



第五章 类的进阶设计



第五章：类的进阶设计

类的进阶设计

5.1 JVM数据区

5.2 多态

5.3 对象初始化

5.4 抽象类和接口

5.5 实践：工厂模式

5.6 类的关系和设计原则

5.7 内部类

5.8 Lambda表达式

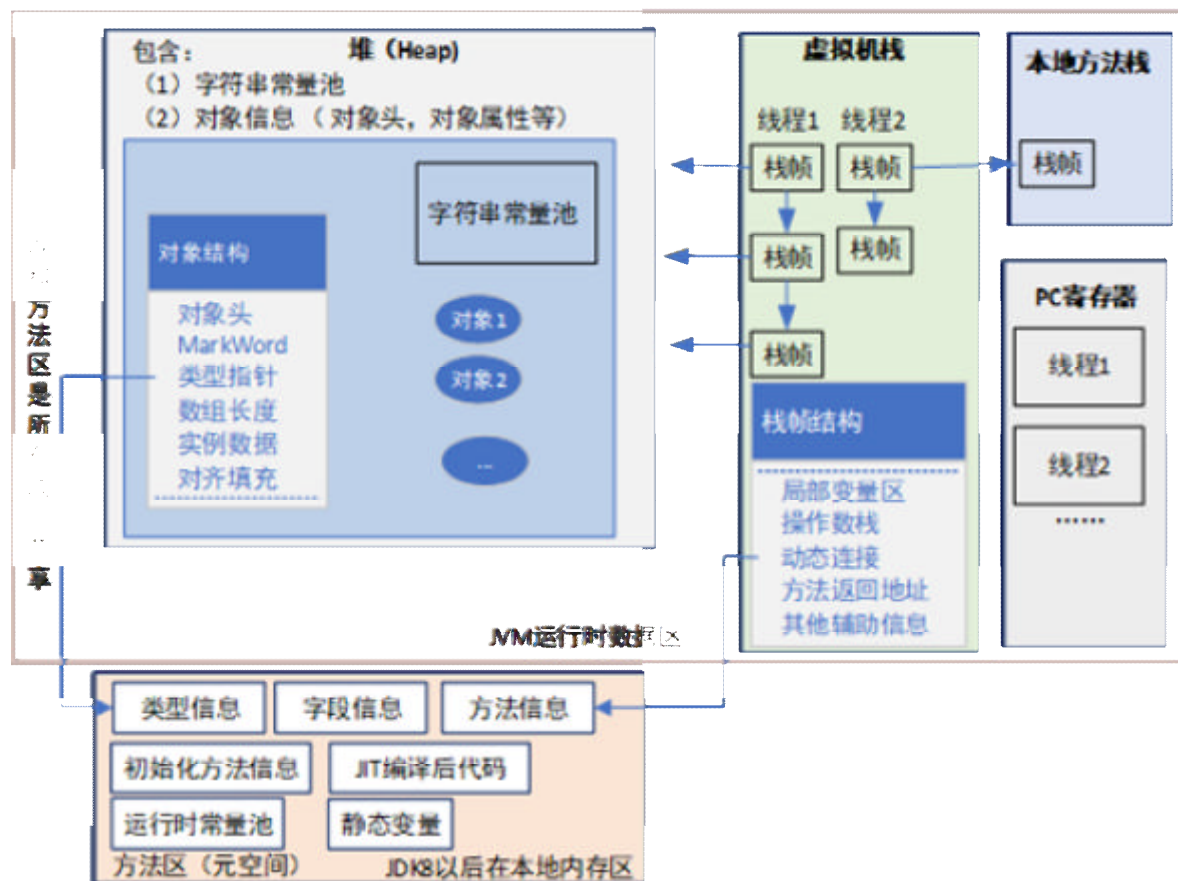
5.9 注解

第五章：类的进阶设计



5.1 JVM的数据区

JVM运行时数据区分为五部分：方法区（元空间）、堆(Heap)、程序计数器、JVM方法栈、本地方法栈。



第五章：类的进阶设计



5.2 多态

Java中的多态分为编译时多态和运行时多态

编译时多态：

主要通过方法的重载实现，是静态的。

运行时多态：

是通过方法重写/覆盖和动态绑定来实现的，即同一段程序可以与可互换的一类对象一起工作，这种能力被称为运行时多态。

运行时同一消息可以根据发送对象是子类还是父类对象而采用不同的行为。



5.2 多态



Java 实现运行时多态有三个必要条件：

向上转型
方法重写
动态绑定。

5.2.1 对象类型转换与instanceof

Java的继承层次创建了一个相关类型的集合，他们在使用中具有兼容性。

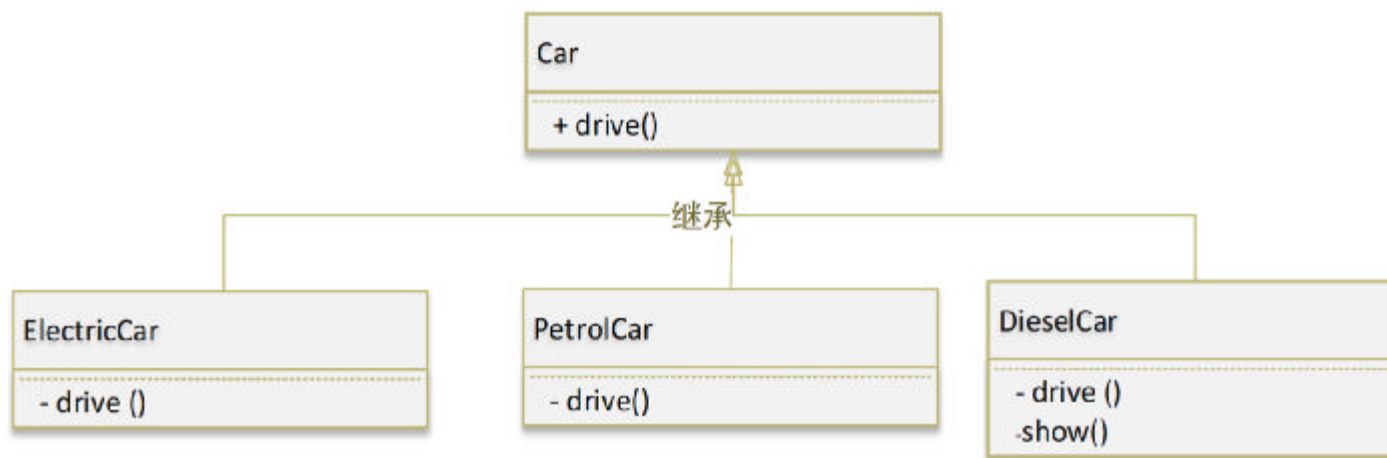
在继承结构中，Java 中引用类型之间的类型转换主要有两种：

- 向上转型（upcasting）
- 向下转型（downcasting）

5.2.1 对象类型转换与instanceof

1、向上转型：自动转换

对象除了属于自身类型，还可以作为它的父类类型对待，这叫作向上转型。



5.2.1 对象类型转换与instanceof

■ 自动转换

利用向上转型，Java 语言允许某个类型的引用指向其子类实例，即把子类对象直接赋给父类引用。

```
Car ins = new DieselCar();
```

```
Car ins = new PetrolCar();
```

自动转换对面向对象编程来说是非常重要的。因为它允许相同的方法参数，可以接收在继承结构里的不同对象。

5.2.1 对象类型转换与instanceof

2. 向上转型的损失

需要注意，使用向上转型的对象引用可以调用父类类型中的成员，但不能调用子类类型中扩展的成员。这是因为Java对象有两种类型：编译时类型和运行时类型。

(1) **编译时类型**：是对象引用的类型，其能调用的方法集由引用类型决定。

(2) **运行时类型**：是对象本身的类型，由存放在对象结构中的类型指针所决定，它决定了运行时运行哪个类型的方法。

自动转换后的损失

```
public class LossDemo{  
    public static void main(String arg[]) {  
        Car demo=new DieselCar();  
        demo.drive(); //合法, 在Car的可见方法集中  
        demo.show(); // 非法, 不在Car的方法集中, 编译时就会出错  
    }  
}
```

5.2.1 对象类型转换与instanceof

3. 向下转型：强制转换

为了解决向上转型不能调用子类扩展方法的问题，我们需要向下转型。与向上转型相反，向下转型是父类的引用被转变成为一个子类的引用，这是一种强制转换。

■强制转换

- 如果原来父类引用指向的是子类对象，那么在向下转型的过程中才是安全的，编译不会出错。例如：

```
Car ins=new PetrolCar();  
PetrolCar w= (PetrolCar)ins;
```

- 如果父类引用指向父类对象，那么在向下转型的过程中是不允许的，编译会出错。例如：

```
Car ins=new Car(); //父类引用指向父类对象  
PetrolCar w= (PetrolCar)ins; //非法，编译错误
```

自动&强制转换

```
public class UseCars1 {  
    public static Car  getCar(String i){ //统一的返回类型  
        Car ins=null;  
        if(i!=null)  
            switch(i){  
                case "Electric": ins=new ElectricCar(); //合法，向上转型  
                    break;  
                case "Petrol": ins=new PetrolCar(); //合法，向上转型  
                    break;  
                case "Diesel": ins=new DieselCar(); //合法，向上转型  
                    break;  
                default: ins=null;  
            }  
        return ins;  
    }  
    public static void main(String arg[]) {  
        DieselCar br =(DieselCar) getCar("Diesel"); //合法，向下转型  
        if(br!=null) br.show(); // 可以调用子类扩展方法  
    }  
}
```

