Java 程序设计

第4章 对象和类

田英鑫 tyx@hit.edu.cn
哈尔滨工业大学软件学院

第4章 对象和类

& 本章导读

第4章 对象和类

& 本章重点

- 4.2 对象和类
- 4.5 静态字段和方法
- 4.7 重载方法
- 4.8 对象构造
- 4.10 包

& 本章难点

- 4.2 对象和类
- 4.5 静态字段和方法
- 4.10 包

4.1 面向对象程序设计简介

- 面向过程程序设计
  - 以功能为核心，数据和对数据的操作是分离的，需要把数据传递给过程或函数
  - 程序 = 算法 + 数据结构
- 面向对象程序设计
  - 把数据和对数据的操作放在同一个数据结构中
  - 以对象为核心，更接近人的思维过程，是计算机语言向人类自然语言发展方向上的研究成果
  - 对问题求解的过程是模拟人类社会对事物的处理过程
  - 一切都是对象

4.2 对象和类

- 对象（object）
  - 现实世界中可以明确标识的任何事物
  - 对象的三个特性
    - 行为：对象能做什么，或能对对象施加什么方法
    - 状态：对对象施加方法时，对象如何反映
    - 身份：如何与具有相同行为、状态的其它对象区别
- 类（class）
  - 类是创建对象的模版或蓝图
  - 对象是类的实例（instance）
  - 创建一个对象实例称为实例化（instantiation）

4.2 对象和类

- OOP 和面向过程程序设计比较
  - 面向过程程序设计
    - 自顶向下（自底向上）、逐步求精
    - 把任务分解成子任务
  - 面向对象程序设计
    - 首先分离出类，确定类的属性
    - 然后去寻找类的方法

4.2 对象和类

- 识别类的一个简单方法
  - 寻找问题分析中的名词，例如订单系统中可能形成如下的类
    - 项目（item）
    - 订单（order）
    - 账户（account）
  - 方法与动词相对应，例如对于订单类可以找出如下动词
    - 添加、取消、支付等

4.2 对象和类

- 类之间的关系
  - 依赖（“use-a”）
    - 如果一个类的方法操作了另外一个类的对象，那么这个类就依赖于另一个类
    - 例如：Order类依赖于Account类
  - 聚合（“has-a”）
    - 一个类的对象包含了另一个类的对象
    - 例如：Order类对象包含Item类的对象，Order类和Item类具有聚合关系
  - 继承（“is-a”）
    - 特殊和一般的关系

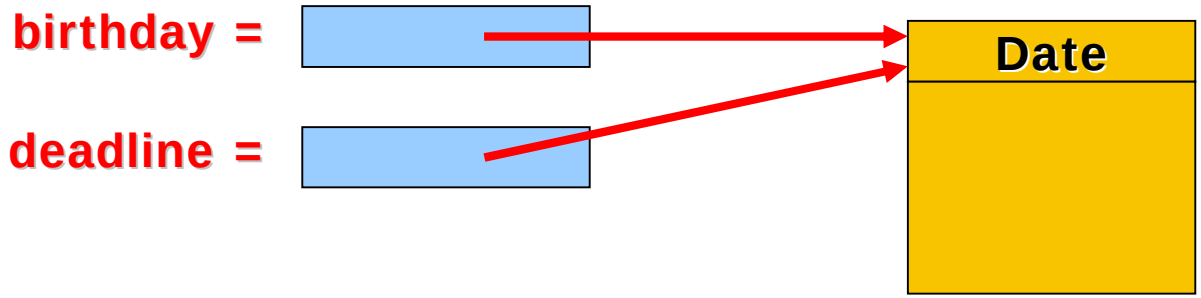
4.3 使用现有类

- 对象和对象变量
  - 创建对象

```
new Date();//这样创建的对象只能使用一次
System.out.println(new Date());
```

  - 对象变量

```
Date birthday = new Date();
Date deadline = birthday;
```



4.4 创建自己的类

- 类的定义
  - Java中，类的定义形式如下

```
class NameOfClass
{
   constructor1
   constructor2      } 构造器（构造方法）
   ...
   method1
   method2           } 方法
   ...
   field1
   field2            } 属性（字段）
   ...
}
```

