第 14 天面向对象

今日内容介绍

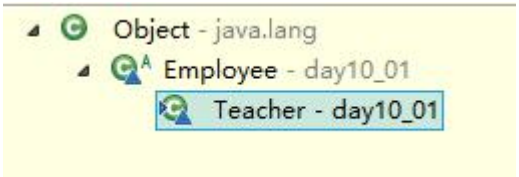
- Eclipse 常用快捷键操作
- Eclipse 文档注释导出帮助文档
- Eclipse 项目的 jar 包导出与使用 jar 包
- 不同修饰符混合使用细节
- 辨析何时定义变量为成员变量
- 类、抽象类、接口作为方法参数
- 类、抽象类、接口作为方法返回值

第 1 章 Eclipse 的应用

1.1常用快捷操作

- Ctrl+T：查看所选中类的继承树

  例如，在下面代码中，选中 Teacher 类名，然后按 Ctrl+T，就会显示出 Teacher 类的继承关系




```
//员工
abstract class Employee{
    public abstract void work();
}
```

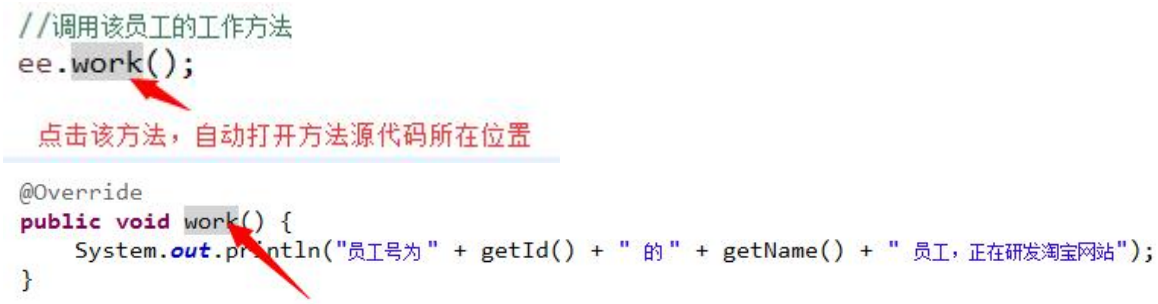
```
//讲师
class Teacher extends Employee {
    public void work() {
        System.out.println("正在讲解 Java");
    }
}
```

- 查看所选中类的源代码

  Ctrl+滑动鼠标点击类名，或者选中类名后，按 F3 键查看所选中类的源代码。

- 查看所选中方法的源代码

  Ctrl+滑动鼠标点击方法名，或者选中方法名后，按 F3 键查看所选中方法的源代码。



- Eclipse 中的 JRE System Library 是默认的 Eclipse 依赖 JRE 中的类库。在该位置可以查找到

  平常使用的 String 类、Random 类、Math 类等。

1.2文档注释导出帮助文档

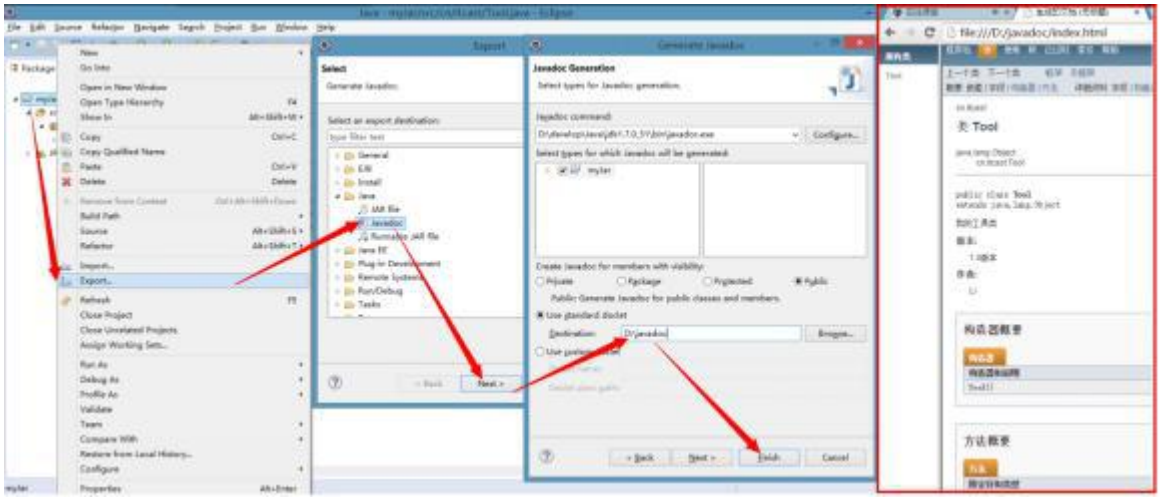
在 eclipse 使用时，可以配合文档注释，导出对类的说明文档，从而供其他人阅读学习与使用。