4、 instanceof 操作符

- ◆ instanceof 可判断该变量属于哪个类。

对象 instanceof 类型 //返回值为boolean类型

- ◆ 若进行测试的引用为null值，其结果总是返回“false”。

例2: String s=null;
s instanceof String

4、instanceof操作符

也可以在超类转换成子类前，先使用instanceof进行检查

```
Car tempCar=getCar("Di esel "); //先获取返回值  
  
Di esel Car br;  
if ( tempCar instanceof Di esel Car)  
    br=(Di esel Car) tempCar; //安全强制转换
```

4、 instanceof 操作符

可写成 `if(tempIns instanceof Brass br)`
(JDK16)

例如:

```
Car br2=getCar("Diesel");  
    DieselCar br;  
if(br2 instanceof Diesel br)  
    br.show();
```

单选题 1分



```
class sup{}  
public class sub extends sup  
{  
    public static void main(String arg[])  
    {  
        sub sb1=new sub();  
        sup sp1=new sub();  
        sup sp2=new sup();  
        System.out.println("sp1 instanceof sub" + (sp1 instanceof sub));  
        System.out.println("sp2 instanceof sub:" + (sp2 instanceof sub));  
    }  
}
```

A true true

B false false

C false true

D true false

提交

第五章：类的进阶设计



5.2.2 方法重写（overriding）

- 方法覆盖体现了Java运行时的多态性。
- 存在于子类和父类之间，当父类中的方法在子类中获得重新定义时，如果方法的方法名、参数一致、返回值类型兼容，只有方法体发生了变化时，就称子类覆盖了父类方法（是一种取代关系）。
- 当父类方法无法满足子类需求时，可通过方法重写进行功能扩展。

在重写方法时，需要遵循下面的规则：

[访问/其他修饰符] 返回值类型 方法名（参数） **throws** 异常

(1)

(2)

(3)

(4)

(5)

- ◆ 1、访问权限: 子类重写方法的访问权限必须等于或高于父类权限。**private** 方法对子类不可见，所以不能被重写，否则只是新定义了一个子类方法，并没有对其进行覆盖。
- ◆ 2、**final**方法不能被重写；不能一个是**static**方法，一个是实例方法。
- ◆ 3、返回的类型：必须与被重写的方法的返回类型相同或兼容（Java1.5 版本之前返回值类型必须一样，之后的 Java 版本放宽了限制，返回值类型级别必须小于或者等于父类方法的返回值类型，返回值如果是引用类型，可是父类方法返回类型、或其子类型）。

在重写方法时，需要遵循下面的规则：

[访问/其他修饰符] 返回值类型 方法名（参数） throws 异常

(1)

(2)

(3)

(4)

(5)

- ◆ 4、方法名、参数列表必须完全与被重写方法的相同。
- ◆ 5、异常：子类重写方法声明的异常与父类的保持一致，或是父类方法返回值类型的子类。不能抛出在父类方法的throws语句中没被定义的异常。（后续第六章讲）。
- ◆ 6、重写的方法可以使用 `@Override` 注解来标识。注解 `@Override` 表示它所标注的方法必须是对父类的重写，如果在父类中没有，编译就会出错。

```
class Sup {  
    public void show()  
    { System.out.println("in superclass "); }  
}  
class Sub extends sup {  
    void show() //编译不通过（权限缩小）  
    { System.out.println(" in subclass"); }  
}
```

```
class Sup {  
    private void show() //private方法不能被重写  
    { System.out.println("in superclass "); }  
}  
  
class Sub extends sup {  
    public void show()  
    { System.out.println(" in subclass"); }  
}
```

父类private方法不可见，子类只是重新定义了一个新方法，编译通过，但不是重写父类方法。

■ @Override注解的使用

注解@Override表示它所标注的方法必须是对父类的重写，如果在父类中没有，编译就会出错。例如：

```
class Sup {  
    public void show() {  
        System.out.println("in superclass ");  
    }  
}  
class Sub extends Sup {  
    @Override  
    public void Show() { //编译错误，父类无Show方法  
        System.out.println(" in subclass");  
    }  
}
```

5.2.3 动态绑定（重点）

■ 动态绑定/链编：

将方法调用同其方法体连接起来叫做“绑定”（Binding）。绑定分为静态绑定和动态绑定（或后期绑定）。

静态绑定：是在程序执行前的由编译器或连接程序绑定，

动态绑定：是在运行时绑定，JVM通过对象的类型指针绑定方法。

5.2.3 动态绑定（重点）

■ 动态绑定/链编：

Java绑定规则如下：

- ① 被final、static、private修饰的方法执行静态绑定，与编译时类型的方法体进行绑定。
- ② 其余实例方法执行动态绑定，与对象的运行时类型的方法体进行绑定。
- ③ 成员变量（包括静态变量和实例变量）执行静态绑定，与引用类型的成员变量绑定。

动态绑定非常重要，使得无需对现存代码进行修改，就可对程序进行扩展。

方法静态绑定。

【运行结果】
in superclass

```
class Sup2 {
    static void show() { System.out.println("in superclass"); }
}
class SubX2 extends Sup2 {
    static void show() { System.out.println(" in subclass"); }
}
public class BindTest {
    public static void main(String arg[]) {
        Sup2 s = new SubX2();
        s.show();    //静态（或者private）方法静态绑定
    }
}
```

实例方法动态绑定与成员变量静态绑定

```
class Sup3{  
    int i=5;  
    void show(){  
        System.out.println("i in superclass is:"+i);  
    }  
}
```

```
class SubX3 extends Sup3{  
    int i=6;  
    void show(){  
        System.out.println("i in subclass is:"+i);  
    }  
}
```

```
class DyStaticBind{  
    public static void main(String[] arg){
```

```
        Sup3 sp=new SubX3();
```

```
        sp.show(); //实例方法， 动态绑定
```

```
        System.out.println(sp.i); //成员变量， 静态绑定
```

```
i in subclass is: 6  
5
```

多态程序设计

```
public class useCars2 {  
    public static void showInfo(Car ins) { //上述对象用统一的类型Car  
        ins.drive(); //同样的方法调用，对不同的输入类型展现多态性。  
    }  
    public static void main(String arg[]) {  
        //多样的子类对象都可以赋给父类引用  
        showInfo(new ElectricCar());  
        showInfo(new PetrolCar());  
        showInfo(new DieselCar());  
    }  
}
```

【运行结果】

```
in ElectricCar class  
in PetrolCar class  
in DieselCar class
```

第五章：类的进阶设计



5.3 对象初始化

对象初始化是指为对象的字段赋以初值。常用方法：

- 1、直接赋值
- 2、使用默认值
- 3、通过初始化语句块
- 4、通过构造方法

成员变量类型	初始值
byte/short/int	0
long	0L
float	0.0F
double	0.0
char	'\u0000' 是一个不可见字符
boolean	false
引用型	null

5.3 对象初始化

2、对象初始化顺序：先父类后子类数据

- 1、若是第一个对象，初始父类和子类中的static数据（只一次）
- 2、初始化父类的其他数据域
- 3、调用父类的构造方法，若子类的构造方法未通过super语句显式调用父类构造方法，则系统会自动先调用父类的无参构造方法（该情况下，父类没有无参构造方法会编译报错）。
- 4、初始化子类中添加的数据域
- 5、执行子类构造方法中的余下的操作。

```

class Base2{
    static int x1=show("static Base2.x1 init");
    int i=5 , j;
    Base2(){
        show(("in Base2 constructor, i= "+i+", j="+j));
        j=20 ;
    }
    static int show(String s) {
        System.out.println(s);
        return 50; }
}

```

```

class SubObject extends Base2{
    static int x2=show("static SubObject.x2 init");
    int k=show("SubObject.k init");
    SubObject(){
        show("in SubObject constructor, k= " + k + " j= " + j);
    }
    public static void main(String arg[]){
        System.out.println("*****the first object*****");
        SubObject b=new SubObject();
        System.out.println("*****the second object*****");
        SubObject b1=new SubObject();
    }}