参见教材例4-2：EmployeeTest.java

4.4 创建自己的类

- 使用多个源文件
  - 可以将每个类放在一个单独的源文件中，例如
    - Employee类放在文件Employee.java中
    - EmployeeTest类放在文件EmployeeTest.java中
- 多个源文件程序的编译
  - 编译可执行类，其它被调用的类会自动编译
    - 当编译器编译EmployeeTest.java时发现其中使用了Employee类，它就会去寻找Employee.class，如果没有找到，编译器将自动编译Employee.java
    - 当编译EmployeeTest.java的时候，可以没有源文件Employee.java，只要Employee.class存在就可以了

4.4 创建自己的类

- 分析 Employee 类
  - 一个构造器（构造方法）和四个方法

```
public Employee(String n,double s,int year,int month,int day);
public String getName();
public double getSalary();
public Date getHireDay();
public void raiseSalary(double byPercent);
```

  - 三个实例字段

```
private String name;
private double salary;
private Date hireDay;
```

4.4 创建自己的类

- 分析 Employee 类
  - 方法被标记为public
    - 关键字public表示在任意类中的任意方法都可以调用这个方法
  - 实例字段被标记为private
    - private关键字用来确保访问这些字段的方法只能是Employee类本身的方法，任何外部方法都无法读或写这些字段
    - 永远不要把实例字段标记为public

4.4 创建自己的类

- 构造器（构造方法）

```
public Employee(String n, double s,
      int year, int month, int day)
{
   name = n;
   salary = s;
   GregorianCalendar calendar
      = new GregorianCalendar(year, month - 1, day);
   // GregorianCalendar uses 0 for January
   hireDay = calendar.getTime();
}
```

4.4 创建自己的类

- 构造器（构造方法）
  - 构造器是用来构造对象的特殊方法，通常在构造器中给实例字段赋初值
  - 构造器和类具有相同的名字
  - 一个类可以有多个构造器
  - 构造器可以有0个、1个或者多个参数
  - 构造器没有返回值
    - 返回void也不行，因为void也是类型
  - 构造器总是和new运算符一起被调用

4.4 创建自己的类

- Employee 类中的方法
  - 以raiseSalary方法为例
    - 该方法为实例方法，实例方法属于特定的实例，实例方法只能通过实例来访问
  - 该方法有两个参数
    - 显式参数是double类型的byPercent
    - 隐式参数是调用该方法的Employee类型的对象
  - 每个方法中，关键字this指向隐式参数
    - 方法中的this指向调用该方法的那个对象
  - 实例方法的调用
    - 对象名.方法名(实参列表);

4.4 创建自己的类

- 为私有实例字段设置访问方法和更改方法
  - 对一个实例字段进行即读取、又设置，需要增加以下三项
  - 一个私有（private）数据字段
  - 一个公有（public）字段访问方法
  - 一个公有（public）字段更改方法

- 例如，如果有实例字段姓名，应该有如下三项

```
//私有实例字段
private String name;
//公有访问方法
//方法名字为get加字段名字
public String getName()
{
   return name;
}
//公有更改方法
//方法名字为set加字段名字
public void setName(String name)
{
   this.name = name;
}
```

4.4 创建自己的类

- 使用访问方法和更改方法的好处
  - 可以改变方法的内部实现，而不影响其外任何调用该方法的代码
    - 例如，如果name的存储方式变成

      String firstName;

      String lastName;

      可以直接修改getName方法，让它返回：

      firstName + " " + lastName
  - 更改方法能进行代码错误检查
    - setSalary方法可以检查salary的值，使它不小于0

4.4 创建自己的类

- 访问私有数据的方法
  - 方法可以访问调用该方法的对象中的私有数据
  - 方法还可以访问其所属类的所有对象的私有数据，例如：

```
class Employee
{
   boolean equals(Employee other)
   {
      return name.equals(other.name);
   }
}
//通常这样调用该方法时 if(harry.equals(boss)) ...
//在harry方法equals中不但可以访问harry的私有字段name
//还可以访问boss的私有字段name
```

