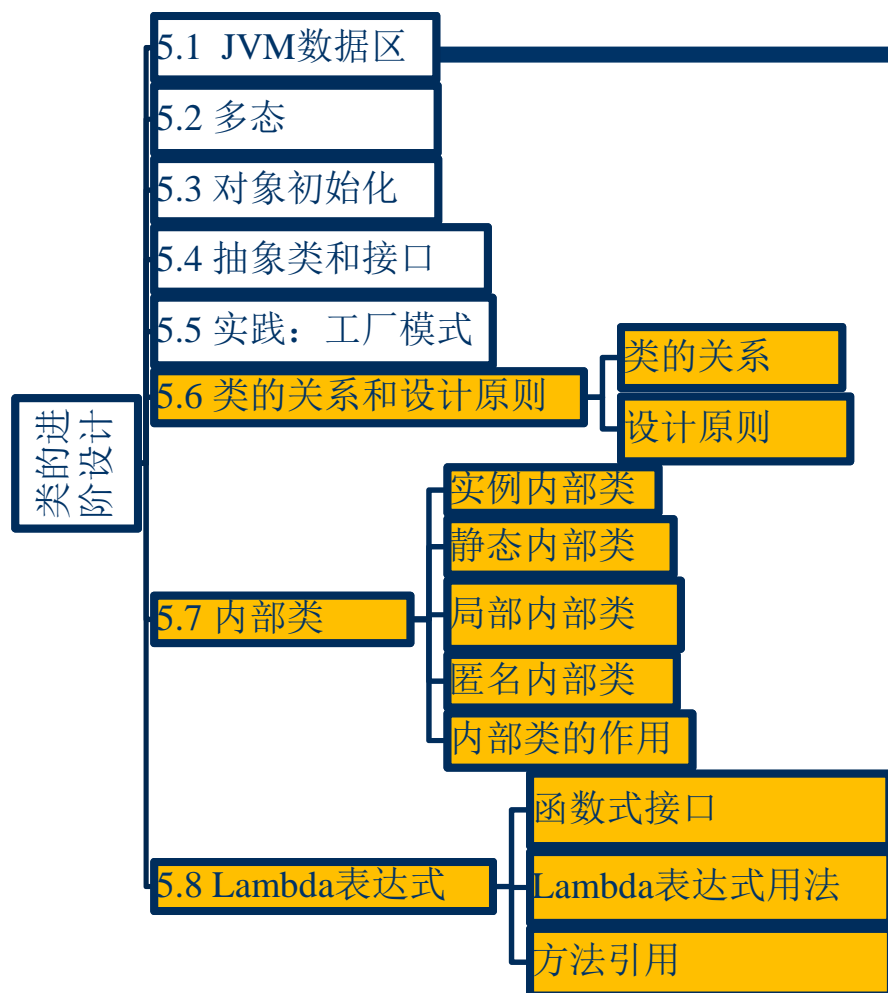




第五章 类的进阶设计（2）



第五章：类的进阶设计



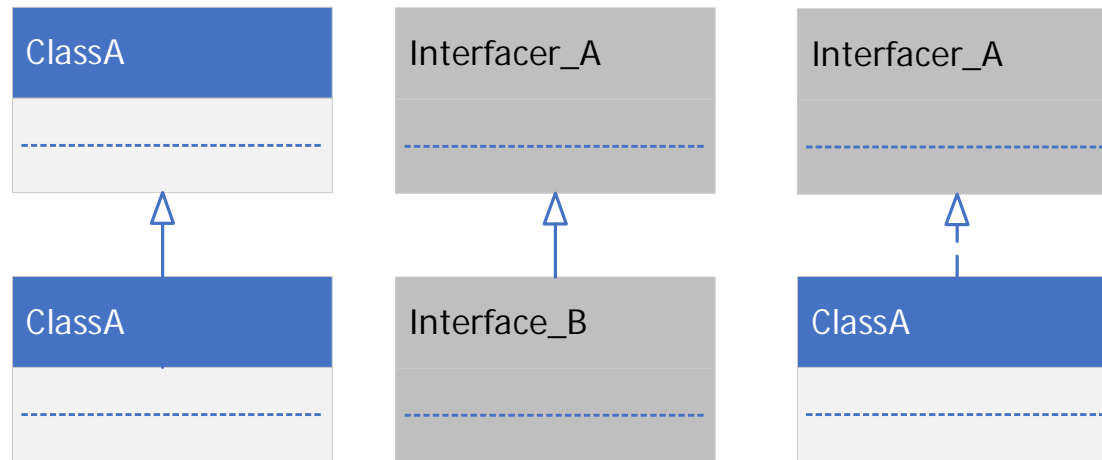
5.6.1 类的关系

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  
    }  
}
```



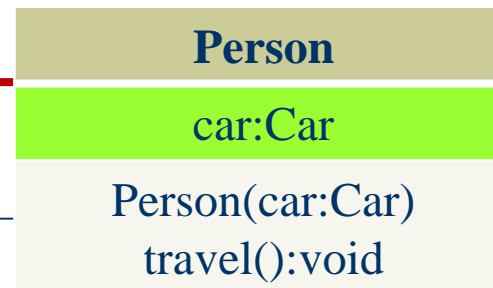
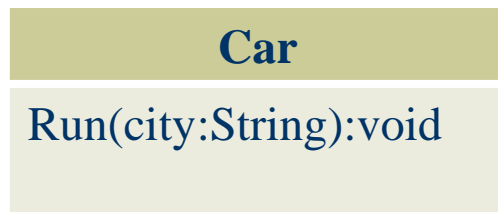
3、包含关系：

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);
}
} }
```