```

【运行结果】

static Base2.x1 init

static SubObject.x2 init

*****the first object*****

in Base2 constructor, i= 5, j=0

SubObject.k init

int subObject constructor,k= 50 j= 20

*****the second object*****

in Base2 constructor, i= 5, j=0

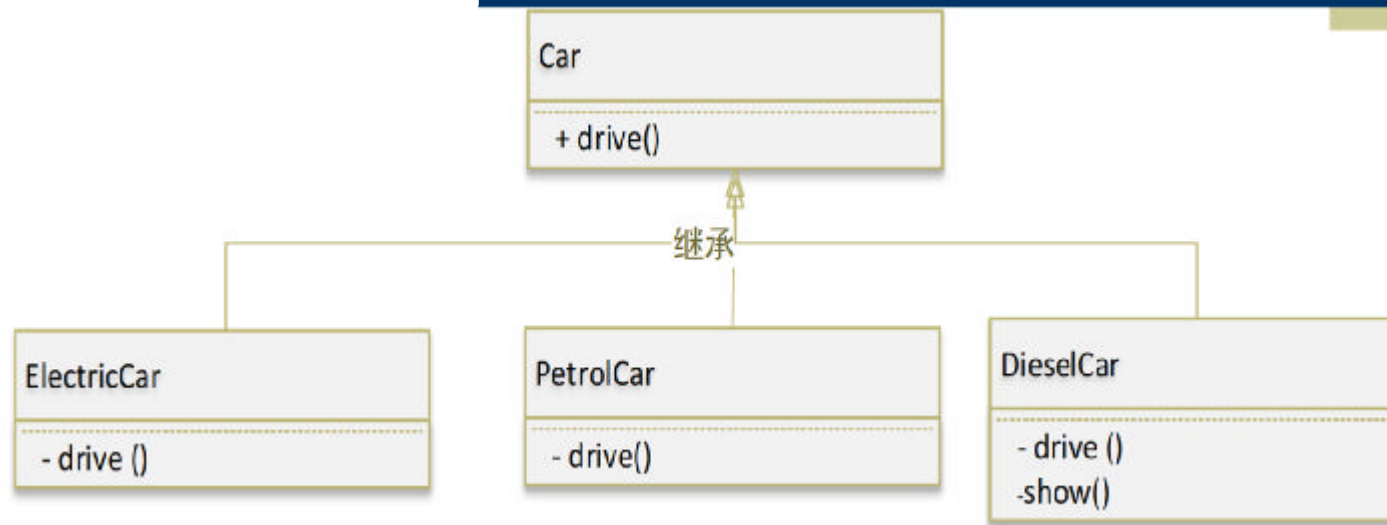
SubObject.k init

int subObject constructor,k= 50 j= 20

第五章：类的进阶设计



5.4 接口 和 抽象类



- ◆ Java提供了“抽象方法”机制，来创建统一的方法接口，这种方法称为abstract方法，是仅有方法声明而没有方法体的方法。
- ◆ Java提供了封装抽象方法的抽象类和接口，用以提供强制机制来保证子类必须重写父类的抽象方法。

5.4.1 abstract method

[访问权限修饰符] **abstract** 返回类型 方法名（[形参表]）；

例如：public abstract void drive();

注意：

抽象方法是要被子类重写的，所以，访问权限不能是private，且不能被final、static修饰。

5.4.2 abstract 类

```
[访问修饰符] abstract class 类名{  
    //类体  
}
```

```
abstract class Car{  
    public abstract void drive();  
}
```

注意：

- (1) 抽象类可以包含零个到多个成员变量、普通方法，也可以含零个到多个抽象方法。
- (2) 抽象类不能被final修饰。
- (3) 一个类只要有一个方法是抽象方法，这个类就要定义成抽象类。

5.4.3 接口

- **接口**：接口用于规范对象的公共行为，提供比抽象类更高级别的抽象，用来定义全局常量和公开抽象方法。

5.4.3 接口

■1、接口定义

常规定义如下：

```
[修饰符] interface 接口名[extends 接口1, 接口2, ...]{  
    [public static final] 类型 变量=初始值;  
    [public abstract] 返回类型 方法名(形参表);  
}
```

说明：

- (1) 所有的方法都是公开的抽象方法，所以，方法可以省略public 和abstract关键字。
- (2) 所有的字段都是公开的静态常量，所以，字段可省略public、static、final关键字。需要定义时赋初值。
- (3) 类似抽象类，接口不能被实例化，只能被实现。
- (4) 接口可以继承其他接口（不能是类），而且支持多继承，多个接口用“,”分割。这一点与类不同。

```
interface A {
    String id; //非法，接口中静态常量需要声明时就初始化
    int flag=2; //等价于 public static final int flag=2
    int size =10; //等价于 public static final int size=10
    void show(); //等价于 public abstract void show()
}

interface B{
    int flag=5;
    void show();
    boolean compare(B temp);
}

public interface MyInterface extends A, B {
    int myFlag=A. flag+B. flag;
}
```

5.4.3 接口

2、接口实现

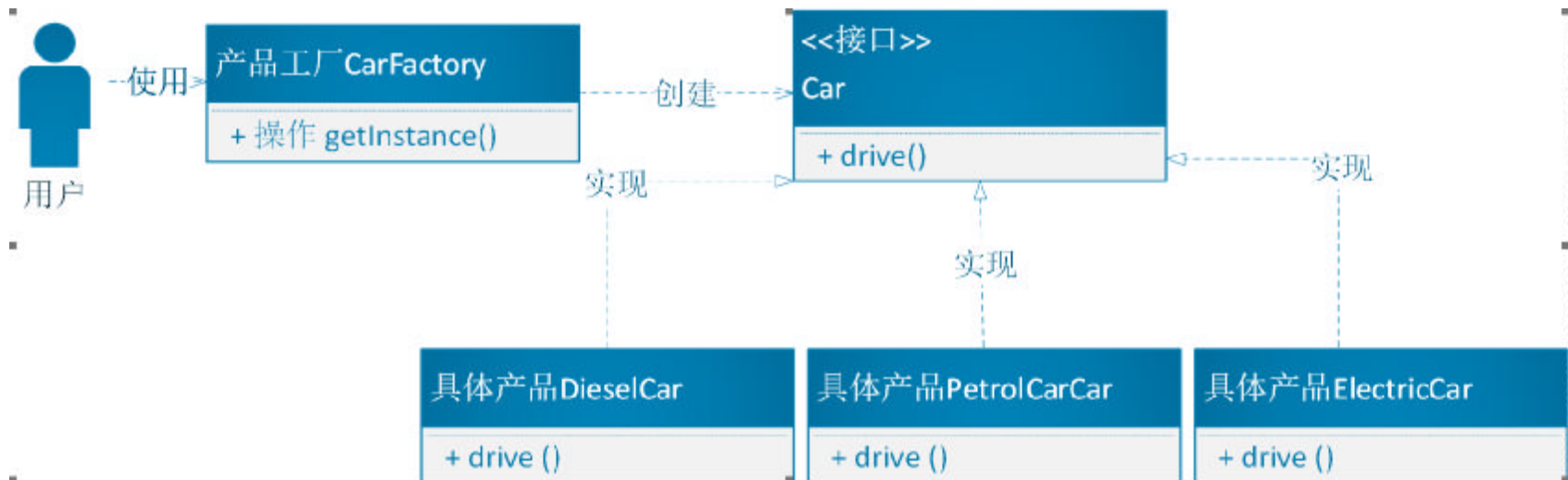
类实现接口，体现了规范和实现分离的设计原则。

语法：[访问符] **class** 类名 **implements** 接口1, [接口2, ..]

- (1) Java不支持类的多重继承，但可实现多个接口，而现实世界中有些事物是属于多类事物，通过接口变相实现多重继承。
- (2) 实现类如果有一个方法没被实现，则该类为抽象类。
- (3) 接口类型的引用可指向实现类对象，支持向上转型。与类继承有所不同，实现接口真正表示的是like-a的逻辑。
- (4) 接口中的方法都是public，所以，重写接口中的方法都必须实现为public权限。


例如：利用简单工厂模式，改写程序

工厂模式专门用于创建对象，工厂模式对外屏蔽对象的具体创建细节，并通过公开接口向对象使用者提供新创建的对象。其中简单工厂模式是工厂模式的一种。




```
interface Car{
    void drive();
}
class DieselCar implements Car{ //具体类
    @Override
    public void drive() {System.out.println("in DieselCar class");}
}
class PetrolCar implements Car{//具体类
    @Override
    public void drive() {System.out.println("in PetrolCar class");}
}
class ElectricCar implements Car{//具体类
    @Override
    public void drive() {System.out.println("in ElectricCar class");}
}
```

```
enum CarType{ DieselCar, PetrolCar, ElectricCar} //产品类型
```



```
class CarFactory{ //工厂类
    public static Car getInstance(CarType ct) { //生成产品，向上转型
        if (ct!=null)
            switch(ct) {
                case DieselCar: return new DieselCar();
                case PetrolCar: return new PetrolCar();
                case ElectricCar: return new ElectricCar();
                default: return null;
            }
        return null;
    }
}

class useFactory{ //用户类
    public static void main(String arg[]) {
        Car c1=CarFactory.getInstance(CarType.ElectricCar);
        Car c2=CarFactory.getInstance(CarType.DieselCar);
        c1.drive();
        c2.drive();
    }
}
```



5.4.3 接口

3、接口中的default/static方法（了解即可）

- 1、接口处于兼容类型的顶层，如果在常用接口中要增加一个新的抽象方法，所有实现类都必须重写，这是无法想象的。但Java不断有新增语法出现，这要求一些常用的接口能适应这些变化，比如Iterable接口新增forEach方法。
- 2、为了能够在不影响已有实现类的前提下为接口增加新的行为，接口中出现了default/static方法。
- 3、含默认方法的接口使用较少，初学者可以先不用关注此特性。

3、接口中的default/static方法（了解即可）

default/static方法。定义格式：

```
public default 返回值类型 方法名（参数列表）{  
    //方法体  
}  
public static 返回值类型 方法名（参数列表）{  
    //方法体  
}
```

