况。

Exception类

RuntimeException及其所有子类

该类异常属于程序运行时异常，也就是由于程序自身的问题导致产生的异常，例如数组下标越界异常 ArrayIndexOutOfBoundsException等。该类异常在语法上不强制程序员必须处理，即使不处理这样的异常也不会出现语法错误

其它 Exception子类

该类异常属于程序外部的问题引起的异常，也就是由于程序运行时某些外部问题导致产生的异常，例如文件不存在异常 FileNotFoundException等。

该类异常在语法上强制程序员必须进行处理，如果不进行处理则会出现语法错误。

异常类类名	功能说明
java.lang.NullPointerException	空指针异常，调用null 对象中的非static 成员变量或成员方法时产生该异常
java.lang.ArrayIndexOutofBoundsException	数组下标越界异常，数组下标数值小于0 或大于等于数组长度时产生该异常
java.lang.IllegalArgumentException	非法参数异常，当参数不合法时产生该异常

抛出异常

```
throw new NullPointerException();
```

或

```
IllegalArgumentExceptione = new IllegalArgumentException();
throw e;
```

```
/**
*将自然数转换为二进制或八进制字符串
*@paramvalue需要转换的自然数
*@paramradix基数，只能取 2或 8
*@return转换后的字符串
*/
public static String toString(int value,int radix){
  //判断异常的代码
  if(value <0){
    throw newIllegalArgumentException("需要转换的数字不是自然数！");
  }
  if(radix != 2 && radix != 8){
    throw newIllegalArgumentException("进制参数非法");
  }
  if(value == 0){
    return "0";
  }
  StringBuffer s = new StringBuffer();
  int temp;     //余数
  while(value != 0){     //未转换结束
    temp = value % radix;        //取余数
    s.insert(0,temp);    //添加到字符串缓冲区
    value /= radix;      //去掉余数
  }
  return s.toString();
}
```

这里，当 value的值小于 0时，则抛出非法参数异常，当 radix的值不是 2或 8时，则抛

这样在执行如下代码：

```
System.out.println(toString(12,2));
System.out.println(toString(12,16));
```

则程序的执行结果是：1100

Exception in thread "main" java.lang.IllegalArgumentException:进制参数非法

```
atThrowException.toString(ThrowException.java:22)
atThrowException.main(ThrowException.java:7)
```

声明异常

异常虽然被抛出了，但是由于抛出异常的代码是在方法或构造方法的内部的，在调用方法或构造方法时一般是无法看到方法或构造方法的源代码的，这样调用的程序员就无法知道该方法或构造方法将出现怎样的异常情况，所以需要有一种语法，可以使得调用的程序员可以看到被调用的结构可能出现的异常情况，这就是声明异常的语法。

声明异常的语法类似于药品上的副作用说明，在患者服用药品时，知道药品的正常功能，但是无法详细了解药品的成分以及每种成分的含量（类似于源代码），但是在药品的说明上都有副作用的说明，例如过敏者不能服用等，这些和声明异常的语法在功能上是类似。

声明异常的语法格式为：
throws异常类名

该语法使用在方法和构造方法的声明以后，在 throws关键字以后，书写该方法或构造方法可能出现的异常，在这里需要书写异常类的类名，如果有多个，则使用逗号分隔这些异常类名即可。
这里需要注意的是：
这些异常必须是该方法内部可能抛出的异常
异常类名之间没有顺序
属于 RuntimeException子类的异常可以不书写在 throws语句以后，但是另外一类异常如果可能抛出则必须声明在 throws语句之后
通过在对应的方法或构造方法声明中书写 throws语句，使得调用该方法或构造方法的程序员可以在调用时看到对应结构可能出现的异常情况，从而提示对于这些异常情况进行处理，从而增强程序的健壮性。

捕获异常及异常处理

为了捕获异常并对异常进行处理，使用的捕获异常以及处理的语法格式为：

```
try{
  //逻辑代码
}catch(异常类名参数名){
  //处理代码
}
```

在该法中，将正常的程序逻辑代码书写在 try语句块内部进行执行，这些代码为可能抛出异常的代码，而 catch语句中书写对应的异常类的类名，在 catch语句块内部书写出现该类型的异常时的处理代码。
程序执行到 try-catch语句时，如果没有发生异常，则完整执行 try语句块内部的所有代码，而 catch语句块内部的代码不会执行，如果在执行时发生异常，则从发生异常的代码开始，后续的 try语句块代码不会执行，而跳转到该类型的异常对应的 catch语句块中。

在实际程序中，也可以根据异常类型的不同进行不同的处理，这样就需要多个 catch语句块，其结构如下：

```
try{
  //逻辑代码
}catch(异常类名 1参数名 1){
  //处理代码 1
}catch(异常类名 2参数名 2){
  //处理代码 2
}
……
}catch(异常类名 n参数名 n){
  //处理代码 n
}
```

例如：

```
String s = "123";
try{
  intn= Integer.parseInt(s);
  System.out.println(n);
  charc = s.charAt(4);
  System.out.println(c);
}catch(NumberFormatExceptione){
  System.out.println("该字符串无法转换！");
}catch(StringIndexOutOfBoundsExceptione){
  System.out.println("字符串索引值越界 ");
}
```

在实际使用时，由于 try-catch的执行流程，使得无论是 try语句块还是 catch语句块都不一定会被完全执行，而有些处理代码则必须得到执行，例如文件的关闭和网络连接的关闭等，这样如何在 try语句块和 catch语句块中都书写则显得重复，而且容易出现问题，这样在异常处理的语法中专门设计了 finally语句块来进行代码的书写。语法保证 finally语句块内部的代码肯定获得执行，即使在 try或 catch语句块中包含 return语句也会获得执行

```
try{
  //逻辑代码
}catch(异常类名参数){
  //异常处理代码
}finally{
  //清理代码
}
```

最后，介绍一下使用异常处理语法时需要注意的问题：
书写在 try语句块内部的代码执行效率比较低。所以在书写代码时，只把可能出现异常的代码书写在 try语句块内部。
如果逻辑代码中抛出的异常属于 RuntimeException的子类，则不强制书写在 try语句块的内部，如果抛出的异常属于非 RuntimeException的子类，则必须进行处理，其中书写在 try语句块是一种常见的处理方式。
catch语句块中只能捕获 try语句块中可能抛出的异常，否则将出现语法错误

try {}里有一个return语句，那么紧跟在这个try后的finally {}里的code 会不会被执行，什么时候被执行，在return前还是后?

```
step1

try{
step2
System.exit(1);
}catch(NullPointException e){
  throw e;
  system.out.print(e.toString);
}finally{
  driver.quit();
  driver = new Browser(log,db,browserprofile);
try{
}catch(){
}
  driver.login;
}

step3
```

homework:Login接口，3个实现loging website的方式。

所有的xpath都写入interface.

Java的开发之道自学。