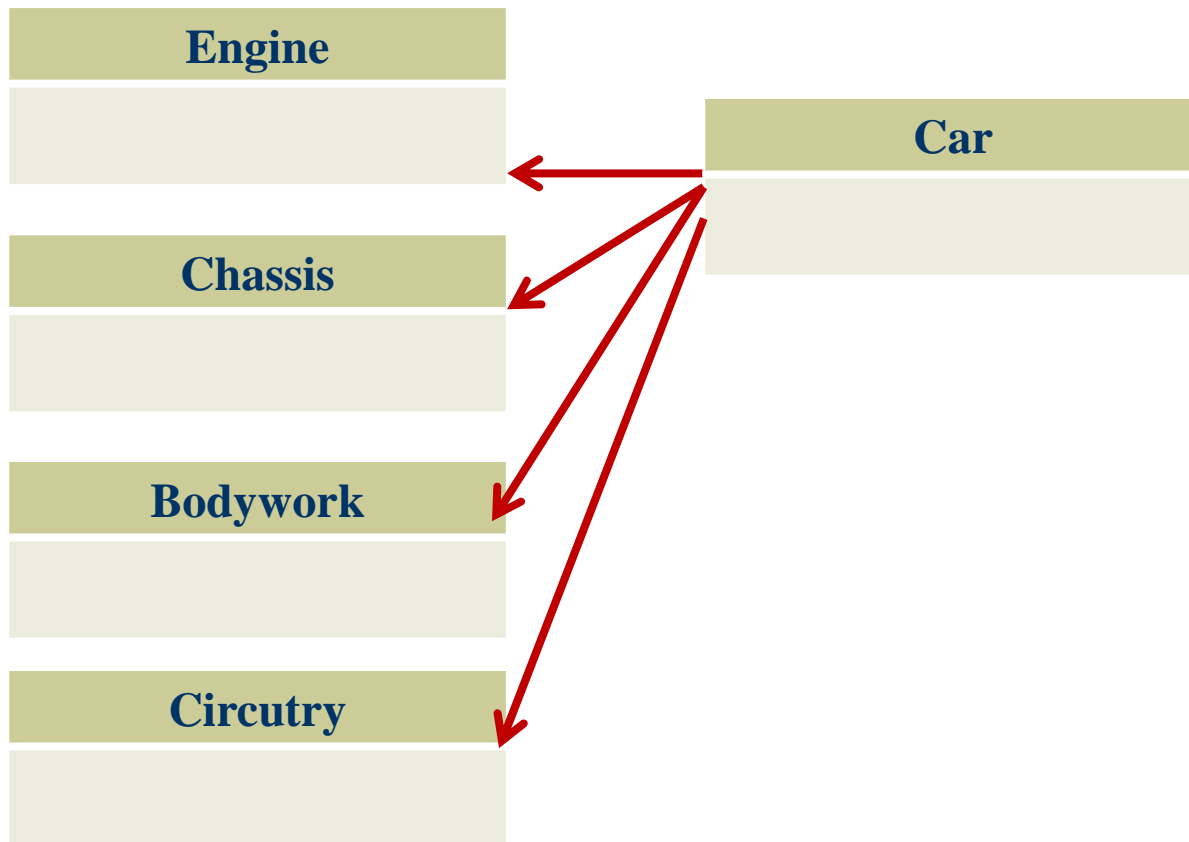
```
public class channelTest{
public static void main(String arg[]){
Employee[] emps={
new Employee("张三"),
new Employee("李四"),
new Employee("王五")};

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。

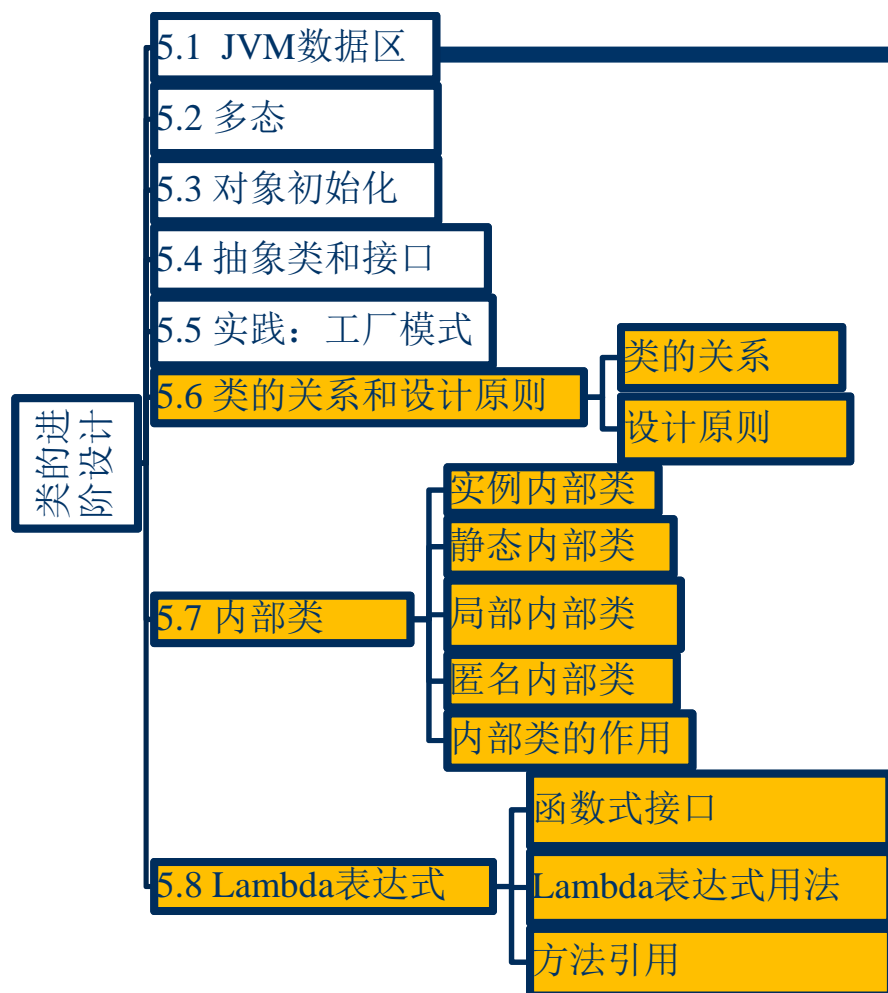
6. 迪米特法则（Law Of Demeter）

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

```


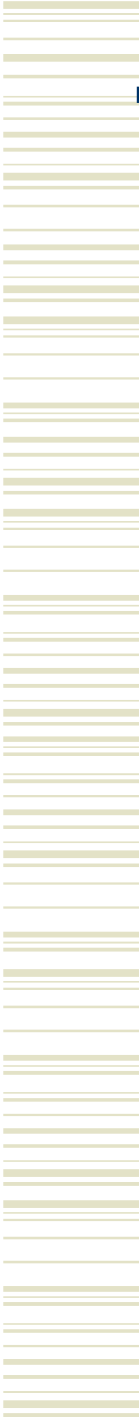
```

// 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。