注意：

1. 默认方法（default）方法不是抽象方法，可以被重写，但不强制。重写时去掉default关键字。另外，不能直接使用接口调用default方法,需通过接口的实现类对象来调用。
2. 静态（static）方法只能通过接口名调用，不能通过实现类名或者对象名调用。
3. public可省略，default/static不能省略，两者分别修饰实例方法和静态方法。


```

interface DefaultStaticInterface{
    void show(); //抽象方法

    default void defaultInfo(){
        System.out.println(" default method in interface");
    }
    default void defaultInfo2(){
        System.out.println("default method in interface(no overriding)");
    }
    static void staticInfo(){
        System.out.println("static method in interface ");
    }
}

public class DefaultStaticClass implements DefaultStaticInterface{
    public void show() {
        System.out.println("override abstract method in class");
    }
    public void defaultInfo(){
        System.out.println("override default method in class");
    }
    public static void main(String arg[]) {
        DefaultStaticInterface ds=new DefaultStaticClass();
        ds.show();           //合法，调用实现的抽象方法
        ds.defaultInfo();    //合法，调用被重写的默认方法
        ds.defaultInfo2();   //合法，调用来自接口的默认方法
        //ds.staticInfo();   //非法，接口中的static方法，只能通过接口调用。
        DefaultStaticInterface.staticInfo(); //合法，用接口调用静态方法。
    }
}

```

5.4 接口 和 抽象类

4、接口与抽象类的比较

抽象类与接口是进行抽象的两种机制，有相似性，也有区别。

- ◆ **语法上：**抽象类用关键字`abstract class`定义，可定义成员变量和非抽象方法；接口用`interface`定义，接口内只有公开的静态常量、成员方法都是公开的，且只能是抽象方法、默认方法或静态方法。
- ◆ **使用上：**抽象类是用来被单继承的，而一个类可以实现多个接口。
- ◆ **设计上，**抽象类作为父类，与子类之间存在“is-a”关系，即父子类本质上是一种类型。接口只能表示类支持接口的行为，具有接口的功能。因此接口和实现类之间表示的是“like-a”关系。因此，在设计上，如果父子类型本质上是一种类型，那父类可设计成抽象类，如果子类型只是想额外具有一些特性，则可以将父类型设计成接口，而且这些接口不易过大，应该设计成多个专题的小接口。这也是面向对象设计的一个重要原则——接口隔离原则。

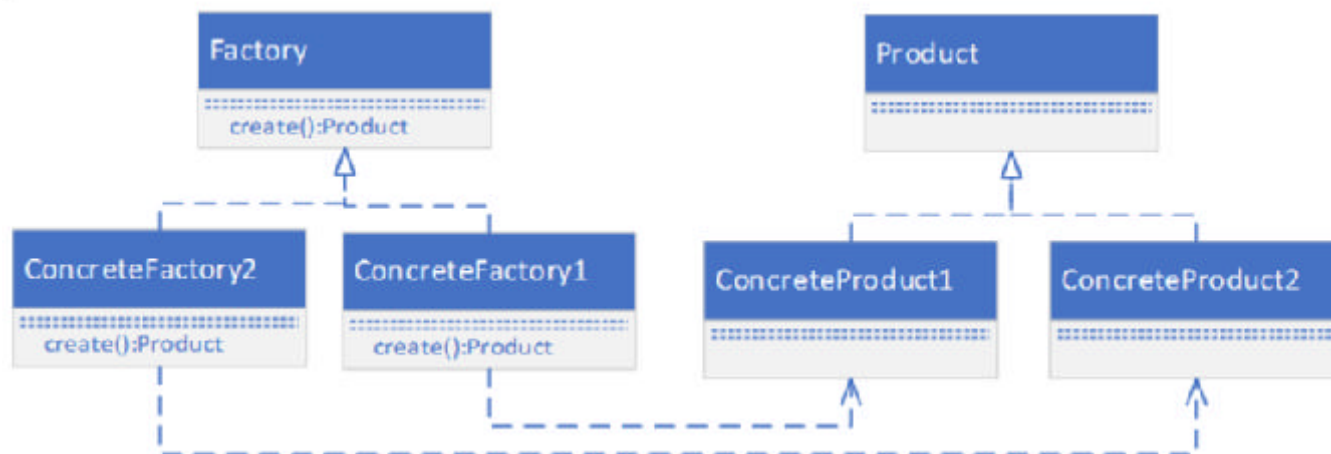
第五章：类的进阶设计



5.5 实践：工厂方法模式

工厂模式包括 简单工厂（Simple Factory）模式、工厂方法（Factory Method）模式和抽象工厂（Abstract Factory）模式。

在工厂方法模式中，核心工厂类不再负责所有产品的创建，仅负责给出具体工厂类必须实现的接口，具体创建的工作交给工厂子类去做。工厂方法模式如图5-4所示，主要由这几部分组成：



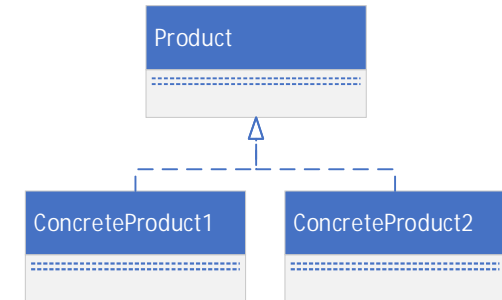
【例5.14】工厂方法模式示例。

需求：为动物园进行动物管理：

- 每一只动物入园要生成一只动物对象，并显式当前的动物信息：
- 信息包括当前这种类型的动物有几只，目前动物园共有多少只动物。
- 后期系统会不定期进行动物种类更新，要求为系统提供扩展性。

【例5.14】工厂方法模式示例。

```
abstract class Animal { //抽象动物类
    private static int sum=0; //所有动物的全局变量
    public Animal () {
        sum++;
    }
    public int getSum() {
        return sum;
    }
    public abstract String getInfo();
} //Animal 定义结束
```



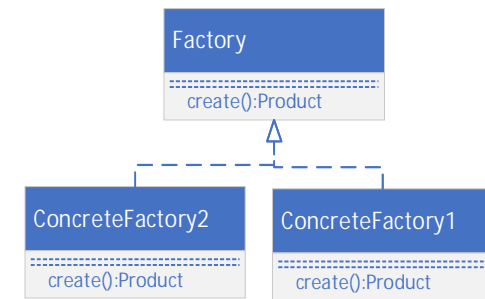
```
class Cat extends Animal {
    private static int count=0; //所有猫的全局变量
    private int id;
    public Cat () { count++; id=count;
    }
    public String getInfo() {
        return "我是第"+id+"只猫，共生成了"+getSum()+"只动物";
    }
} //Cat 定义结束
```

```
class Pig extends Animal {  
    private static int count=0; //所有猪的全局变量  
    private int id;  
    public Pig() {  
        count++; id=count;  
    }  
    public String getInfo() {  
        return "我是第"+id+"只猪，共生成了"+getSum()+"只动物";  
    }  
} //Pig定义结束
```

```
interface Factory{ //抽象工厂接口
    public Animal create();
}
```

```
class CatFactory implements Factory{ //具体的猫工厂类
    public Animal create() {
        return new Cat();
    }
}
```

```
class PigFactory implements Factory{ //具体的猪工厂类
    public Animal create() {
        return new Pig();
    }
}
```



【运行结果】

我是第1只猫，共生成了1只动物

我是第2只猫，共生成了2只动物

我是第1只猪，共生成了3只动物

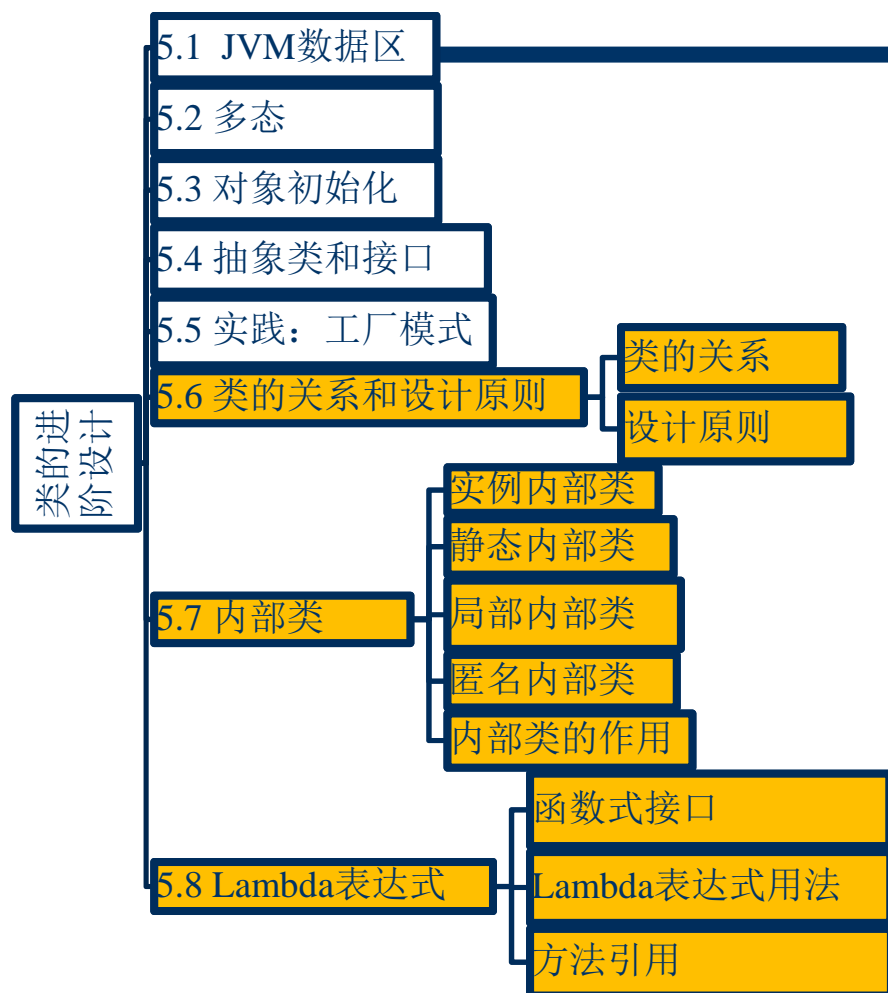
```
public class FactoryMethodDemo { //使用工厂模式
    public static void main(String arg[]) {
        Factory f1, f2;
        Animal a1, a2, a3;
        f1=new CatFactory();
        a1=f1.create();
        System.out.println(a1.getInfo());
        a2=f1.create();
        System.out.println(a2.getInfo());
        f2=new PigFactory();
        a3=f2.create();
        System.out.println(a3.getInfo());
    }
}
```



