讲义05：类的特性

主要内容

- 讲授内容
  - 包的使用、访问权限
  - 方法重载
  - 静态成员
  - this关键字
- 内容资料来源
  - 教材第7章、第8章(8.5节)
  - 其他资料补充

1 包 package (8.5)

- Java使用包(package) 对类进行组织和管理.
- 使用包的理由如下:
  - 查找定位类
  - 避免命名冲突
  - 便于发布软件
  - 保护类

1.1 包的命名习惯

包是有层次关系的, 一个包中还可以有包, 称为子包.

例如：java.lang.Math
其中：Math是包lang中的类; lang是包java中的一个子包.

保证包名的唯一性非常重要, Java建议采用Internet域名的倒序作为包的前缀.

例如: 在华南农业大学的主机上创建一个包mypackage。该主机的域名是scau.edu.cn, 则按照Java建议方式, 该包的完整名称: cn.edu.scau.mypackage.

通常包名都用小写字母命名.

1.2 包与目录

Java中，包名与文件系统的目录结构一一对应.

例如：包 cn.edu.scau.mypackage 的目录结构如下图.

1.3 在包中添加类

Java中的每个类都属于某一个包, 类在编译时被添加到包中.
默认情况下, 类在编译时放在当前目录(默认包).

把类添加到指定包中, 只需在源程序的最前端加上:

```
package packageName;
```

```
package cn.edu.scau.mypackage;

public class Demo {
   public static double format() {
      //……
   }
}
```

1.4 import语句

有2种使用包中类的方式

- 使用类的全称
  - javax.swing.JOptionPane;
- 使用import语句导入包中的类
  - 导入包中全部类:
    import javax.swing.*;
  - 导入包中指定的类:
    import javax.swing.JOptionPane;

- import语句位于package语句之后, 类定义之前。

2 访问权限(7.1及补充)

访问权限是指限制在一段代码中能否访问一个类或能否通过“.”访问的类中定义的方法或成员变量。

Java语言使用3个关键字实现了4种访问权限。

- public
  修饰目标：类、方法和成员变量。
  访问范围：公开范围，即应用程序中任何位置均可访问。
- private
  修饰目标：方法、成员变量。
  访问范围：私有范围，即被修饰目标所在的类中可以访问。
- protected，与继承有关
- 默认情况，即无修饰符
  修饰目标：类、方法和成员变量
  访问范围：包范围，即被修饰目标同一包的任何类中访问。

2 访问权限

可见性修饰符对类成员的作用

类的成员可以使用3种修饰符或默认范围

包p1;

```
public class C1 {
  public int x;
  int y;
  private int z;

  public void m1(){}
  void m2(){}
  private void m3(){}
}
```

```
public class C2 {
  C1 o = new C1();
  可以访问 o.x
  可以访问 o.y
  不能访问 o.z

  可以调用 o.m1()
  可以调用 o.m2()
  不能调用 o.m3()
}
```

包p2;

```
public class C3 {
  C1 o = new C1();
  可以访问 o.x
  不能访问 o.y
  不能访问 o.z

  可以调用 o.m1()
  不能调用 o.m2()
  不能调用 o.m3()
}
```

2 访问权限

可见性修饰符对类的作用

顶层类可以使用public或默认范围

包p1;

```
class C1 {
  ……
}
```

```
public class C2 {
  可以访问 C1
}
```

包p2;

```
public class C3 {
  不能访问C1
}
```

2.1 访问权限使用 – 数据域封装

使用访问权限对成员变量(数据域)进行封装

允许通过对象直接修改数据域不是好的方法，会使类难于维护且不易修改.

```
public class Student {
  public int id;
  ……
}
```

使用

```
public class Client {
  public static void main(String[] args) {
    Student student = new Student();
    student.id = 20040101;
    ……
  }
}
```

问题: 如果Student的学号使用int不能存储，需要改为String，则哪些程序需要修改呢?