4.4 创建自己的类

- 私有方法
  - 通常，大多数方法都是public的
  - 私有（private）方法只能被同一类中的其它方法调用
- 下面的情况应该使用私有方法
  - 与类的使用者无关的方法
    - 公有方法是类对使用者的接口，在实际应用中方法一旦设置为公有（public），在更新类时将很难降低其访问权限
  - 如果类的实现改变了，就难以维护的那些方法
    - 类的设计者可以放心的将某些私有方法丢弃

4.4 创建自己的类

- final 实例字段
  - 可以用final关键字修饰一个实例字段，这种字段在构造对象时必须初始化
  - 用final修饰的实例字段将不能再修改
    - 例如，Employee类的name字段可以声明为final，因为在对象被构造后它的值就不再变化 – 类中没有setName方法

```
class Employee {
    public Employee(String name) {
        this.name = name;
    }
    private final String name;
}
```

4.5 静态字段和方法

- 静态字段（类变量）
  - 用static关键字修饰的字段称为静态字段
  - 静态字段对类的每个实例都是共享的

```
class Employee {
    ...
    public void setId() {
       id = nextId;
       nextId ++;
    }
    //字段id属于类Employee的每一个实例
    private int id;
    //字段nextId属于类Employee，其被该类的每一个对象所共享
    private static int nextId = 1;
}
```

4.5 静态字段和方法

- 常量
  - 静态变量并不常见，静态常量却很普遍
    - 例如：Math类中的PI，System类的out
  - 常量（即final字段）可声明为公有（public）的

```
public class Math {
    ...
    public static final double PI = 3.14159265...;
    ...
}
public class System {
    ...
    public static final PrintStream out = ...;
    ...
}
```

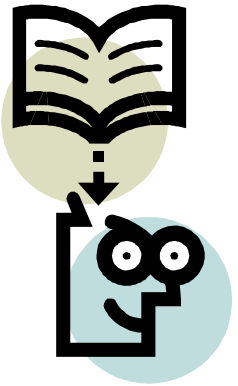
4.5 静态字段和方法

- 静态方法（类方法）
  - 用static关键字修饰的方法称为静态方法
  - 静态方法是不向对象施加操作的方法
- 使用静态方法
  - 静态方法一般完成与其所在类对象无关的功能
    - 例如，Math类中的pow是一个静态方法
  - 静态方法用来访问类中的静态字段

```
public static int getNextId()
{
    return nextId;
}
//调用该方法：int n = Employee.getNextId();
```

4.5 静态字段和方法

- 静态方法的注意事项
  - 静态方法只能访问同类中的静态变量或静态方法，而不能访问非静态成员
  - 对于非静态成员需要先创建类的对象后才能使用，而静态成员在使用前不用创建任何对象
  - 实例方法可以访问同类中的静态成员和非静态成员
- 使用静态方法
  - 方法不需要访问对象状态
  - 方法所需参数都通过显式参数提供
  - 方法只需要访问类的静态字段



4.5 静态字段和方法

- 工厂方法
  - 类中用于产生该类对象的方法称为工厂方法
  - 工厂方法是静态方法
- 使用工厂方法生成对象的原因
  - 构造器无法命名，其名字总是与类名相同

```
NumberFormat.getNumberInstance();
NumberFormat.getCurrencyInstance();
```

  - 工厂方法不但可以返回其所在类的对象，还可以返回从该类继承来的子类的对象，而构造器则不能

4.5 静态字段和方法

- main 方法
  - 为什么main方法是静态的？
    - main方法是java应用程序的入口方法，其不对任何对象施加操作
    - main方法在执行时还不存在任何对象
  - main方法应该放在哪个类中？
    - main方法与其所在的类没有任何关系，可以放在任何类中
    - 为了对类进行单元测试，可以给每个类中都增加一个main方法

参见教材例4-3：StaticTest.java

4.6 方法参数

- 传值调用（call by value）

```
//测试交换两个基本类型（字符、数字、布尔）数据
public class TestPassByValue {
    public static void main(String[] args) {
        int num1 = 1;
        int num2 = 2;
        System.out.println("调用前,num1="+num1+" num2="+num2);
        swap(num1, num2);
        System.out.println("调用后,num1="+num1+" num2="+num2);
    }
    static void swap(int n1, int n2) {
        System.out.println("交换前, n1=" + n1 + " n2=" + n2);
        int temp = n1; n1 = n2; n2 = temp;
        System.out.println("交换后, n1=" + n1 + " n2=" + n2);
    }
}
```

4.6 方法参数

- 传值调用（call by value）
  - 当方法的形式参数是基本类型时，方法不能修改实际参数
  - 当方法的形式参数是对象类型时，方法可以修改实际参数所指对象的状态
  - 当方法的形式参数是对象类型时，方法不能让实际参数指向新的对象

参见教材例4-4： ParamTest.java

4.7 重载方法

- 什么是重载方法？
  - 在类中可以存在方法名相同的多个方法，但方法的原型不能完全相同
    - 方法的原型包括方法名和方法参数列表
- 重载解析
  - 编译器通过匹配具体的方法调用中所使用的值的类型和多个方法头中的参数类型挑选出正确调用的方法，这个过程称为重载解析
  - 如果编译器找不到匹配的参数或者找到多个可能的匹配，则会产生一个编译错误
    - java对调用重载方法采取静态联编的手段

4.7 重载方法

```
public class TestMethodOverloading {
   public static void main(String[] args) {
      System.out.println("3和4的最大值是：" + max(3,4));
      System.out.println("3.0和5.4的最大值是：" + max(3.0,5.4));
      System.out.println("3.0、5.4和10.14的最大值是："
         + max(3.0,5.4,10.14));
   }
   static int max(int num1, int num2) {
      if (num1 > num2) return num1;
      else return num2;
   }
   static double max(double num1, double num2) {
      if (num1 > num2) return num1;
      else return num2;
   }
   static double max(double num1, double num2, double num3) {
      return max(max(num1, num2), num3);
   }
}
```

4.8 对象构造

- 默认字段初始化
  - 如果在构造器中没有显式地给某个字段赋值，那么它会被自动赋为默认值

成员变量类型	初始值
byte、short、int、long	0
float	0.0F
double	0.0D
char	'\u0000'
boolean	false
All reference type	null

4.8 对象构造

- 默认构造器
  - 默认构造器是指没有参数的构造器
  - 如果没有显式编写构造器，则编译器会自动创建一个方法体为空的默认构造器，并且对字段进行默认初始化
  - 当重载一个类的构造器时，编译器将不会创建默认的构造器
  - 一般的，当我们重载一个类的构造器时，应该显示的创建该类的无参构造器

4.8 对象构造

- 显式字段初始化
  - 可以在类的定义中简单地把初值赋给任何字段
  - 初始值也可以是方法调用，例如：

```
class Employee {
    ...
    static int assignId() {
       int r = nextId;
       nextId ++;
       return r;
    }
    ...
    private String name = "";//显式赋值给字段
    private int id = assignId();//用静态方法初始化字段
}
```

4.8 对象构造

- 构造方法的参数名

```
public Employee(String n,double s) {
    name = n;
    salary = s;
}//方法一：使用单个字母做参数名

public Employee(String aName,double aSalary) {
    name = aName;
    salary = aSalary;
}//方法二：在字段名前加前缀“a”作为参数名

public Employee(String name,double salary) {
    this.name = name;
    this.salary = salary;
}//方法三：参数名与字段名相同
```

4.8 对象构造

- 调用其他构造器
  - 可以在构造器中的第一条语句使用this(...)的形式调用其他构造器，例如：

```
public Employee(double s)
{
   //该语句必须是第一条语句
   this("Employee #" + nextId,s);
   nextId ++;
}
//当调用new Employee(60000)时，Employee(double)构造器将
//调用Employee(String,double)构造器
```

4.8 对象构造

- 初始化块
  - 除了可以使用初始化语句和构造器对字段进行初始化外，还可以使用初始化块进行初始化

```
class Employee {
   public Employee(String n,double s) { name = n; salary = s; }
   public Employee() { name = "";salary = 0; }

   private int id;
   private static int nextId;
   //必须在初始化块之前定义字段
   {
      id = nextId;
      nextId ++;
   }
   private String name;
   private double salary;
}
```

4.8 对象构造

- 静态初始化块
  - 对于静态字段，可以通过提供初始值或使用静态初始化块进行初始化
    - 静态初始化块在类被第一次加载时被调用

```
class Employee {
    ...
    static int nextId = 1;
    static
    {
        Random generator = new Random();
        nextId = generator.nextInt(10000);
    }
    ...
}
```

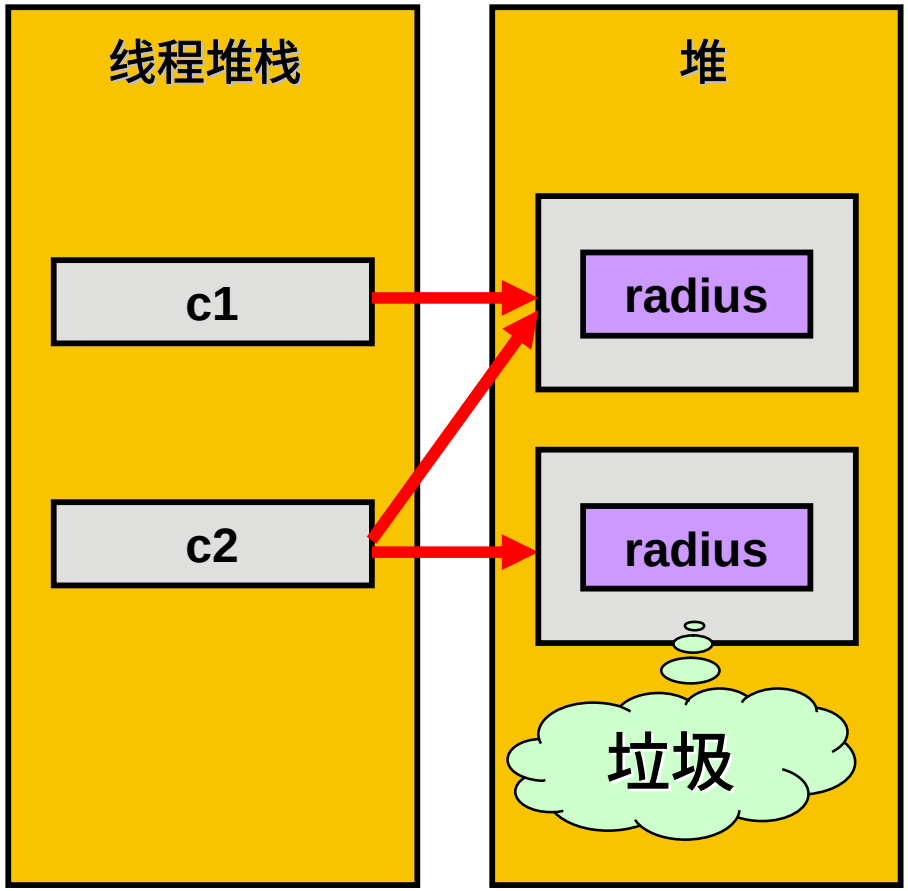
4.8 对象构造

- 构造对象过程中代码的执行过程
  - 参见教材例4-5：ConstructorTest.java

4.8 对象构造

- 使用new关键字创建对象过程分析

```
class Circle
{
  int radius;
}
class TestCircle
{
  void someMethod()
  {
     Circle c1 = new Circle();
     Circle c2 = new Circle();
     c2 = c1;
  }
}
```



4.8 对象构造

- 对象析构和 finalize 方法
  - 可以在任何类中添加finalize方法，该方法会在垃圾收集器清除对象之前被调用
  - 可以在该方法中释放一些系统资源，如：
    - 文件句柄
    - 数据库连接
    - 网络连接等
  - 不要依赖finalize方法回收任何短缺资源
    - 我们很难知道finalize方法什么时候被调用
  - 可以使用如下方法强制垃圾回收器工作

    System.gc();

4.8 对象构造

```
class FinalizeTest {
  public FinalizeTest() {
    System.out.println("构造对象");
  }
  public void connectToDB() {
    System.out.println("连接到数据库...");
  }
  public void finalize() {
    System.out.println("断开数据库...");
  }
  public static void main(String[] args) {
    FinalizeTest o = new FinalizeTest();
    o.connectToDB();
    o = null;
    System.gc();
    System.out.println("end...");
   }
}
```

4.9 变量的作用域

- 字段（成员变量）的作用域
  - 成员变量的作用域是整个类，类中的所有方法在规则允许的情况下都可以访问这些变量
- 局部变量的作用域
  - 方法内定义的变量以及方法的参数是局部变量
  - 局部变量的作用域从它的声明开始延续到包含它的块尾，即局部变量的作用域在其所属栈空间范围之内

4.10 包

- 什么是包？
  - 在Java程序中，每一个类和接口都包含在某个包（package）中
  - 包把程序合理的分类存放，使程序的文件结构更加清晰
  - 包是把类和接口聚集起来，以避免它们发生命名冲突的机制
- 标准 Java 库中常用的包
  - java.lang、java.util、java.net
  - java.awt、java.awt.event、java.applet
  - java.io、java.sql、javax.swing

4.10 包

- 使用包
  - 类可以使用其所在包中的所有类
  - 还可以使用其他包中的所有public类
    - 在每个类名前加上完整的包名

      ```
      java.util.Date today = new java.util.Date();
      ```
    - 使用import关键字

      ```
      import java.util.Date;
      Date today = new Date();
      ```
    - 可以使用*导入整个包中的类

      ```
      import java.util.*;
      //该语句只导入了util包下的类，并未导入其子包中的类
      ```

4.10 包

- 使用包
  - 名字冲突问题
    - 例如：java.util包和java.sql包中都有Date类，当程序需要同时引入这两个包时，可能有如下语句

      ```
      import java.util.*;
      import java.sql.*;
      ```

    - 使用Date类

      ```
      Date today;//产生编译错误
      ```

  - 解决办法
    - 单独导入欲使用的类或在每个类前加上完整的包名

4.10 包

- 包的声明
  - package语句

    package packagename;
  - 一个java源文件只能有一条package语句
  - 声明包的语句必须为Java源文件的第一条语句
    - 在一个Java源文件中如果没有package语句，则原文件中的类属于默认包，默认包没有包名
  - 包的名字可以是分层的，如：
    - package cn.edu.hit.software;//包名一般小写
    - 参见教材例4-6、例4-7
  - 编译时使用-d参数自动生成包对应的目录

4.10 包

- 虚拟机定位类的方法
  - jre/lib/和jre/lib/ext目录下档案文件中存放的类
  - 按classpath所指定的路径依次查找
    - 可以为解释器指定classpath参数
    - 也可以使用环境变量
- 编译器定位类的方法
  - 询问所有import指示，察看其中是否包含了要找的类
    - 首先查找java.lang包中的类
    - 再依次查找程序中显式用import引入的包中的类

4.10 包

- 包作用域
  - public成员可以被任何类使用
  - private成员只可以被定义它们的成员使用
  - 如果没有任何修饰符的成员则可以被同一包中的类访问，这样的成员具有包作用域
- 包存放的位置
  - 同一个包中的类一般存放在同一个目录下，也可以存放在不同的目录下，只要相对于classpath设置的路径的目录一样就可以

4.11 文档注释

- javadoc 工具
  - 使用java SDK中的javadoc工具，可以从java源文件生成HTML文档，源文件中以/**和*/包含的注释部分会被生成到HTML文档中
- javadoc 工具从以下几项中提取信息
  - 包
  - 公有类与接口
  - 公有方法和受保护方法
  - 公有字段和受保护字段

4.12 类设计技巧

- 一定要让数据私有
- 一定要初始化数据
- 不要在类中使用过多的基本类型
- 并非所有字段都需要独自的字段访问方法和更改方法
- 为类定义使用标准格式
- 分解职责太多的类
- 让类和方法的名字反映它们的职责

Any Question?