第五章 类的进阶设计（2）



第五章：类的进阶设计



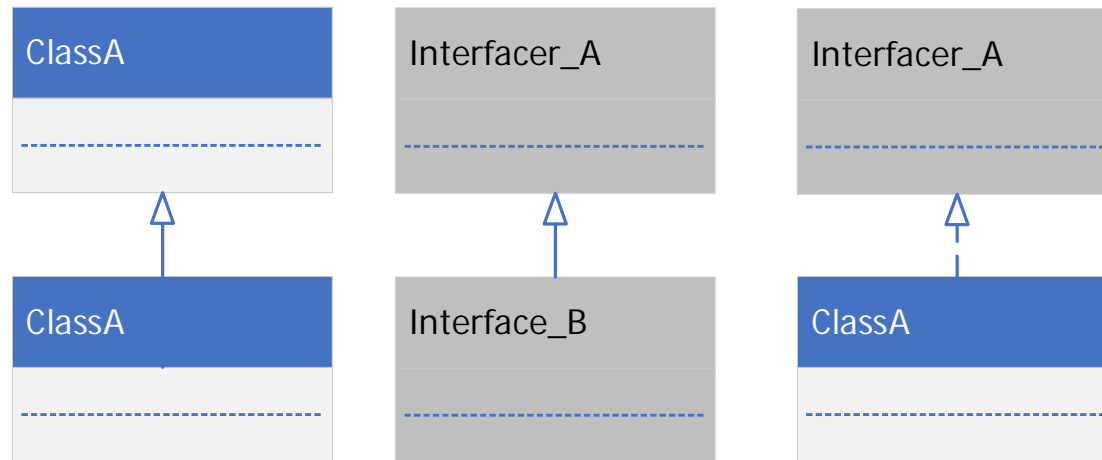
5.6.1 类的关系

类之间的最常见的关系有6种：继承关系、实现关系、依赖关系、关联关系、聚合关系、组合关系。在此基础上人们进一步归纳，将这些关系分为三类或四类。我们按照3大类进行归纳：

- 泛化关系（继承关系、实现关系）。
- 依赖关系（松散一些）
- 包含关系（关联关系、聚合关系、组合关系）。

1、泛化关系(Generalization):

泛化关系也称一般化关系，表示的是类之间的继承关系、接口之间的继承关系以及类和接口之间的实现关系。如图5-5所示。



2、依赖关系(Dependency):

也称**使用关系**，是一个类A中的方法使用到了另一个类B，这种关系非常弱。
一般而言，依赖关系在Java中体现为**局域变量、方法的形参，或者对静态方法的调用**。



```
abstract class Vehicle{
    public abstract void run(String city);
}
class MotorBike extends Vehicle{ //泛化关系
    public void run(String city){
        System.out.println("摩托车行驶: "+city);}
}
```

```
class Person{
    void travel(Vehicle car,String city){ //依赖关系。
        car.run(city);}
}
```

```
public class DepGenRel {  
    public static void main(String arg[]) {  
        Vehicle motor=new MotorBike();  
        Person p=new Person();  
        p.travel(motor, "北京- 南京"); //依赖调用  
    }  
}
```



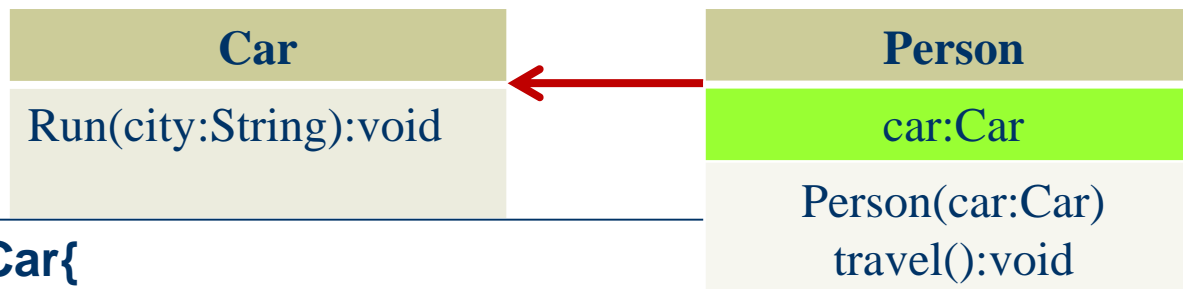
3、包含关系：

包含关系包括关联关系、聚合关系、组合关系，其中聚合关系和组合关系都是特殊的关联关系。

从代码上看，这三种子关系是一致的，都可设计成类与成员变量的包含关系。但在语义上有差别。

◆ 关联关系（比依赖关系更紧密）

通常体现为一个类使用另一个类的对象作为该类的**成员变量**



这里可表示，每个人都可拥有一辆车。

```
class Car{
void run(String city){
System.out.println("汽车开到 "+city);
}
}
class Person{
Car car;
Person(Car car){this.car=car;}
void travel(String city){
car.run(city);
}}
```

```
public class test{
public static void main(String arg[]){
Car car=new Car();
Person p=new Person(car);
p.travel("xuzhou");
}
}
```

◆ 聚合关系（关联关系的一种特例）

- 体现的是**整体与部分**的关系，通常表现为一个类（整体）是由多个其他类的对象作为该类的成员变量，此时整体与部分之间是**可以分离**的，具有各自的生命周期。



```
class Employee{
String name;
Employee(String name){
this.name=name; }
}
class Department{
Employee[] emps;
Department(Employee[] emps){
this.emps=emps;
}
void show(){
for(Employee emp:emps){
System.out.println(emp.name);
}
} }
```

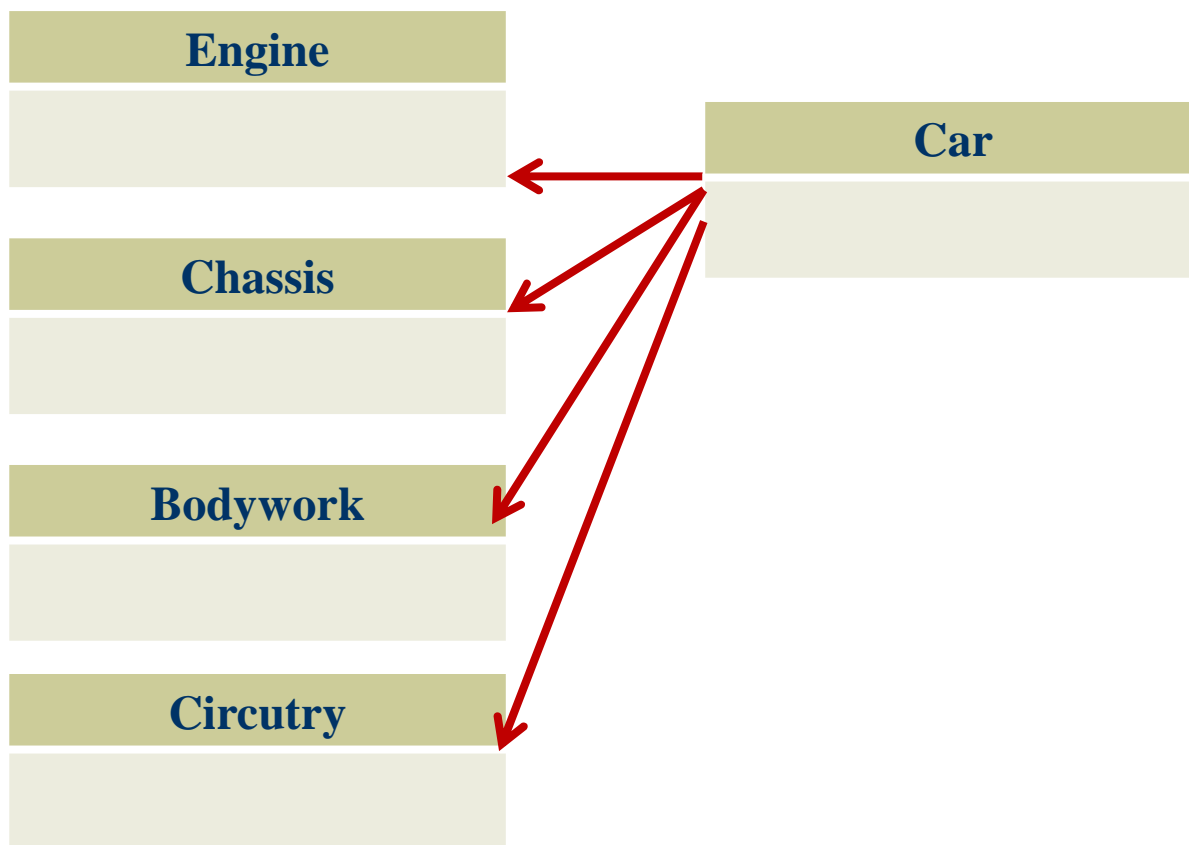
```
public class channelTest{
public static void main(String arg[]){
Employee[] emps={
new Employee("张三"),
new Employee("李四"),
new Employee("王五")};

Department dept=new Department(emps);
dept.show();
}
}
```

部门和员工聚合可理解为，部门由员工组成，同一个员工可属于多个部门，部门解散后员工依然存在。

- ◆ 组成关系（比聚合关系更高一层的关联关系）

- 体现的也是是整体与部分的关系，但整体与部分之间是不可分离的，具有各自的生命周期。



5.6.2 面向对象设计原则

1. 单一职责原则（Single Responsibility Principle）

每一个类应该专注于做一件事情。

2. 开闭原则（Open Close Principle）

面向扩展开放，面向修改关闭。（抽象）

3. 依赖倒置原则（Dependence Inversion Principle）

实现尽量依赖抽象，不依赖具体实现。（抽象）

4. 里氏替换原则（Liskov Substitution Principle）

超类存在的地方，子类是可以替换的。（抽象）

5. 接口隔离原则（Interface Segregation Principle）

应当为客户端提供尽可能小的单独的接口，而不是大的总的接口。

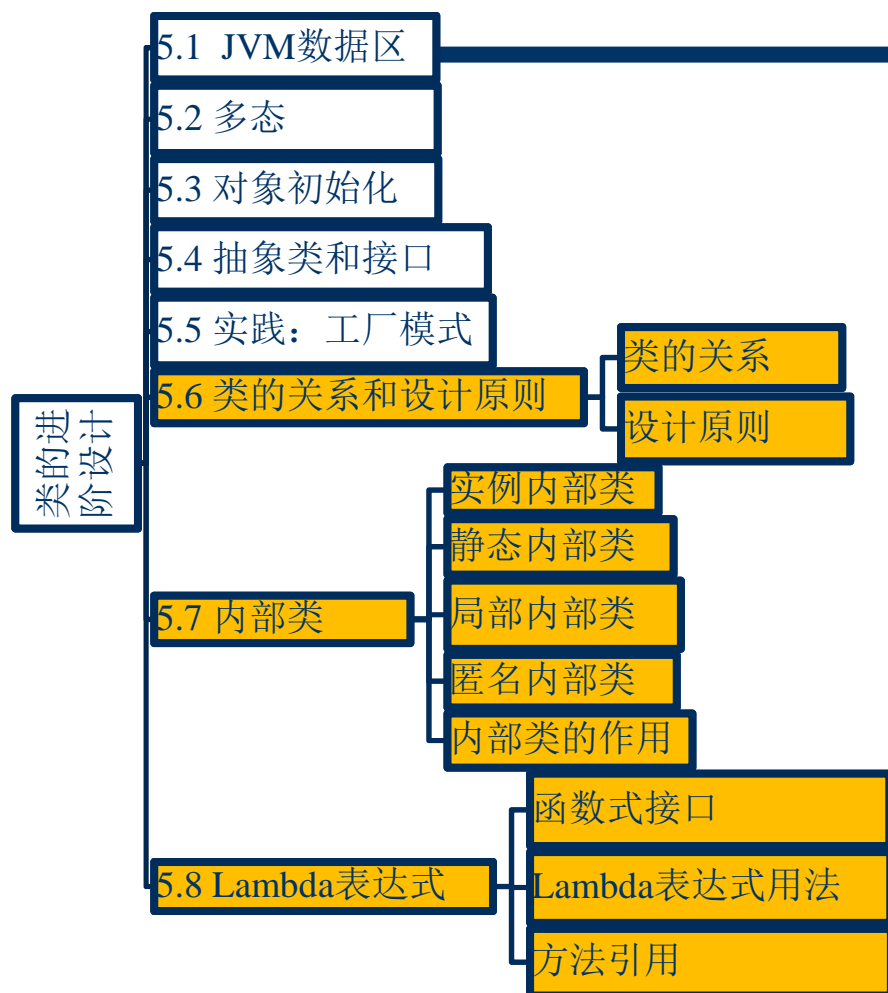
6. 迪米特法则（Law Of Demeter）

又叫最少知识原则，一个软件实体应当尽可能少的与其他实体发生相互作用（封装）。

7. 组合/聚合复用原则（Composite/Aggregate Reuse Principle CARP）

尽量使用合成/聚合达到复用，尽量少用继承。继承后父类的改变会影响子类。即使使用继承，继承层次一般不超过3层。

第五章：类的进阶设计



5.7 内部类

◆ 内部类

- 内部类是定义在一个类的类体中的类，它也可以包含变量和方法。包含内部类的类称为内部类的外嵌类。

◆ 引入内部类的原因：

- 内部类能够隐藏起来，不为同一包的其他类所访问
- 内部类可访问其所在的外部类的所有属性(包括private)，在回调方法处理中，内部类尤为便捷。

5.7 内部类

内部类分为实例内部类、静态嵌套类和局部内部类，每种内部类都有它的特点，内部类的特点如下：

- (1) 内部类仍是一个独立的类，在编译外部类时，内部类也会被编译成独立的class文件，文件名前面冠以外部类的类型和\$符号，如OuterClass\$InnerClass。
- (2) 内部类是外部类的成员，因此，内部类可以访问外部类的成员，无论是否为private。如果内部类声明成static，相应的，只能访问外部类的静态成员。
- (3) 内部类可作为外部类的成员，也可作为方法的成员（局部内部类）。如果作为外部类的成员，则可以使用4种访问权限修饰符，如果作为方法成员则没有访问权限修饰符。

5.7.1 实例 内部类

1、定义

实例内部类是指没有用 `static` 修饰的成员内部类。相当于实例成员。可以说，实例内部类仅存在于其外部类的对象中。需要先有外部类的对象，才能创建内部类的对象。

5.7.1 实例内部类

2、实例内部类的语法规则如下所示：

1) 在外部类的静态方法和外部类以外的其他类中，必须通过外部类的实例创建内部类的实例。其语法如下所示：

```
OutClass outer=new OutClass();
```

```
OutClass.InnerClass inObject=outer.new InnerClass();
```

2) 在外部类中不能直接访问内部类的实例成员，而必须通过内部类的实例去访问。
`inObject.Xxx`

3) 在实例内部类中不能定义 static 成员，除非同时使用 final 和 static 修饰。

4) 在实例内部类中，可以访问外部类的所有成员。

5) 在实例内部类中使用 this 关键字，其指的是内部类的当前对象，如果要表示外部类的当前对象，需要使用：外部类.this。

6) 内部类不能与外部类同名。

```
class Outer{
    private int a=100;
    static int b=200;
    int c=300;
    class Inner{                                //规则6: 不与外部类同名
        //static int sum=0; //规则3: 非法, 不能定义 static 成员
        final static int id=5; //规则3: 合法, 可以定义 final static 成员
        String name="";
        public String getOutInfo() {
            return "Outer: "+a+b+c; //规则4: 可以访问外部类的成员
        }
        public Outer getOuter() {
            return Outer.this; //规则5: 内部类中用外部类.this表外部类对象
        }
        public String toString() {
            return this.name+this.hashCode(); //规则5: this指当前内部类对象。
        }
    } //实例内部类定义结束
}
```

```
public void method1() {
```

```
    System.out.println(Inner.id); //规则2: 合法, 因为是内部类的 static数据  
    System.out.println(Inner.name); //规则2: 非法, 不用直接访问内部类实例数据  
    Inner i = new Inner(); // 规则2: 合法, 直接访问内部类型。  
    i.getOutInfo(); //规则2: 合法, 通过引用调用内部类方法  
    System.out.println(i.toString()); //规则2: 合法, 同上。  
}
```

```
public static Inner method2() {
```

```
    Inner i = new Outer().new Inner(); //规则1: 静态方法需要创建外部类实例  
    return i;  
}
```

```
} //外部类定义结束
```

```
public class InnerRule12 {
```

```
    //规则1: 其他外部类, 需要创建外部类实例  
    Outer.Inner i = new Outer().new Inner();  
}
```

5.7.2 静态内部类

□ 静态内部类是指用static修饰的内部类。例如：

```
class OuterClass{  
    static class InnerClass{  
    }  
}
```

作为静态成员，它与所属的**外部类而不是外部对象**相关联。在内部类不需要访问外围类的对象时，应该使用静态内部类（也称其为嵌套类）。

5.7.2 静态内部类

□ 静态内部类遵循如下规则：

(1)通过外部类的类名可直接访问静态内部类，所以，在创建静态内部类实例时，无需创建外部类的实例。如：

```
OutClass.InnerClass sic=new OutClass.InnerClass();
```

(2)静态内部类中可定义静态成员和实例成员。外部类以外的其他类可通过类名访问静态内部类中的静态成员，如果要访问静态内部类中的实例成员，则必须通过静态内部类的实例。

```
class Outer1{
    static class Inner{
        int dyM=0;           //规则2: 实例变量m
        static int StaN=0; //规则2: 静态变量n
    }
}

public class StaticInnerTest {
    public static void main(String arg[]) {
        Outer1.Inner oi=new Outer1.Inner();
        System.out.println(oi.dyM); //规则2: 访问静态类的实例变量
        System.out.println(Outer1.Inner.StaN); //规则2: 访问静态类的静态变量
    }
}
```

5.7.2 静态内部类

□ 静态内部类遵循如下规则：

(3)类似于类的静态方法，静态内部类可以直接访问外部类的静态成员，如果要访问外部类的实例成员，则需要通过外部类的实例去访问。例如：

```
class Outer1{  
    int a=0;  
    static int b=5;  
    static class Inner{  
        int M1=new Outer1().a; // 规则3: 静态内部类访问外部类的实例变量  
        int N2=b; //规则3: 访问静态类访问外部类的静态变量  
    }  
}
```

(4) 接口中可以定义内部类，且默认是static内部类，这种类可以被某个接口的所有不同实现所共用。

5.7.3 局部内部类(重点)


- ❑ 局部内部类是指定义在方法内的内部类。其有效范围只在定义它的方法内。
- ❑ 局部内部类遵循如下规则：
 - (1) 局部内部类与局部变量一样，不能使用访问修饰符和 `static` 修饰符。
 - (2) 在局部内部类中可以访问外部类的所有成员。
 - (3) 在局部内部类中只可以**读取而不能修改**当前方法中变量或常量。
 - (4) 如果方法中的成员与外部类成员同名，则可用
`OuterClass.this`访问外部类中的**实例成员** `OuterClass.this.MemberName`
或用**`OuterClass.MemberName`**访问外部类的**静态成员**。


```
class Outer2 {
    int a = 0;
    int d = 0;
    public void methodA() {
        int b = 0;
        final int d = 10;

        class Inner {
            int a2 = a;        // 规则2: 访问外部类中的成员a
            // int b2 = b;      // 非法, 访问方法中的非final量。
            int d2 = d;        // 规则3: 访问方法中的成员
            int d3 = Outer2.this.d; // 规则4: 访问外部类中的重名成员
        }
        Inner in=new Inner();
        System.out.println(in.a2+in.d2+in.d3);
    }
    public static void main(String[] args) {
        Outer2 ot = new Outer2();
        ot.methodA();
    }
}
```

5.7.4 匿名内部类

- ◆ 匿名类是指没有类名只有类体的内部类，如果程序定义某个类却只需要创建一个对象，这时可以考虑使用匿名内部类。
- ◆ 由于匿名类没有类名，所以创建匿名类的对象时需要用到该匿名类的父类或接口，而且匿名类的定义和对象创建是同时进行的，因此，在定义的同时，使用 `new` 语句来声明对象。
- ◆ 其语法形式如下：
`new 接口/父类([构造方法实参列表]) {
 // 类的主体 通常重写父类（父接口）所定义的方法
};`



例如：定义Person类的匿名子类，重写toString()方法，并生成一个对象。

```
new Person() {  
    public String toString() {  
        System.out.println("in AnonymousInnerClass");  
    }  
}
```

5.7.4 匿名内部类

匿名内部类的特点：

- (1) 匿名类和局部内部类一样，可以访问外部类的所有成员。其他局部内部类的特性也适用于匿名内部类。
- (2) 匿名类没有名字，所以不能定义构造方法，但可定义非静态字段，重写父类型方法。
- (3) 匿名内部类编译后对应的字节码文件名为：外部类\$数字序号（序号从1开始）。
- (4) 匿名类常用方式是向方法传参，当匿名内部类重写的父类（接口）只有一个方法时，建议使用Lambda表达式。详见5.10。

【例5.27】匿名内部类的用法。

```
interface superInterface{
    String str="in superInterface";
    void show() ;//抽象方法
}

abstract class superClass{
    static int sum=10;
    public superClass() {//无参构造方法
        System.out.println("in default constructor");
    }
    public superClass(int i) {//带参构造方法
        sum+=i;
        System.out.println("in constructor with arg");
    }
    public abstract void show() ; //抽象方法
}
```

```

class AnonyInnerTest {
    String info="in OutClass";
    void connect(superClass sc) {
        sc.show();
    }
    void connect(superInterface si) {
        si.show();
    }
    public void useConMethod() {
        (1) connect(new superInterface() { //通过接口，匿名内置类
            public void show() {
                System.out.println("AnonyInnerTest: "+info); //可访问外部类成员
                System.out.println("InnerClass: "+str); //可使用父接口的成员变量
            }
        });
        (2) connect(new superClass() { //利用父类的无参构造方法，创建匿名内置类
            public void show() {
                System.out.println("AnonyInnerTest: "+info); //可访问外部类成员
                System.out.println("InnerClass sum: "+sum); //可使用父接口成员
            }
        });
        (3) connect(new superClass(10) { //利用父类的带参构造方法，创建匿名内置类
            public void show() {
                System.out.println("AnonyInnerTest: "+info); //可访问外部类成员
                System.out.println("InnerClass sum: "+sum); //可使用父接口成员
            }
        });
    }
}

```

【运行结果】

```

AnonyInnerTest:in OutClass
innerClass:in superInterface
in default constructor
AnonyInnerTest:in OutClass
innerClass sum: 10
in constructor with arg
AnonyInnerTest:in OutClass
innerClass sum: 20

```

```
public static void main(String arg[]) {  
    AnonyInnerTest ai=new AnonyInnerTest();  
    ai.useConMethod();  
}  
}
```

【运行结果】

```
AnonyInnerTest:in OutClass  
innerClass:in superInterface  
in default constructor  
AnonyInnerTest:in OutClass  
innerClass sum: 10  
in constructor with arg  
AnonyInnerTest:in OutClass  
innerClass sum: 20
```

5.7.5 内部类的作用

- 无论外部类是否继承了父类，内部类都可以再继承一个类。从这个角度看，内部类使得多重继承的解决方案变得完整。接口解决了部分问题，而内部类有效地实现了“多重继承”。
- 另一方面，内部类被外部类包裹，自动拥有一个指向外部类或类对象的引用，在此作用域内，内部类有权操作外部类的成员，包括private成员。这样内部类就相当于闭包，即包含创建它的作用域A的信息的可调用对象B，这样一来，如果A调用B,而B又反过来调用于A，就能实现回调（callback）。

5.8 LAMBDA表达式

Lambda表达式本质上是一个匿名函数，它主要包括三部分：参数列表，箭头（->），以及一个表达式或语句块。

例如：

```
(int x, int y) -> {return x+y;}
```

作用：

Lambda表达式也是Java 8的重要新特性。它源自函数式（Functional Programming, FP）编程的思想，该思想的基本特性是将函数整体当做一种类型，并能以参数形式传递给其他函数，或作为其他函数的输出。

5.8.1 函数式接口

- Java是一种面向对象语言，在Java8以前，Java是不能直接传递代码段的，因为方法必须被封装在类里，不能单独存在。
- 在Java8里，Lambda类型是一种函数式接口。如果一个接口中有且只有一个抽象的方法，那这个接口就可称为函数式接口，可以（但不强求）用注解@FunctionalInterface来表示。
- Lambda表达式本身是以一种简化的语法重写了接口中的唯一方法的匿名类。

/**正确，只有一个抽象方法*/

@FunctionalInterface

```
interface FunctionInterface {  
    void show();  
}
```

/**正确，只有一个抽象方法*/

@FunctionalInterface

```
interface FunctionInterface1 {  
    void show();  
    default long cube(int n) {return n*n;}  
    static void print() {  
        System.out.println("in FunctionInterface.print()");  
    }  
}
```

/**错误，有两个抽象方法*/

@FunctionalInterface

```
interface FunctionInterface2{  
    void show();  
    String getInfo();  
}
```

5.8.2 Lambda表达式用法

1、（形参列表）->表达式 或者 {代码块;}

说明如下：

- ① 形参列表对应于被重写的抽象方法的形参表。
- ② 形参类型可选：参数类型可以明确声明，也可以由上下文自动推断。
- ③ 形参圆括号可选：当参数只有一个时可省略圆括号，没有或多个参数需要使用圆括号。
- ④ “表达式”或“代码块”对应于重写方法的方法体。如果主体包含一个语句，可以省略大括号。
- ⑤ 返回关键字return可选：如果主体只有一个表达式返回值，则编译器会自动返回值；如果有大括号，需要指定表达式返回了一个数值。

```
(int x, int y) -> x+y //规则2：明确声明参数类型，返回一个表达式的值
(x, y) -> x+y; // 规则2：由JVM自动推断参数类型，返回一个表达式的值
x -> x*2; // 规则3：只有一个参数，圆括号可省略。
() -> 10; //规则3：没有参数，圆括号不能省略，返回一个值。
(x, y) -> {return x+y;} //规则5：有大括号，需要返回值，要用return语句
```

5.8.2 Lambda表达式用法

@FunctionalInterface

```
interface InterfaceA{//函数式接口  
    int calc(int m, int n);  
}
```

```
public class LambdaDemo{  
    public static void main(String arg[]){  
        InterfaceA a=(m, n) ->m*n ; //Lambda表达式  
        System.out.println(a.calc(1, 2)); //使用  
    }  
}
```

5.8.2 Lambda表达式用法

2、使用Lambda表达式

Lambda 表达式一种常见的用途就是作为参数传递给方法，这需要方法的形参类型声明为函数式接口类型。

例如：用lambda表达式定义算数运算操作。

```

interface MathOperator{
    int operation(int x, int y);
}
public class UseLambdaDemo {
    private static int execute(int a, int b, MathOperator mo){
        return mo.operation(a, b);
    }
    public static void main(String arg[]) {
        MathOperator add=(int a, int b)->a+b; //参数类型声明
        MathOperator sub=(a, b)->a-b; //参数不声明类型
        MathOperator mul=(a, b)->{return a*b;};
        //MathOperator div=(a, b)->a/b;

```

//调用Lambda表达式

```

int re1=execute(40, 20, add);
int re2=execute(40, 20, sub);
int re3=execute(40, 20, mul);
int re4=execute(40, 20, div);
int re5=execute(40, 20, (a, b)->a/b);
System.out.printf("a+b=%d, a-b=%d, ", re1, re2);
System.out.printf("a*b=%d, a/b=%d", re3, re4);
} }

```

【运行结果】

a+b=60,a-b=20,a*b=800,a/b=2

5.8.2 Lambda表达式用法

3、变量作用域

Lambda表达式的设计初衷之一就是用来代替匿名内部类。他们之间有相似也有区别。

- (1) 相同：可以访问但不能修改其所在方法声明的局部变量。
- (2) 区别：匿名内置类会被编译成独立的类字节码文件，但Lambda表达式会被编译为类的私有方法，所以在Lambda表达式中出现的this表示表达式所在类的当前对象，而在匿名内置类中this表示匿名内部类本身的对象。


```
class LambdaFinalTest {
    static String first = "Hello! ";
    interface Greeting {
        void sayMessage(String message);}

    private void test() {
        int id=10; //方法的局部变量
        Greeting greet1 = message ->{
            (1)//id=id+1; //非法,
            (2)first="Bye! "; //合法
            (3)System.out.println(this.toString()); //合法
            (4)System.out.println(first + message+id); //合法
        };
        greet1.sayMessage("java");
    }
    public static void main(String args[]){
        LambdaFinalTest lf=new LambdaFinalTest();
        lf.test();
    }
}
```

【运行结果】

LambdaFinalTest@1e643faf
Bye! java10

5.8.3 方法引用

- Java 8 之后增加了双冒号 “::” 运算符，该运算符用于“方法引用”。方法引用可以理解为Lambda表达式的一种更加的简洁形式。
- Lambda表达式的方法体如果仅包含一条方法调用语句，此时可以使用方法引用。语法格式如下：

类名/对象名::方法名//只有方法名，没有参数，参数通过函数接口方法推断

引用方法	语法	示例	等价的Lambda表达式
静态方法	类名::静态方法	Integer::valueOf	(str,ra)->Integer.valueOf(str,ra)
实例方法	对象名::实例方法	stra::compareTo	strb->stra.compareTo(strb)
实例方法	类名::实例方法	String::compareTo	(stra,strb)->stra.compareTo(strb)
构造方法	类名::new	String::new	str->new String(str)

【扩展阅读，课堂介绍】

例：方法引用示例。

- 1、定义一个学生类Student，
- 2、该类包括name和age两个私有成员变量，
- 3、同时定义了以下成员方法：构造方法、name的读写方法、age的读写方法、比较学生的方法。
- 4、主程序的功能是对学生进行排序。

```
import java.util.Arrays;
```

```
/**
```

```
 * 下列代码中使用数组存储学生对象，并调用java.util.Arrays中的sort  
方法进行排序，
```

```
 * sort(T[] a, Comparator<? super T> c) 方法接收一个Comparator  
函数式接口，接口唯一的抽象方法compare接收两个参数，
```

```
 * 返回一个int型的比较结果。下面是Comparator接口的定义（该接口定义  
用到了泛型，请参* 看第八章）
```

```
 * @FunctionalInterface
```

```
    public interface Comparator<T>{
```

```
        int compare(T o1, T o2);
```

```
    }
```

```
 * */
```

```
//定义Student
class Student{
    private String name; //姓名
    private int age; //年龄
//带参构造方法
    public Student(String name, int age) {
        this.name=name;
        this.age=age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public static int compareByAge(Student st1, Student st2) {
//已存在的比较法
        return st1.getAge()-st2.getAge();
    }
    public int compareByAge2(Student s2) { //已存在的比较法
        return this.getAge()-s2.getAge();
    }
} //Student定义结束
```



//比较接口

```
interface Comparable{  
    int compare(String s);  
}
```

//学生工厂接口

```
interface StudentFactory{  
    Student create(String name, int age);  
}
```

```

public class LambdaMethodRef {
    public static void main(String arg[]) {
        Student[] st=new Student[4];
        String sa="wangwu";
        //等价于 (name, age) ->new Student(name, age), 实现具体工厂类
        StudentFactory sf=Student::new;
        st[0]=sf.create("zhangsan", 50);
        st[1]=sf.create("lisi", 40);
        st[2]=sf.create("wangwu", 30);
        st[3]=sf.create("zhaoliu", 60);
    }
}

```

【运行结果】

```

wangwu , 30
lisi , 40
zhangsan, 50
zhaoliu , 60
*****
zhangsan, 50
zhaoliu , 60

```

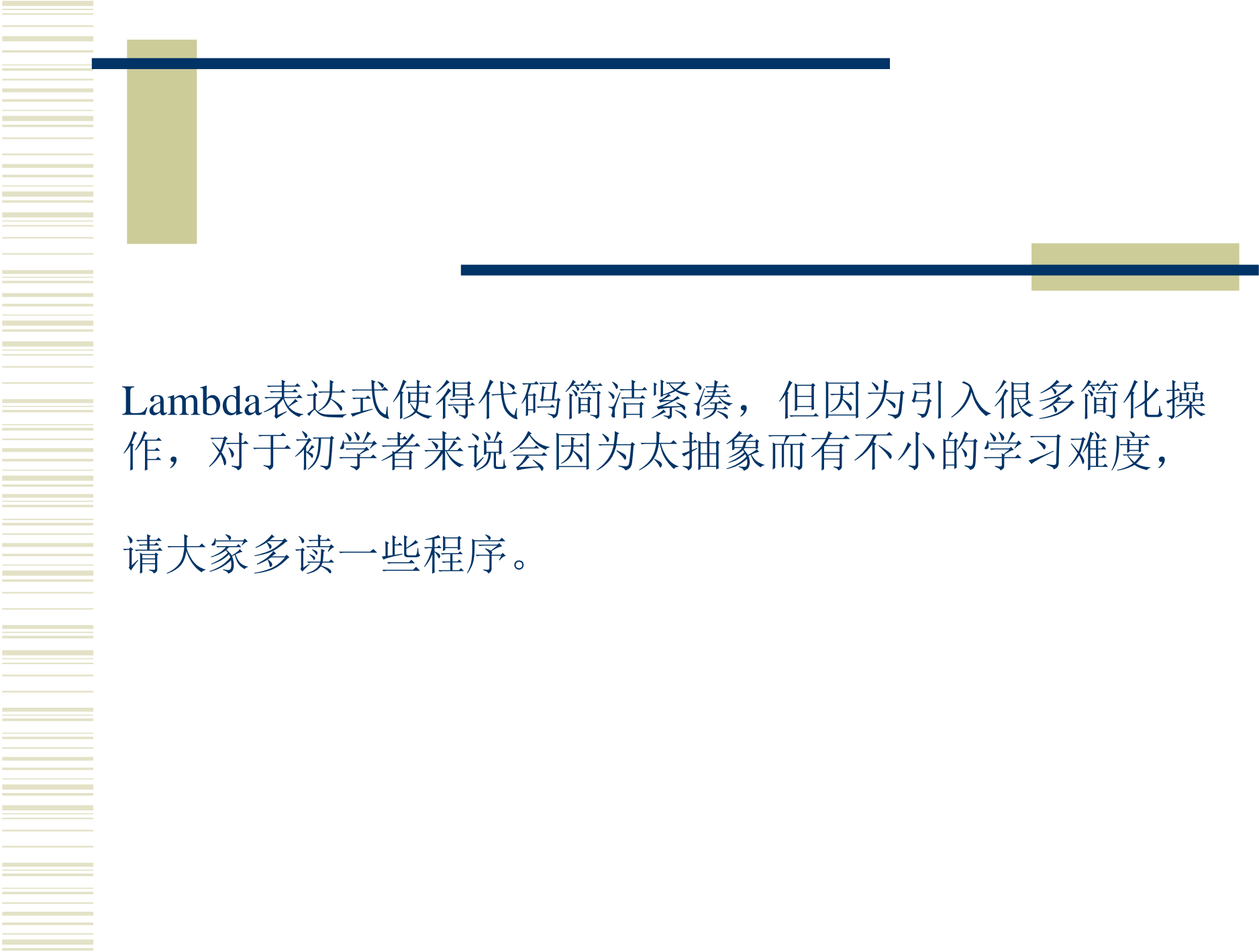
```

// 类名引用静态方法, 等价于lambda表达式 (s1, s2) ->Student.compareTo(s1, s2));
Arrays.sort(st, Student::compareTo);

//类名引用实例方法, 等价于用lambda表达式表示: (s1, s2) ->s1.compareTo(s2));
Arrays.sort(st, Student::compareTo);

for(Student s:st) //打印排序后的学生信息
    System.out.printf("%-8s, %d\n", s.getName(), s.getAge());
System.out.println("*****");
//对象引用String类型的实例方法compareTo(String), 等价于sb->sa.compareTo(sb)
Comparable cp=sa::compareTo;
//名字按字典比较, 比"wangwu"大的学生的信息会被打印出来。
for(Student s:st) {
    if(cp.compare(s.getName())<0)
        System.out.printf("%-8s, %d\n", s.getName(), s.getAge());
}
}

```



Lambda表达式使得代码简洁紧凑，但因为引入很多简化操作，对于初学者来说会因为太抽象而有不小的学习难度，
请大家多读一些程序。