2.1 访问权限使用 – 数据域封装

数据域封装(data field encapsulation)

作用：把对数据域的直接访问变成间接访问。

实现：

步骤1, 使用private修饰数据域

步骤2, 为每个数据域创建访问器方法和修改器方法

访问器方法：

```
public  返回类型  get属性名() {…}
public  boolean  is属性名() {…}
```

修改器方法：

```
public void set属性名(数据类型 参数) {…}
```

2.1 访问权限使用 – 数据域封装

```
public class Book {
    private String isbn;
    private boolean notPublished;

    public String getIsbn() {
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public boolean isNotPublished() {
        return notPublished;
    }
    public void setNotPublished(boolean notPublished) {
        this.notPublished = notPublished;
    }
}
```

数据域与方法局部变量同名时可以使用this引用数据域

完整代码:examples/lecture05/encapsaluation的Book类

2.2 访问权限使用 – 如何修饰方法

一个类中定义的方法根据其调用范围确定其访问修饰符：

- 只在本类内部使用的方法使用private。
- 提供给其他类使用的方法使用public 或其他访问权限。

```
public class Book {
    // 获得某本图书销售金额的方法
    public double getSaleAmount() {
        return price * salesVolume;
    }
    ......
}
```

完整代码:examples/lecture05/encapsaluation的Book类

2.2 访问权限使用 – 如何修饰构造方法

类的构造方法根据需要创建对象的范围确定其访问修饰符：

- 只在本类内部创建该类的对象使用private修饰构造方法。
- 在其他类创建对象使用public或其他访问权限修饰构造方法。
- 默认构造方法的访问权限与类的访问权限相同
- 类的所有构造方法均为private时，无法在类外部创建对象

```
public class Book {
    public Book() {
    }
    public Book(String isbn, String title, double price, int salesVolume, boolean notPublished) {
        super();
        this.isbn = isbn;
        this.title = title;
        this.price = price;
        this.salesVolume = salesVolume;
        this.notPublished = notPublished;
    }
    ......
}
```

完整代码:examples/lecture05/encapsaluation的Book类

3 this关键字(集中补充)

this关键字主要有2种使用方式：

- 在类的构造方法中使用，调用本类的其他构造方法。
  语法：this(实参表)
  如果使用该语句，必须在构造方法的第一句

- 在类的构造方法和实例方法中使用，代表对当前对象的引用。
  访问实例变量语法：this.实例变量名
  方法实例方法语法：this.实例方法名(实参表)
  每个构造方法和实例方法中都隐含有一个this引用。

3 this关键字

```
public class Rectangle {
    double width;
    double height;
    public Rectangle(double width, double height) {
        this.width = width; //this必须使用,引用正在创建的对象
        this.height = height;
    }
    public Rectangle() {
        this(1.0, 1.0);      //调用本类的其他构造方法
    }
    public void setWidth(double width) {
        this.width = width; //this必须使用,引用当前对象
    }
    public double getArea() {
        return this.width * this.height; //this可以省略
    }
}
```

3 this关键字

```
public class Rectangle {
    //省略其它部分
    public void setWidth(double width) {
        this.width = width;
    }
}
```

分析以下语句的执行了解this值的获取：

```
Rectangle r1 = new Rectangle(10.0, 10.0);
Rectangle r2 = new Rectangle(20.0, 20.0);

r1.setWidth(100.0); //r1-->this, 100.0-->width
r2.setWidth(200.0); //r2-->this, 200.0-->width
```

4 实例成员与静态成员(7.4)

- static修饰是静态成员
  - 静态成员变量
  - 静态方法
- 未用static修饰是实例成员
  - 实例成员变量
  - 实例方法

```
public class StaticDemo {
    // 实例数据域
    int instanceData;
    // 静态数据域
    static int staticData;
    // 实例方法
    int instanceMethod() {
        return instanceData;
    }
    // 静态方法
    static int staticMethod() {
        return staticData;
    }
}
```

4 实例成员与静态成员

实例变量与静态变量

- 不同对象的实例变量互不相同
  - 实例变量只有其对象创建后才会分配内存空间。
  - 使用new运算符创建对象时，每个对象的实例变量占用各自不同的内存空间。
- 所有对象共享静态变量
  - 静态变量在类加载到内存时分配内存空间。
  - 使用new运算符创建对象时，各个对象的实例变量共享已经存在的静态变量的内存空间。
- 通过类名直接访问静态变量
  - 对静态变量的使用方式是：**类名.静态变量**
  - 对实例变量的使用方式是：**引用变量.实例变量**

4 实例成员与静态成员

实例方法与静态方法

- 实例方法调用方式：**引用变量.实例方法()**
  - 当创建类的对象时，实例方法才会分配入口地址。
  - 注意，多个对象的同一实例方法的入口地址是共享的。
  - 在实例方法中可以直接访问实例成员和类成员。
- 静态方法调用方式：**类名.静态方法()**
  - 类加载到内存时，为静态方法分配内存空间。
  - 在静态方法中不能访问未创建对象的实例成员。