通过使用文档注释，将类或者方法进行注释用@简单标注基本信息。如@author 作者、

@version 代码版本、@param 方法参数、@return 方法返回值等。

```
package cn.itcast;
/**
 * 我的工具类
 * @author Li
 * @version 1.0 版本
 */
public class Tool {
    /**
     * 返回两个整数的累加和
     * @param num1  第一个数
     * @param num2  第二个数
     * @return  返回累加和
     */
    public static int getSum(int num1, int num2){
        return num1 + num2;
    }
}
```

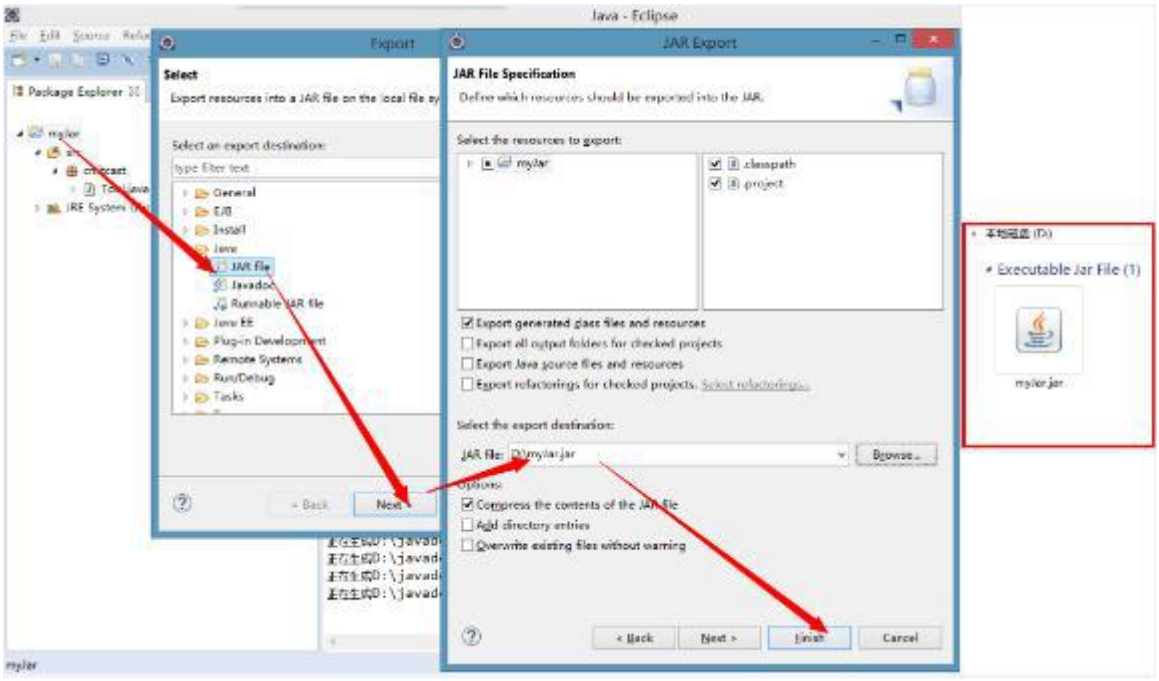
使用 Eclipse 导出 javadoc 文档即可，操作步骤如下图：



1.3项目的 jar 包导入与导出

jar 包是一个可以包含许多.class 文件的压缩文件。我们可以将一个 jar 包加入到项目的依赖中，从而该项目可以使用该 jar 下的所有类；也可以把项目中所有的类打包到指定的 jar 包，提供给其他项目使用。

