一、包

1.0 概念

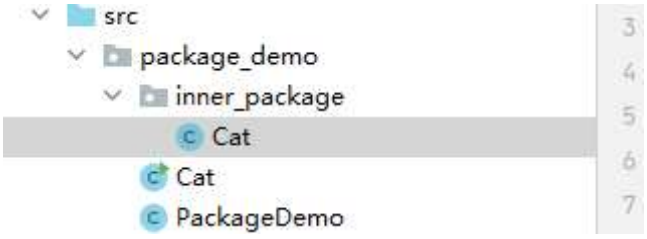
1. 包是用来分门别类的管理各种不同类的、建包利于程序的管理和维护
2. 建包的语法格式：package 公司域名倒写.技术名称。报名建议全部英文小写，且具备意义
3. 建包语句必须在第一行，一般IDEA工具会帮助创建

```
        package com.itheima.javabean;//建包语句
        public class Student {

        }
  //带包编译某个目录下的全部java文件
 注意：>javac -d . *.java
```

2.0 导包

相同包下的类可以直接访问，不同包下的 类必须导包,才可以使用！导包格式：import 包名.类名;
假如一个类中需要用到不同类，而这个【两个类的名称】是一样的，那么默认只能导入一个类，另一个类要带包名访问。




```
package package_demo;

/**
 * ClassName: Cat <br/>
 * Description: <br/>
 * date: 2022/3/7 8:53<br/>
 *
 * @author fgcy<br />
 * @since JDK 1.8
 */
public class Cat {
    public void run() {
        System.out.println("外猫跑");
    }

    public static void main(String[] args) {
        final package_demo.inner_package.Cat cat = new
package_demo.inner_package.Cat();//一个包下的包与该包属于不同的包，所以要导包（当一个类中出现两个同名的类时，一个可以倒包，一个需要用全限名访问）
        final Cat cat1 = new Cat();
        cat.run();
        cat1.run();
    }
}
```

```
--------------------------------------------------------------------------------
------------------------------------------
    package package_demo.inner_package;

/**
 * ClassName: Cat <br/>
 * Description: <br/>
 * date: 2022/3/7 8:53<br/>
 *
 * @author fgcy<br />
 * @since JDK 1.8
 */
public class Cat {
    public void run(){
        System.out.println("内猫跑");
    }
}


内猫跑
外猫跑
```

三、权限修饰符

1.0 什么是权限修饰符?

什么是权限修饰符?

权限修饰符：是用来控制一个成员【能够被访问】的范围的。

可以修饰成员变量，方法，构造器，内部类(没有代码段)，不同权限修饰符修饰的成员能够被访问的范围将受到限制。

2.0 权限修饰符的分类和具体作用范围:

权限修饰符：有四种作用范围由小到大（private -> 缺省 -> protected - > public ）
private:本类中
缺省:包权限
protected:包权限+子类
public:所有的类都可以

修饰符	同一 个类中	同一个包中 其他类	不同包下的 子类	不同包下的 无关类
private	√			
缺省	√	√		
protected	√	√	√	
public	√	√	√	√

四、final关键字

1.0 概念

final的作用
final 关键字是最终的意思，可以修饰（方法，变量，类）
修饰方法：表明该方法是最终方法，不能被重写。
修饰变量：表示该变量第一次赋值后，不能再次被赋值(有且仅能被赋值一次)。[所有变量，局部、成员]，基本数据类型数值不能改变，引用数据类型地址不能改变（但对象 内容可以改变）
修饰类：表明该类是最终类，不能被继承（java.lang.String）

1. public static final修饰的变量称为常量，需要在定义的时候赋值；或者在静态代码块中赋值（静态代码块本来就是用于初始化静态成员变量）
2. 用final修饰实例成员变量，此时该实例成员变量必须在定义的时候赋值（没有意义）
3. 用final修饰的局部成员变量可以不在定义的时候赋初始值

五、常量

1.0 常量概述和基本作用

1. 常量是使用了public static final修饰的成员变量，必须有初始化值，而且执行的过程中其值不能被改变。
2. 常量的作用和好处：可以用于做系统的配置信息，方便程序的维护，同时也能提高可读性
3. 常量命名规范：英文单词全部大写，多个单词下划线连接起来。

常量的执行原理
在编译阶段会进行“宏替换”，把使用常量的地方全部替换成真实的字面量。
这样做的好处是让使用常量的程序的执行性能与直接使用字面量是一样的。

2.0 常量做信息标志和分类

选择常量做信息标志和分类：
代码可读性好，实现了软编码形式

```
package constant_demo;

import com.sun.org.apache.bcel.internal.generic.BREAKPOINT;

import javax.swing.*;
import java.awt.event.ActionEvent;

/**
 * ClassName: ConstantDemo1 <br/>
 * Description: <br/>
 * date: 2022/3/7 9:55<br/>
 *
 * @author fgcy<br />
 * @since JDK 1.8
 */
public class ConstantDemo1 {

    public static final int UP = 1;//使用常量作为信息分类的标志
    public static final int DOWN = 2;
    public static final int LEFT = 3;
```

```
    public static final int RIGHT = 4;



    public static void main(String[] args) {
        final JFrame win = new JFrame();
        final JPanel panel = new JPanel();

        final JButton btn1 = new JButton("上");
        final JButton btn2 = new JButton("下");
        final JButton btn3 = new JButton("左");
        final JButton btn4 = new JButton("右");

        btn1.addActionListener(new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                moveTo(UP);
            }
        });
        btn2.addActionListener(new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                moveTo(DOWN);
            }
        });
        btn3.addActionListener(new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                moveTo(LEFT);
            }
        });
        btn4.addActionListener(new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                moveTo(RIGHT);
            }
        });

        panel.add(btn1);
        panel.add(btn2);
        panel.add(btn3);
        panel.add(btn4);
        win.add(panel);
        win.setLocationRelativeTo(null);
        win.setSize(300, 400);
        win.setVisible(true);
    }

    public static void moveTo(int flag) {
        switch (flag) {
            case UP:
                System.out.println("UP");
                break;
            case DOWN:
                System.out.println("DOWN");
                break;
            case LEFT:
                System.out.println("LEFT");
```

```
                break;
            case RIGHT:
                System.out.println("RIGHT");
                break;
        }
    }
}
```

六、枚举

1.0 枚举的概述

枚举是Java中的一种特殊 类型（特殊的类）
枚举是多例模式（构造器是私有的，不能对外创建对象，且有多个实例；第一行罗列的名称就是实例的名字）
枚举的作用："是为了做信息的标志和信息的分类"。

2.0 枚举定义的格式

```
 修饰符 enum 枚举名称{
    第一行都是罗列枚举类实例的名称。
}
```

- **java源文件**

```
//java源文件
enum Season{
    SPRING , SUMMER , AUTUMN , WINTER;
}
```

- **反编译文件**

```
//将java源文件编译成字节码文件，再通过javap命令反编译，如下
Compiled from "Season.java"
    public final class Season extends java.lang.Enum<Season> {
        public static final Season SPRING = new Season();
        public static final Season SUMMER = new Season();
        public static final Season AUTUMN = new Season();
        public static final Season WINTER = new Season();
        public static Season[] values();
        public static Season valueOf(java.lang.String);
    }
```

- 伪代码

```
public class Season extends Enum{
        public static final Season SPRING;
        public static final Season SUMMER;
        public static final Season AUTUMN;
        public static final Season WINTER;
    public Season(String name,int ori){
        super(name,ori);
```

```
    }
    static{
          SPRING=new Season("SPRING",1);
          SPRING=new Season("SUMMER",2);
          SPRING=new Season("AUTUMN",3);
          SPRING=new Season("WINTER",4);
    }
}
```

3.0枚举的特征

1. 枚举类都是继承了枚举类型：java.lang.Enum
2. 枚举都是最终类，不可以被继承。
3. 构造器都是私有的，枚举对外不能创建对象。
4. 枚举类的第一行默认都是罗列枚举对象的名称的。
5. 枚举类相当于是多例模式

4.0 使用枚举作为信息标志与分类

选择常量做信息标志和分类：
    虽然可以实现可读性，但是入参值不受约束，代码相对不够严谨
枚举做信息标志和分类：
    代码可读性好，入参约束严谨，代码优雅，是最好的信息分类技术！建议使用

```
package constant_demo;

import com.sun.org.apache.bcel.internal.generic.BREAKPOINT;

import javax.swing.*;
import java.awt.event.ActionEvent;

/**
 * ClassName: ConstantDemo1 <br/>
 * Description: <br/>
 * date: 2022/3/7 9:55<br/>
 *
 * @author fgcy<br />
 * @since JDK 1.8
 */
public class ConstantDemo1 {

    public static void main(String[] args) {
        final JFrame win = new JFrame();
        final JPanel panel = new JPanel();

        final JButton btn1 = new JButton("上");
        final JButton btn2 = new JButton("下");
        final JButton btn3 = new JButton("左");
        final JButton btn4 = new JButton("右");

        btn1.addActionListener(new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                moveTo(Direction.UP);//之前使用常量作为信息的标志与分类，不严谨；因为只要满足方法的参数列表类型的参数就可以填入，容易产生问题
```

```
            }
        });
        btn2.addActionListener(new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                moveTo(Direction.DOWN);
            }
        });
        btn3.addActionListener(new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                moveTo(Direction.LEFT);
            }
        });
        btn4.addActionListener(new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                moveTo(Direction.RIGHT);
            }
        });

        panel.add(btn1);
        panel.add(btn2);
        panel.add(btn3);
        panel.add(btn4);
        win.add(panel);
        win.setLocationRelativeTo(null);
        win.setSize(300, 400);
        win.setVisible(true);
    }

    public static void moveTo(Direction direction) {
        switch (direction) {
            case UP://switch兼容枚举类型，可以直接不带类名，
                System.out.println("UP");
                break;
            case DOWN:
                System.out.println("DOWN");
                break;
            case LEFT:
                System.out.println("LEFT");
                break;
            case RIGHT:
                System.out.println("RIGHT");
                break;
        }
    }
}
--------------------------------------------------------------------------------
----------------------------------------
package constant_demo;

/**
 * ClassName: Direction <br/>
 * Description: <br/>
 * date: 2022/3/7 10:30<br/>
 *
 * @author fgcy<br />
```

```
 * @since JDK 1.8
 */
public enum Direction {
    UP, DOWN, LEFT, RIGHT;
}
```

七、 抽象类

1.0 抽象类基本概述

某个父类知道所有子类都要完成的功能，但是每个子类完成的情况又不一样，父类就只是定义该功能的基本要求，具体由子类实现，这个类就是抽象类；抽象类其实是一种不完全的设计图

- 抽象类定义格式

```
修饰符 abstract class 类名{}

必须用abstract修饰

如果该类中有抽象方法，那么该类一定要定义成抽象类
```

- 抽象方法

```
修饰符 abstract 返回值类型 方法名称(形参列表);
    没有方法体，只有方法签名，必须用abstract修饰
    抽象类的子类必须要实现父类的所有抽象方法，否则该子类也要申明成抽象类
```

```
package abstract_demo;

/**
 * ClassName: Cards
 * Description:父类
 * date:2022/3/7
 *
 * @author fgcy
 * @since JDK 1.8
 */

public abstract class Cards {
    private String name;
    private double money;


    public abstract void pay(double money);

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public double getMoney() {
        return money;
    }

    public void setMoney(double money) {
        this.money = money;
    }
}
-----------------------------------------------------------------------------------
----------------------------------
    package abstract_demo;

/**
 * ClassName: GoldCard
 * Description:子类
 * date:2022/3/7
 *
 * @author fgcy
 * @since JDK 1.8
 */

public class GoldCard extends Cards {
    @Override
    public void pay(double money) {
        double rs = getMoney() - money * 0.8;
        System.out.println(getName() + "当前账户余额：" + getMoney() + "本次支付" +
money * 0.8 + "账户剩余:" + rs);
        setMoney(rs);
    }


    public static void main(String[] args) {
        final GoldCard goldCard = new GoldCard();
        goldCard.setMoney(1000);
        goldCard.setName("swx");
        goldCard.pay(100);
    }
}
```

2.0 特征和注意事项

1. 有得有失 得到了抽象方法，失去了创建对象的能力。

2. 抽象类为什么不能创建对象？**
   因为抽象类中可能存在抽象方法，抽象方法没有方法体，不能实现功能；
3. 类有的成员（成员变量、方法、构造器）抽象类都具备

4. 抽象类中不一定有抽象方法，有抽象方法的类一定是抽象类

5. 一个类继承了抽象类必须重写完抽象类的全部抽象方法，否则这个类也必须定义成抽象类。

6. 不能用abstract修饰变量、代码块、构造器（abstract只能修饰类和方法）

7. 抽象类必须有无参构造，因为抽象类必须有子类继承并实现抽象方法，子类实例自身时，需要先获取父类的属性、方法

3.0 final和abstract是什么关系?

互斥关系

abstract定义的抽象类作为模板让子类继承，final定义的类不能被继承。

抽象方法定义通用功能让子类重写，final定义的方法子类不能重写

4.0 抽象类的应用知识：模板方法模式（设计模式）

使用场景说明：当系统中出现同一个功能多处在开发，而该功能中大部分代码是一样的，只有其中部分可能不同的时候。

模板方法模式实现步骤

把功能定义成一个所谓的模板方法，放在抽象类中，模板方法中只定义通用且能确定的代码。

模板方法中不能决定的功能定义成抽象方法让具体子类去实现

```
package abstract_demo;

/**
 * ClassName: Cards
 * Description:父类
 * date:2022/3/7
 *
 * @author fgcy
 * @since JDK 1.8
 */

public abstract class Cards {
    private String name;
    private String password;
    private double money;

    /**
     * ClassName: Cards
     * Description:模板方法
     * date:2022/3/7
     *
     * @author fgcy
     * @since JDK 1.8
     */

    public final void handle(double money) {//模板方法是给子类直接使用的，不是让子类重写的，一旦子类重写了模板方法就失效了。

        if ("admin".equals(name) && "password".equals(password)) {
            System.out.println("登录验证成功");

            pay(money);//抽象方法，由子类具体实现

            System.out.println("祝消费愉快");
        } else {
            System.out.println("登录验证失败");
        }

    }
```

```
    public abstract void pay(double money);

    public Cards() {
    }

    public Cards(String name, String password) {
        this.name = name;
        this.password = password;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getMoney() {
        return money;
    }

    public void setMoney(double money) {
        this.money = money;
    }
}
--------------------------------------------------------------------------------
--------------
  package abstract_demo;

/**
 * ClassName: GoldCard
 * Description:子类
 * date:2022/3/7
 *
 * @author fgcy
 * @since JDK 1.8
 */

public class GoldCard extends Cards {
    public GoldCard() {
    }

    public GoldCard(String name, String password) {
        super(name, password);//直接把值传给父类构造器
    }

    @Override
    public void pay(double money) {
        double rs = getMoney() - money * 0.8;
        System.out.println(getName() + "当前账户余额: " + getMoney() + "本次支付" +
money * 0.8 + "账户剩余:" + rs);
        setMoney(rs);
    }


    public static void main(String[] args) {
        final GoldCard goldCard = new GoldCard("admin", "password");
```

```
        goldCard.handle(10000);//先是跑父类的模板方法，由于模板方法中存在抽象方法，所以此时跑的是子类实现的抽象方法，再回到模板方法
    }
}
登录验证成功
admin当前账户余额：0.0本次支付8000.0账户剩余:-8000.0
祝消费愉快
```

模板方法我们是建议使用final修饰的，这样会更专业，那么为什么呢？
模板方法是给子类直接使用的，不是让子类重写的，
一旦子类重写了模板方法就失效了。

模板方法模式解决了什么问题？
极大的提高了代码的复用性
模板方法已经定义了通用结构，通用功能已经定义好了，差异化功能由子类具体实现；模板不能确定的定义成抽象方法。
使用者只需要关心自己需要实现的功能即可

八、接口

1.0 什么是接口

是一种规范

2.0 接口格式

接口用关键字interface来定义

```
public interface 接口名 {
        // 常量 (public static final )
        // 抽象方法(public abstract)
}
```

3.0 注意

1. JDK8之前接口中只能是抽象方法和常量，没有其他成分了。
2. 接口不能实例化。
3. 接口中的成员都是public修饰的，写不写都是，因为规范的目的是为了公开化。
4. 接口是更加具体的抽象，成员方法都是abstract修饰的

```
package inteface_demo;

/**
 * ClassName: User <br/>
 * Description: <br/>
 * date: 2022/3/8 7:48<br/>
 *
 * @author fgcy<br />
 * @since JDK 1.8
 */
public interface User {
    //    成员变量全是常量
```

```
//    public static final String NAME = "苏未晓";
    String NAME = "苏未晓";//接口成员变量都是常量，所以前面的修饰可以省略不写

    //    public abstract void test(final String name);
    void test(final String name);//接口是一种约束别人的规范，所以是方法是公开抽象的可以省略public abstract
}
```

4.0 接口与类的关系（多实现）

1. 接口是用来被类实现（implements）的，实现接口的类称为实现类。实现类可以理解成所谓的子类。

2. 格式：
    修饰符 class 实现类 implements 接口1, 接口2, 接口3 , ... {

    }

3. 实现的关键字：implements
4. 接口可以被类单实现，也可以被类多实现。
5. 一个类实现接口，必须重写完全部接口的全部抽象方法，否则这个类需要定义成抽象类

```
public interface Law {
    void rule();
}

public interface SportMan {
    void eat();
}


public class PingPongMan implements Law, SportMan {
    @Override
    public void rule() {
        System.out.println("遵纪守法");
    }

    @Override
    public void eat() {
        System.out.println("吃饭");

    }


    public static void main(String[] args) {
        final PingPongMan man = new PingPongMan();
        man.eat();
        man.rule();
    }
}
```

```
public abstract class PingPongMan implements Law, SportMan {
    @Override
    public void rule() {
        System.out.println("遵纪守法");
```

```
    }

    //当没有重写完所实现接口的所有抽象方法，就要将这个实现类定义为抽象类
/*    @Override
    public void eat() {

    }*/

    public static void main(String[] args) {
        //不能实例
    }
}
```

5.0 接口与接口的关系（多继承）

1. 接口和接口的关系：多继承，一个接口可以同时继承多个接口
2. 接口多继承的作用
   规范合并，整合多个接口为同一个接口，便于子类实现。
3. 类和类的关系：单继承。
4. 类和接口的关系：多实现

```
package inteface_demo;

/**
 * ClassName: Law
 * Description:接口
 * date:2022/3/8
 *
 * @author fgcy
 * @since JDK 1.8
 */

public interface Law {
    void rule();
}


--------------------------------------------------------------------------------
-------------------------------------------

package inteface_demo;

/**
 * ClassName: SportMan
 * Description:接口
 * date:2022/3/8
 *
 * @author fgcy
 * @since JDK 1.8
 */

public interface SportMan {
    void eat();
}
```

```
-----------------------------------------------------------------------------------------------------------
```

```java
package inteface_demo;

/**
 * ClassName: User
 * Description:规范合并接口
 * date:2022/3/8
 *
 * @author fgcy
 * @since JDK 1.8
 */

public interface User extends Law, SportMan {
    String NAME = "苏未晓";//接口成员变量都是常量，所以前面的修饰可以省略不写

    void test(final String name);//接口是一种约束别人的规范，所以是方法是公开抽象的可以省略public abstract

    @Override
    void eat();

    void eat(int a);
}
```

```
-----------------------------------------------------------------------------------------------------------
```

```java
package inteface_demo;

/**
 * ClassName: PingPongMan
 * Description:实现类
 * date:2022/3/8
 *
 * @author fgcy
 * @since JDK 1.8
 */

public class PingPongMan implements User {//只是继承了一个接口，但是因为所继承的接口多实现了其他接口（规范合并），所以也要遵守全部接口的规范

    @Override
    public void rule() {

    }

    @Override
    public void eat() {//当多个接口都出现同一个一样的方法是，实现类仅仅需要实现一次，因为接口最终跑的是子类的方法

    }

    @Override
    public void eat(int a) {
```

```
    }

    @Override
    public void test(String name) {

    }

    public static void main(String[] args) {
        final PingPongMan man = new PingPongMan();
        man.rule();
        man.eat();
        man.test("a");
    }
}
```

- 规范冲突

```
/**
 * ClassName: SportMan
 * Description:接口
 * date:2022/3/8
 *
 * @author fgcy
 * @since JDK 1.8
 */

public interface SportMan {
    void eat();//与Law中的eat方法一起形成规范冲突
}


/**
 * ClassName: Law
 * Description:接口
 * date:2022/3/8
 *
 * @author fgcy
 * @since JDK 1.8
 */

public interface Law {
    void rule();

    String eat();//与SprotMan中的eat方法一起形成规范冲突
}


/**
 * ClassName: PingPongMan
 * Description:实现类
 * date:2022/3/8
 *
 * @author fgcy
 * @since JDK 1.8
```

```
 */

public abstract class PingPongMan implements SportMan, Law {
    //当所实现的两个不同的接口中出现，方法名相同，先参列表相同，不构成重构，但返回值类型不一致，形成了规范冲突
}
```

6.0 JDK1.8开始接口的新增方法

6.1 第一种：默认方法

```
第一种：默认方法
  类似于之前写的实例方法，必须用default修饰
  默认会用public修饰
  此方法只属于该接口的实现类，接口不能调用
  默认方法可以实现，在丰富接口功能的同时又不对子类代码进行更改
```

```
public class PingPongMan implements SportMan {//如果所实现的接口全是默认方法，则该对象可以直接是实例出来
    @Test
    public void test1(){
        final PingPongMan pingPongMan = new PingPongMan();
        pingPongMan.run();
    }
}


interface SportMan {
    default void run() {//默认方法只存在于接口中
        System.out.println("只能由实现类调用的，默认方法！");
    }
}
//  默认方法！只能由实现类调用的，
```

6.2 第二种:静态方法

```
第二种:静态方法
  默认会public修饰
  必须用static修饰
  接口的静态方法必须用本身的接口名来调用（与类区分开----》因为类的静态方法可以有实现类调用，可以由子类类名调用）
```

```
public class PingPongMan implements SportMan {

    @Test
    public void test1() {
        SportMan.go();//接口的静态方法只能由接口调用，不能由实现类调用（区别子类可以调用父类的静态方法------》共享，非继承）
    }
}
```

```
interface SportMan {
    static void go() {
        System.out.println("接口中的静态方法");
    }
}
```

6.3第三种:私有方法

```
第三种:私有方法
  必须使用private修饰
  JDK 1.9才开始有的
  只能在本类中被其他的默认方法或者私有方法访问
```

```
public class PingPongMan implements SportMan {

    @Test
    public void test1() {
        new PingPongMan().go();//通过实例调用接口默认方法，再通过默认方法调用私有方法
    }
}


interface SportMan {
    default void go() {
        System.out.println("默认方法，跑。。。");
        run();
    }

    /**
     * 私有方法（实例方法）,实例方法不能被接口调用，但因为是私有权限，也不能被实现类调用；只能被本接口中的实例方法（默认方法）调用
     */
    private void run() {
        System.out.println("私有方法");
    }
}
```

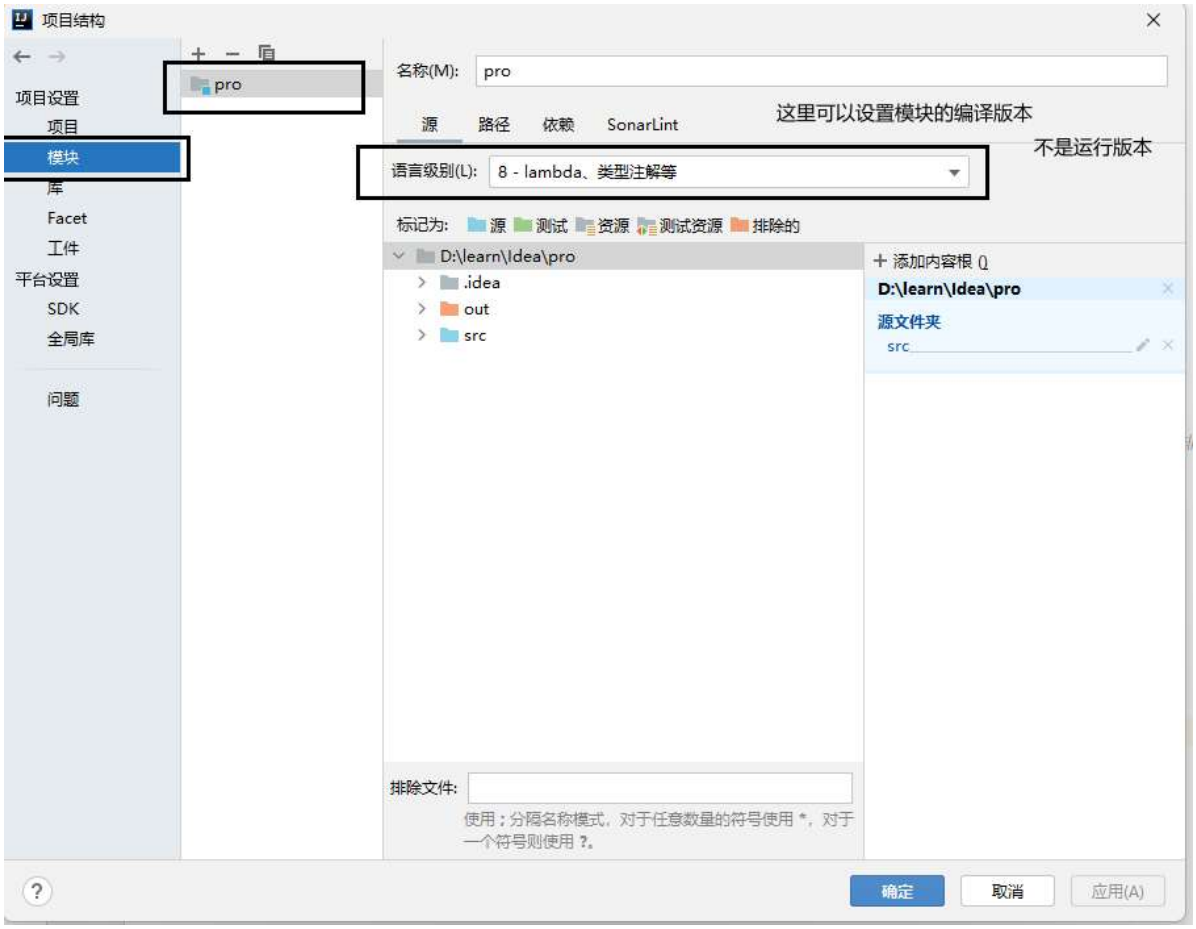
6.4 小结

```
public interface MyT {
    private void test1(){
        test2();//调用静态方法
        test3();//调用默认方法
        System.out.println("接口私有方法！");
    }
    static void test2(){
        System.out.println("接口静态方法");
    }

    default void test3(){
        test1();//调用私有方法
        test2();//调用静态方法
        System.out.println("接口默认方法");
    }
```

```
}
```

```
CRTL+ALT+SHIFT+S---------------------->进入项目结构
CRTL+ALT+S---------------------------->进入设置
```



7.0 接口的注意事项

接口的注意事项

1、接口不能创建对象

2、一个类实现多个接口，多个接口中有同样的静态方法不冲突。
接口的静态方法只能由接口调用，实现类不能调用；所以不会冲突

3、一个类继承了父类，同时又实现了接口，父类中和接口中有同名方法，默认用父类的。
如果子类重写了就使用子类的
4、一个类实现了多个接口，多个接口中存在同名的 默认方法，不冲突，这个类重写该方法即可。
如果多个接口中存在不构成同构的同名方法，则存在规范冲突；不能实现这两个接口（不能多实现）

5、一个接口继承多个接口，是没有问题的，如果多个接口中存在 规范冲突则 不能多继承。