4 实例成员与静态成员

示例: examples\lecture05\StaticDemo

```
public class Circle {
    // 静态变量 pi
    private static double pi = 3.14;
    // 实例变量 radius
    private double radius;
    // 2个public的构造方法
    public Circle() {    }
    public Circle(double radius) {   this.radius = radius;    }
    // 实例方法，返回圆对象的面积
    // 在本方法中可以省略 Circle. 和 this.
    public double getArea() {   return Circle.pi * this.radius * this.radius;    }
    // 以下是对静态变量pi的封装，均为静态方法
    public static double getPi() {   return pi;    }
    public static void setPi(double pi) {   Circle.pi = pi;   }
    // 以下是对实例变量radius的封装，均为实例方法
    public double getRadius() {  return radius;    }
    public void setRadius(double radius) {   this.radius = radius;   }
}
```

4 实例成员与静态成员

示例: examples\lecture05\StaticDemo

```
public class Main {
    public static void main(String[] args) {
        // 程序中首次使用一个类时，该类加载到内存
        System.out.println(Circle.getPi());

        Circle c1 = new Circle(1.0);
        Circle c2 = new Circle(10.0);
        System.out.println("c1的面积: " + c1.getArea());
        System.out.println("c2的面积: " + c2.getArea());

        System.out.println("------------------");

        Circle.setPi(4);
        System.out.println(Circle.getPi());
        System.out.println("c1的面积: " + c1.getArea());
        System.out.println("c2的面积: " + c2.getArea());
    }
}
```

5 静态初始化器与实例初始化器(7.4.4及补充)

初始化器是直接在类中定义的用一对{}括起来的语句组。

- **静态初始化器**使用static关键字修饰，用来初始化静态变量。
- **实例初始化器**没有修饰关键字，用来初始化实例变量。

一个类可以有0个或多个静态和实例初始化器。

静态初始化器 →

实例初始化器 →

```
public class ClassName {
    static {
        语句组;
    }
    {
        语句组;
    }
}
```

5 静态初始化器与实例初始化器

静态初始化器的执行：类首次加载到内存时，首先是静态数据域变量的变量初始化；然后按排列顺序执行类中staic初始化器。

实例初始化器的执行：每次使用“new 构造方法();”创建对象时，首先是实例数据域变量的变量初始化，然后开始执行本类的构造方法，但在构造方法第一条语句执行之前，按排列顺序执行类中的实例初始化器，然后执行构造方法中的剩余语句。

示例：examples/lecture05/initializer

6 方法重载(7.2)

- 静态多态：基于重载(overload)实现
- 一个类中2个或更多的方法具有相同的名称但不同的形参列表，称为**方法重载**(method overloading).
- 被重载的方法的形参必须不同
  - 形参个数不同
  - 形参的类型不同
  - 形参的排列顺序不同
- 当进行方法调用时，Java编译器寻找最合适的匹配方法调用.
- 不能基于方法的修饰符和返回值进行重载.

- 构造方法、实例方法和静态方法均可以重载。

6 方法重载

示例：examples/lecture05/overload_demo

```
public class NumberUitls {
    public static int max(int a, int b) {
        return a > b ? a : b;
    }

    public static double max(double a, double b) {
        return a > b ? a : b;
    }

    public static int max(int a, int b, int c) {
        return a > b ? a : b > c ? b : c;
    }
}
```

6 方法重载

示例：examples/lecture05/overload_demo

```
// 精确匹配
int n1 = NumberUitls.max(10, 100);
double n2 = NumberUitls.max(10.0, 20.0);
int n3 = NumberUitls.max(1, 2, 3);

// 非精确匹配
float n4 = (float) NumberUitls.max(10.0F, 20.0F);
double n5 = NumberUitls.max(10, 20.0F);

// 错误的调用
double n6 = NumberUitls.max(1.0, 2, 3);
```

7 对象的应用(7.5及补充)

Java的变量分为**基本类型变量**和**引用类型变量**。

引用类型变量存放其引用的对象的地址，因此在通过引用变量要特别注意。

7.1 对象的赋值与复制

两个引用变量之间进行赋值，赋值的是地址，使2个引用变量引用同一对象。

```
Circle c1 = new Circle();
Circle c2 = c1;      // 使c2和c1引用同一对象
```

实现对象复制，如果类有clone方法，可以使用clone方法进行复制。自定义类的clone实现后续章节介绍。

```
import java.util.Date;

Date d1 = new Date();
Date d2 = (Date)d1.clone(); // 克隆与原对象相同的对象
```