- **导出 jar 包：**即把项目中所有类，打包到指定的 jar 包中，步骤如下图：



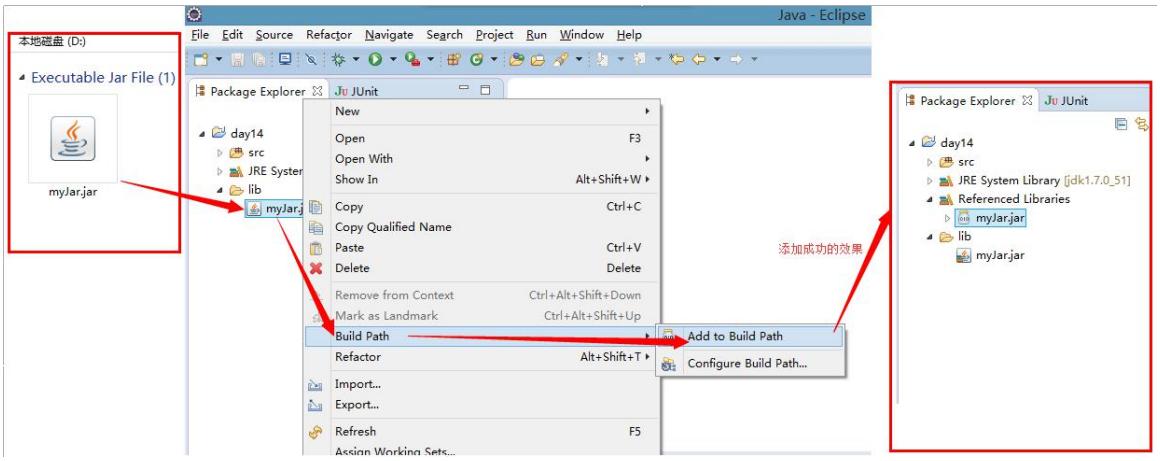
- **导入 jar 包：**即把指定的 jar 包，加入到指项目中，提供给项目使用。

导入 jar 包的过程是将 jar 包加入到项目的.classpath 文件中去，让项目识别，便可以使用 jar 包中所有的.class 文件类。以下是加入步骤：

1：项目根文件夹下创建 lib 文件夹，用于同一管理所有的 jar 文件

2：把 jar 文件复制到 lib 文件夹中

3：右键点击 jar 文件，点击 Build Path，选择 Add to Build Path，此时查看项目根文件夹下的.classpath 文件，发现新加入的 jar 包路径被配置到了该文件中。说明可以使用 jar 包中所有类了。



- 注意：

  Jar 包加入后，必须 Add to Build Path 才能使用

  Jar 包加入后，加入的类也必须导包，如果加入的类其包名与现有类包名相同，则视作在同一个包下。(不常见)

第 2 章 面向对象

2.1不同修饰符使用细节

常用来修饰类、方法、变量的修饰符如下：

- public 权限修饰符，公共访问，类,方法,成员变量
- protected 权限修饰符，受保护访问，方法,成员变量
- 默认什么也不写 也是一种权限修饰符，默认访问，类,方法,成员变量
- private 权限修饰符，私有访问，方法,成员变量
- static 静态修饰符 方法,成员变量
- final 最终修饰符 类,方法,成员变量,局部变量
- abstract 抽象修饰符 类 ,方法

我们编写程序时，权限修饰符一般放于所有修饰符之前，不同的权限修饰符不能同时使用；

同时，abstract 与 private 不能同时使用；

同时，abstract 与 static 不能同时使用；

同时，abstract 与 final 不能同时使用。

- 修饰**类**能够使用的修饰符：

  修饰类只能使用 public、默认的、final、abstract 关键字

  使用最多的是 public 关键字

```
public class Demo {} //最常用的方式
class Demo2{}
public final class Demo3{}
public abstract class Demo4{}
```

- 修饰**成员变量**能够使用的修饰符：

  public： 公共的

  protected： 受保护的

  ： 默认的

  private ：私有的

  final： 最终的

  static： 静态的

  使用最多的是 private

```
public int count = 100;
protected int count2 = 100;
int count3 = 100;
private int count4 = 100; //最常用的方式
public final int count5 = 100;
public static int count6 = 100;
```

- 修饰**构造方法**能够使用的修饰符：

  public： 公共的

  protected： 受保护的

  ： 默认的

  private ：私有的

  使用最多的是 public

```
public Demo(){} //最常用的方式
protected Demo(){}
Demo(){}
private Demo(){}
```

- 修饰**成员方法**能够使用的修饰符：

  public： 公共的

  protected： 受保护的

  ： 默认的

  private ：私有的

  final： 最终的

  static： 静态的

  abstract： 抽象的

  使用最多的是 public

```
public void method1(){}//最常用的方式
protected void method2(){}
void method3(){}
private void method4(){}
public final void method5(){}
public static void method6(){}//最常用的方式
```

```
    public abstract void method7();//最常用的方式
```

第 3 章　自定义数据类型的使用

3.1辨析成员变量与方法参数的设计定义

- 定义长方形类，包含求周长与求面积的方法
- 定义数学工具类，包含求两个数和的二倍与求两个数积的方法

思考：这两个类的计算方法均需要两个数参与计算，请问两个数定义在成员位置还是形参位置更好，为什么？

如果变量是该类的一部分时，定义成成员变量。

如果变量不应该是类的一部分，而仅仅是功能当中需要参与计算的数，则定义为形参变量。

- 数学工具类

```
public class MathTool {
    //求两个数的和的二倍
    public double sum2times(int number,int number2) {
        return (number+number2)*2;
    }
    //求两个数的积
    public double area(int number,int number2) {
        return number*number2;
    }
}
```

- 长方形类

```
public class CFX {
    //因为长与宽，在现实事物中属于事物的一部分，所以定义成员变量
    private int chang;
    private int kuan;

    public CFX(int chang, int kuan) {
```

```
            this.chang = chang;
            this.kuan = kuan;
        }

        //求长与宽的周长
        public double zhouChang() {
            return (chang+kuan)*2;
        }
        //求长与宽的面积
        public double mianJi() {
            return chang*kuan;
        }
        public int getChang() {
            return chang;
        }
        public void setChang(int chang) {
            this.chang = chang;
        }
        public int getKuan() {
            return kuan;
        }
        public void setKuan(int kuan) {
            this.kuan = kuan;
        }
    }
```

3.2类作为方法参数与返回值

- 类作为方法参数

在编写程序中，会经常碰到调用的方法要接收的是一个类类型的情况，那么这时，要向方法中传入该类的对象。如下代码演示：

```
    class Person{
        public void show(){
            System.out.println("show 方法执行了");
        }
    }
    //测试类
    public class Test {
        public static void main(String[] args) {
            //创建 Person 对象
            Person p = new Person();
```

```
        //调用 method 方法
        method(p);
    }

    //定义一个方法 method，用来接收一个 Person 对象，在方法中调用 Person 对象的 show 方法
    public static void method(Person p){
        p.show();
    }
}
```

- 类作为方法返回值

写程序调用方法时，我们以后会经常碰到返回一个类类型的返回值，那么这时，该方法要返回一个该类的对象。如下代码演示：

```
class Person{
    public void show(){
        System.out.println("show 方法执行了");
    }
}
//测试类
public class Test {
    public static void main(String[] args) {
        //调用 method 方法，获取返回的 Person 对象
        Person p = method();
        //调用 p 对象中的 show 方法
        p.show();
    }

    //定义一个方法 method，用来获取一个 Person 对象，在方法中完成 Person 对象的创建
    public static Person method(){
        Person p = new Person();
        return p;
    }
}
```

3.3抽象类作为方法参数与返回值

- 抽象类作为方法参数

今后开发中，抽象类作为方法参数的情况也很多见。当遇到方法参数为抽象类类型时，要传入

一个实现抽象类所有抽象方法的子类对象。如下代码演示：

```
//抽象类
abstract class Person{
    public abstract void show();
}
class Student extends Person{
    @Override
    public void show() {
        System.out.println("重写了 show 方法");
    }
}
//测试类
public class Test {
    public static void main(String[] args) {
        //通过多态的方式，创建一个 Person 类型的变量，而这个对象实际是 Student
        Person p = new Student();
        //调用 method 方法
        method(p);
    }

    //定义一个方法 method，用来接收一个 Person 类型对象，在方法中调用 Person 对象的 show 方法
    public static void method(Person p){//抽象类作为参数
        //通过 p 变量调用 show 方法,这时实际调用的是 Student 对象中的 show 方法
        p.show();
    }
}
```

- 抽象类作为方法返回值

抽象类作为方法返回值的情况，也是有的，这时需要返回一个实现抽象类所有抽象方法的子类对象。如下代码演示：

```
//抽象类
abstract class Person{
    public abstract void show();
}
class Student extends Person{
    @Override
    public void show() {
        System.out.println("重写了 show 方法");
    }
}
```

```
//测试类
public class Test {
    public static void main(String[] args) {
        //调用 method 方法，获取返回的 Person 对象
        Person p = method();
        //通过 p 变量调用 show 方法,这时实际调用的是 Student 对象中的 show 方法
        p.show();
    }

    //定义一个方法 method，用来获取一个 Person 对象，在方法中完成 Person 对象的创建
    public static Person method(){
        Person p = new Student();
        return p;
    }
}
```

3.4接口作为方法参数与返回值

- 接口作为方法参数

接口作为方法参数的情况是很常见的，经常会碰到。当遇到方法参数为接口类型时，那么该方法要传入一个接口实现类对象。如下代码演示。

```
//接口
interface Smoke{
    public abstract void smoking();
}
class Student implements Smoke{
    @Override
    public void smoking() {
        System.out.println("课下吸口烟，赛过活神仙");
    }
}
//测试类
public class Test {
    public static void main(String[] args) {
        //通过多态的方式，创建一个 Smoke 类型的变量，而这个对象实际是 Student
        Smoke s = new Student();
        //调用 method 方法
        method(s);
    }

    //定义一个方法 method，用来接收一个 Smoke 类型对象，在方法中调用 Smoke 对象的 show 方法
```

```
    public static void method(Smoke sm){//接口作为参数
        //通过 sm 变量调用 smoking 方法，这时实际调用的是 Student 对象中的 smoking 方法
        sm.smoking();
    }
}
```

- 接口作为方法返回值

接口作为方法返回值的情况，在后面的学习中会碰到。当遇到方法返回值是接口类型时，那么该方法需要返回一个接口实现类对象。如下代码演示。

```
//接口
interface Smoke{
    public abstract void smoking();
}
class Student implements Smoke{
    @Override
    public void smoking() {
        System.out.println("课下吸口烟，赛过活神仙");
    }
}
//测试类
public class Test {
    public static void main(String[] args) {
        //调用 method 方法，获取返回的会吸烟的对象
        Smoke s = method();
        //通过 s 变量调用 smoking 方法,这时实际调用的是 Student 对象中的 smoking 方法
        s.smoking();
    }

    //定义一个方法 method，用来获取一个具备吸烟功能的对象，并在方法中完成吸烟者的创建
    public static Smoke method(){
        Smoke sm = new Student();
        return sm;
    }
}
```

第 4 章　星级酒店案例

4.1案例介绍

某五星级酒店，资金雄厚，要招聘多名员工（经理、厨师、服务员）。入职的员工需要记录个人信息（姓名、工号、经理特有奖金属性）。他们都有自己的工作要做。

本案例要完成如下需求：

- 获取酒店幸运员工；
- 酒店开设 VIP 服务，酒店的厨师与服务员可以提供 VIP 服务。（厨师做菜加量、服务员给顾客倒酒）。
- 编写测试类
  - 向酒店中，增加多名员工（其中包含 1 名经理，1 名厨师、2 名服务员）；
  - 调用酒店员工的工作功能
  - 调用酒店员工的 VIP 服务功能

4.2案例需求分析

- 根据“某五星级酒店，资金雄厚……都有自己的工作要做。”分析出，该题目中包含酒店，可以把它封装成类，多名员工）。

```
class 员工 {
    属性：姓名
    属性：工号
    方法：工作
}
class 厨师 extends 员工{}
class 服务员 extends 员工{}
class 经理 extends 员工 {
```

```
        属性：奖金
    }
```

员工的类型有经理、厨师、服务员，它们有共同的属性（姓名、工号、），经理额外属性（奖金）。

- 根据“向酒店中，增加多名员工（其中包含 1 名经理，1 名厨师、2 名服务员）”。分析出，要创建一个酒店对象，并添加 4 名员工到酒店对象的员工集合中。

```
酒店员工集合添加新员工： 经理对象
酒店员工集合添加新员工： 厨师对象
酒店员工集合添加新员工： 服务员对象
酒店员工集合添加新员工： 服务员对象
```

- 根据“获取酒店幸运员工”。分析出，从酒店员工集合随机得到一名员工对象。

```
1. 从酒店员工集合长度范围内，随机产生一个随机数
2. 使用该随机数作为集合的索引，返回该索引处对应的员工对象
```

- 根据“酒店开设 VIP 服务，酒店的厨师与服务员可以提供 VIP 服务。（厨师做菜加量、服务员给顾客倒酒）”。分析出，这是要增加一个 VIP 的接口，接口中提供个 VIP 服务的方法。让厨师与服务员实现该接口。

```
interface VIP 服务{
        抽象方法：服务
}
class 厨师 extends 员工 implements VIP 服务{ 重写服务方法 }
class 服务员 extends 员工 implements VIP 服务{ 重写服务方法 }
```

4.3实现代码步骤

- VIP 服务

```
public interface VIP {
    public abstract void server(); //服务
}
```

- 员工

```
/*
 *  员工：
        姓名 String
        工号 String

 */
public abstract class YuanGong {
    // 成员变量
    private String xingMing;
    private String gongHao;
    // 构造方法
    public YuanGong() {
        super();
    }
    public YuanGong(String xingMing, String gongHao) {
        super();
        this.xingMing = xingMing;
        this.gongHao = gongHao;

    }
    // 抽象方法
    public abstract void work();

    // getters 与 setters
    public String getXingMing() {
        return xingMing;
    }
    public void setXingMing(String xingMing) {
        this.xingMing = xingMing;
    }
    public String getGongHao() {
        return gongHao;
    }
    public void setGongHao(String gongHao) {
        this.gongHao = gongHao;
    }

}
```

- 服务员

```
/*
```

```
 * 定义员工的子类 服务员类
 */
public class FuWuYuan extends YuanGong implements VIP {
    public FuWuYuan() {
        super();
    }

    public FuWuYuan(String xingMing, String gongHao) {
        super(xingMing, gongHao);
    }
    @Override
    public void work() {
        System.out.println("亲，全身心为您服务，记得给好评哦");
    }
    @Override
    public void server() {
        System.out.println("给顾客倒酒");
    }
}
```

- 经理

```
/*
 * 经理在员工的基础上，添加了奖金成员
 */
public class JingLi extends YuanGong {
    private double jiangJin;

    public JingLi() {
        super();
    }
    public JingLi(String xingMing, String gongHao, double jiangJin) {
        super(xingMing, gongHao);
        this.jiangJin = jiangJin;
    }

    public double getJiangJin() {
        return jiangJin;
    }
    public void setJiangJin(double jiangJin) {
        this.jiangJin = jiangJin;
    }

    @Override
```

```
        public void work() {
            System.out.println("哪个员工让顾客不满意，我扣谁钱");
        };
    }
```

- 厨师

```
/*
 * 定义员工的子类 厨师类
 */
public class ChuShi extends YuanGong implements VIP{
    public ChuShi() {
        super();
    }
    public ChuShi(String xingMing, String gongHao) {
        super(xingMing, gongHao);
    }

    @Override
    public void work() {
        System.out.println("我做饭，放心吃吧，包您满意");
    }
    @Override
    public void server() {
        System.out.println("做菜加量加料");
    }
}
```

- 测试类

```
public class Test {
    public static void main(String[] args) {
    }
}
```

第 5 章　总结

5.1知识点总结

- 不同修饰符的使用
  - 类，最常使用 public 修饰
  - 成员变量，最常使用 private 修饰
  - 成员方法，最常使用 public 修饰
- 自定义数据类型的使用
  - 类作为方法参数时，说明要向方法中传入该类的对象
  - 类作为方法返回值时，说明该方法要返回一个该类的对象。
  - 抽象类作为方法参数时，说明要传入一个实现抽象类所有抽象方法的子类对象。
  - 抽象类作为方法返回值时，说明需要返回一个实现抽象类所有抽象方法的子类对象。
  - 接口作为方法参数时，说明该方法要传入一个接口实现类对象。
  - 接口作为方法返回值时，说明该方法需要返回一个接口实现类对象。