7.2 对象的比较

使用关系运算符“==”是比较2个引用变量是否引用同一对象。

```
Circle c1, c2;
c1 == c2   // 两个引用指向同一对象结果为true，否则为false
```

比较2个对象的是否相等，可以使用API提供的方法，例如：

```
Arrays.equals(array1, array2);          // 比较2个数组
Arrays.deepEquals(array1, array2); // 深度比较2个数组

String s1, s2;
s1.equals(s2); //比较2个字符串
```

也可以自己为类定义比较方法，为类提供equals方法后续章节介绍。

7.3 引用变量作为方法的返回值

```
// 方法定义
public Circle createACirlce(double radius) {
    Circle temp = new Circle();
    return temp;   //返回对象的地址
}

// 方法调用
Circle c = createACircle(10.0);  // c得到返回的对象地址
```

方法中的局部变量temp与方法调用中的变量c引用同一对象。

7.4 对象的组合

- 一个类的成员变量可以是Java允许的任何类型，当然也包括其它类的对象类型。
- 例如类A使用类B的对象作为成员变量，则A的对象把B的对象作为组成部分。称为：A has-a B。
- 示例：
  - 一个圆锥对象 包含 一个圆对象 作为底。
  - 一个班级对象 包含 一个教师对象(班主任)和多个学生对象。
  - 扑克对象 包含 多个牌对象

7.4 对象的组合

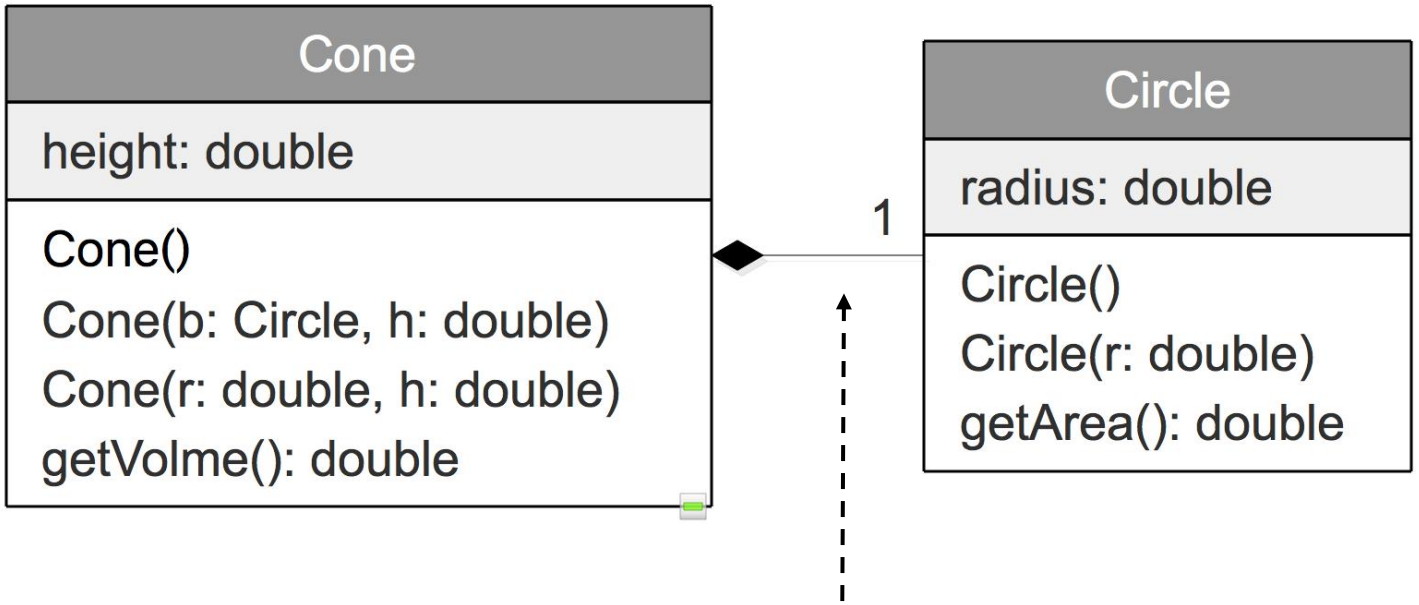
使用UML类图表示类与类之间的关系

类名

成员变量

构造方法

方法



表示2个类之间的组合

类Cone中需要定义一个Circle类型数据域

代码：examples/lecture05/hasademo

课后工作

- 结合教材内容，复习课堂讲授内容
- 作业: