第3章 Java程序工程规范

3.1 为什么要有规范

软件开发是一个集体协作的过程，程序员之间的代码经常进行交换阅读，因此，Java 源程序有一些约定俗成的命名规定，主要目的是为了提高 Java 程序的可读性以及管理上的方便。好的程序代码应首先易于阅读，其次才是效率高低的问题。

3.2 Java 程序编程规范

(1) 有多个 import 语句时，先写 Java 包(都是 Java 包时，按照字母先后顺序排序)，后写 javax，最后写其他公司的包和自己定义的包。

(2) 命名规则为：

- 包名中的字母一律小写，如 xxxyyyzzz。
- 类名、接口名的每个单词的首字母大写，如 XxxYyyZzz。
- 方法名第一个单词的字母小写，后面每个单词的首字母大写，如 xxxYyyZzz。
- 常量中的每个字母大写，如 XXXYYYZZZ。

(3) 程序{}强调匹配的同时，要保持适当的缩进，以便于阅读。

(4) 必要时应有一定的程序注释量(20%～50%)，注释内容有：程序头说明，属性说明，方法说明。Java 中的注释共有两种方式：

- 多行注释：/* 文字或程序语句 */
- 单行注释：//文字或程序语句

注意：多行注释不能嵌套，即/* /* 文字或程序语句 */ */是非法的。

3.3 帮助文档的自动生成

Java 工程规范一方面体现在程序上，另一方面体现在程序帮助文档上，文档的规范和程序的规范同等重要。文档规范要求必须按照一定的书写格式以及与程序保持一致。然而，要真正实现这一目标并不容易，因为一定的书写格式虽然便于人们之间的沟通，但是却消耗了程序员宝贵的时间；文档与程序保持一致也并不容易，因为程序可能会经常修改，这

种修改并不都能及时反映到文档中。

要解决上述问题,没有一定的工具是不可能做到的,javadoc 工具就是用来解决这样的问题。javadoc 工具的引入,将程序员从枯燥、繁琐的工作中解放出来,程序员只要在书写程序时按照一定的要求书写注释,将来就可用 javadoc 自动生成帮助文档。程序员需要注意的规则如下:

(1) 程序头说明:注释为"/ ** 说明部分 * /",在说明部分一般包括文档的标题、描述、版权、作者、版本等信息。其中作者用"@author <作者>"的形式体现,内容和关键字之间用空格隔开。其他为:@version <版本>;@see <相关内容或类>;@since <本内容在哪个版本以后开始出现>。

(2) 方法说明:用于说明本方法的主要用途及实现的基本思路。属性信息有@param <属性名称> <参数说明> ;@return <返回值说明>;@exception <例外说明>;@throws <异常类>;@deprecated <功能逐渐被淘汰说明>。

如果按照上面的方式书写注释,则它们可以反映到帮助文档中。

【例 3.1】

```
import java.awt.*;
import java.applet.*;
/**
 * Title: 这是一个演示程序<br>
 * Description:用于说明 Applet 程序的典型特征<br>
 * @author 无名氏
 * @version 1.0
 */
public class HelloWorldApplet extends Applet{
    /** 初始化 */
    public void init(){
    }
    /** 用于绘制界面
     * @param g 为内部对象
     */
    public void paint(Graphics g){
      g.drawString("Hello World!",25,25);
    }
}
```

程序说明

的含义是在生成的 HTML 中换行;而@author 后不用写
的原因是这种属性可以自动换行。

使用 javadoc 工具按照如下方式书写并执行:

```
javadoc -d HelloWorldDoc -version -author HelloWorldApplet.java
```

-d 的含义是将所有生成的帮助文件全部放入本目录的子目录 HelloWorldDoc 下;-version 和-author 是在帮助文件中列出这方面的相关信息。生成的 index.html 如图 3.1 所示。

注意:如果想知道更多 javadoc,输入 javadoc help。

从图 3.1 可见,帮助文档一般的格式信息如下:

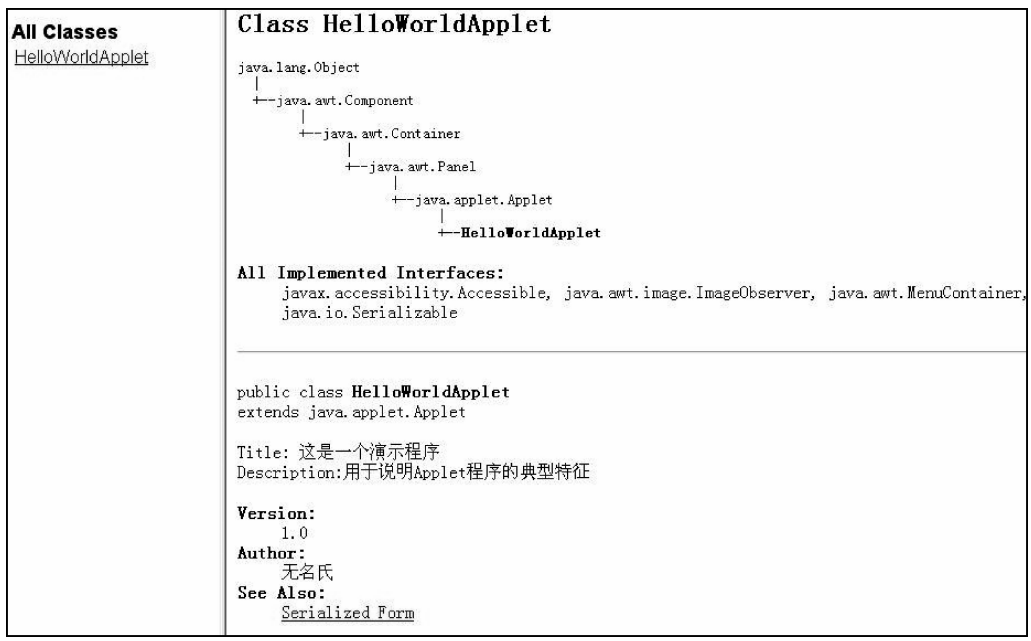

图　3.1

- 类的继承层次；
- 类和类的一般目的描述；
- 成员变量列表；
- 构造方法列表；
- 方法列表；
- 变量详细列表及目的和用途的描述；
- 构造方法详细列表及描述；
- 方法详细列表及描述。

小　结

介绍了Java程序的一些编程规范以及这些规范在工程中的作用。特别强调了程序注释的重要性和规范性。在此基础上，给出了利用注释规范生成帮助文档的方法。

习　题

1. 编写程序为什么要写注释？
2. Java都有哪些命名规范？
3. Javadoc生成帮助文档的时候，都有哪些命令行开关，它们的作用是什么？
4. 选择前两章的一道习题，加上适当的注释，并产生帮助文档。

第4章 面向对象(上)

4.1 抽象的含义

抽象(abstraction)是从被研究对象中舍弃个别的、非本质的或与研究主旨无关的次要特征,而抽取与研究有关的共性内容加以考察,形成对研究问题正确、简明扼要的认识。

图 4.1 很好地说明了抽象的含义:青蛙、狮子、马虽然不同,但人可以从中抽象出一个共同的概念——动物。

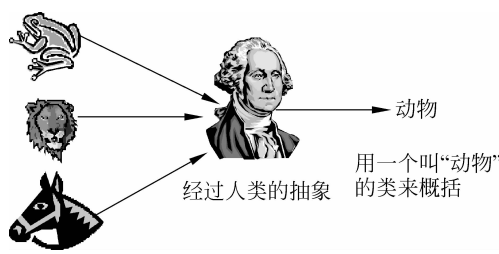

图 4.1

注意:计算机和哲学世界的抽象并不是人们通常所理解的"难"的意思,下面一段英语很好地反映了西方人对抽象和难的理解差异。

Assembly language is too difficult to understand,but not abstract。

汇编语言很难,不容易理解,但不抽象。

4.2 类与对象

对象是对客观事物的抽象,类是对对象的抽象。类是一种抽象的数据类型,其定义为:

```
class  类名{
}
```

它们的关系是,对象是类的实例,类是对象的模板。图 4.2 很好地说明了它们之间的关系。所有的战斗机都是按照一个图纸设计出来的,其中一个飞机改装后不会对其他飞机造成影响,但如果修改图纸,则会影响到以后生产出来的所有飞机。

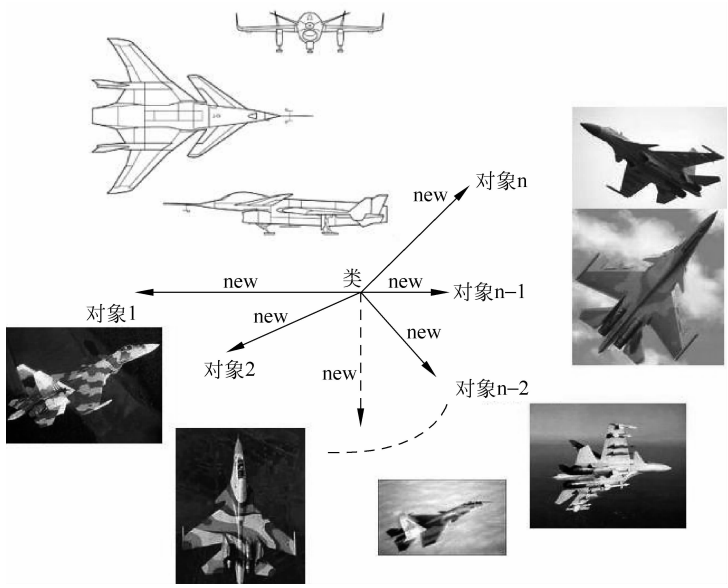

图　4.2

4.3　类的域(属性)与方法(操作)

类和对象都有域和方法。域(也称属性或是数据成员)是事物静态特征的抽象。在下面的例 4.1 中,name 和 missileNum 就是类 FighterPlane 的域变量。定义域的一般方式是:

类型　域变量名

方法(也称操作或成员方法)是事物动态特征的抽象,在下面的例 4.1 中,fire()就是 FighterPlane 的方法。方法定义的一般形式如下:

[修饰符] 返回值类型 方法名 (参数类型 参数 1,[参数类型 参数 2]…){
**　方法体**
}

【例 4.1】

```
class FighterPlane{
   String name;
   int missileNum;
   void fire(){
       if (missileNum > 0){
         missileNum -= 1 ;
         System.out.println("now fire a missile !");
       }
       else{
         System.out.println("No missile left !");
       }
   }
}
```

类中的域变量和方法存在以下关系：

- 类中定义的域变量可以被类中的所有方法访问。
- 方法中的形式参数和定义的局部变量的作用域仅限于方法，局部变量在使用前必须进行赋值初始化。如果局部变量和类中的域变量重名，则在方法中对同名变量改变的是局部变量。例如：

```
class FighterPlane1{
   String name = "su35";
   int missileNum;
   void init(String _name){
       String name = _name;
       System.out.println(name);
       System.out.println(this.name);
   }
}
```

程序说明

(1) init 中的形式参数_name 的作用范围只是在 init 方法中。

(2) init 中定义的局部变量 name，它和类的域变量 name 不同，通过_name 赋值改变的 name 是局部变量，而不是域变量。

(3) 在 init 方法当中，如果要使用类的域变量，则需要引入 this 关键字。

- 类中定义的方法可以进行递归调用。

【例 4.2】 求斐波那契数列(1,1,2,3,5,8,13,…)的第 10 项。

```
public class Fibonacci{
   public static int fseq(int n){
       if(n<1)  return -1;                    //进行参数校验
       if(n == 1 || n == 2) return 1;
       else return fseq(n-1) + fseq(n-2);
   }
   public static void main(String args[]){
       System.out.println(fseq(10));
   }
}
```

输出结果为 55。

程序说明

(1) 递归问题的动机：对于一个较为复杂的问题，把原问题分解成为若干相对简单且雷同的子问题，并且子问题的解决同父问题解决方法类似，如此进行分解，直到这样的子问题可以直接求解，这种情况可以使用递归。对于本例，f(n)＝f(n－1)＋f(n－2)，f(1)＝1，f(2)＝1 就满足这样的条件，而“if(n==1 || n==2) return 1;”就是程序的递归出口。

(2) 递归程序直接或间接地调用自身。对于系统而言，这和 A 方法调用 B 方法没有什么区别，其原理如图 4.3 所示。递规过程是程序的依次调用，由不断入栈、得到结果、不断出栈三个重要步骤组成。

(3) 当递归的规模很大时，例如需要求出本例第 50 万项的值，如用递归，则由于不断压

栈,很容易造成内存空间不足而溢出。因此,这种情况的求解,往往需要将递归转为循环进行求解。

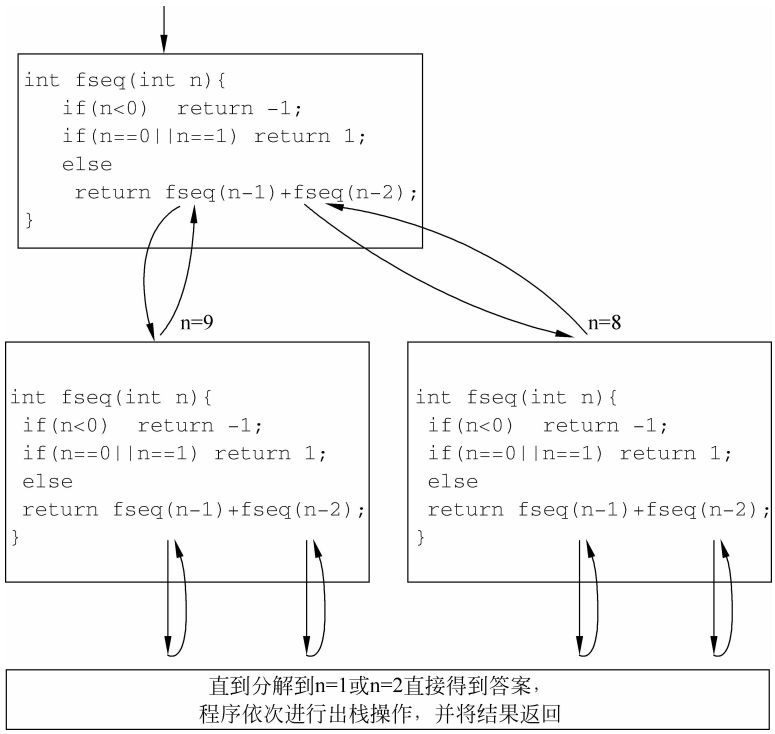

图　4.3

4.4　对象

4.4.1　对象的创建

图 4.4 是根据类生成对象的一个示意图,左图表示战斗机的类,右图表示根据类产生的对象(对象名称下面有一个横线)。

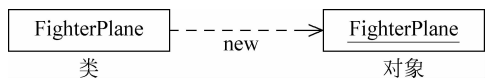

图　4.4

产生对象的代码是“new FighterPlane();”,这句代码的含义是:根据类模板产生一个对象,并在计算机内存中为此对象开辟一块新的独立内存空间。以类为模板产生对象,实质上就是将类中定义的属性复制到生成的对象当中,这些属性虽然在类中定义,实际上是为对象服务,因而称它们为对象属性;而方法在调用时,系统会为方法开辟一个栈空间,用于存放方法中的局部变量和形式参数,并且方法在执行时还能访问复制到对象中的属性,其效果

就如同方法也被复制到对象中一样。方法执行完毕后,栈空间被释放。虽然方法在类中定义,但从方法可以访问对象属性角度而言,类中定义的方法实际上是为对象而服务的,因而称为对象方法。

注意:需要注意的是,类中被 static 修饰的属性专属于类,并不复制到对象当中;被 static 修饰的方法也专属于类。static 方法调用时,也开辟栈空间存放局部变量,但方法不能访问任何复制到对象中的属性,只能访问专属于类的 static 属性。从这个意义上讲,static 修饰的方法是类方法,static 修饰的属性是类属性,详细内容见第 4.6.1 节。

有了对象,人们还希望对对象进行控制,就像人们通过遥控器控制电视机一样。现在电视已经有了,那么什么是遥控器呢? 下面的代码会起到一个遥控器的作用。

```
FighterPlane fp ;
```

图 4.5

注意:这句代码的作用是产生一个 FighterPlane 的声明(用图 4.5 来表示),此时并没有任何此类的对象产生,也没有为此对象分配内存空间。而 C++则不同,在 C++中,此时已产生了一个对象。

虽然 fp 相当于一个遥控器,但是此时它并不能遥控任何对象,需要为它指明遥控哪个对象。要做到这一点,需要下面的代码。

```
fp = new FighterPlane();
```

当然,也可以声明后立即赋予对象:"FighterPlane fp= new FighterPlane();",此时,上述代码可以用图 4.6 来描述。

图 4.6 说明,"FighterPlane fp= new FighterPlane();"代码的执行过程是先产生对象,之后将对象赋予声明 fp;此外,图 4.6 还说明对象和对象的声明不是一个概念,有声明时可以没有对象,正像有遥控器时可以没有电视机一样。当声明被赋予特定对象后,声明就被另一个概念——"引用"所取代。

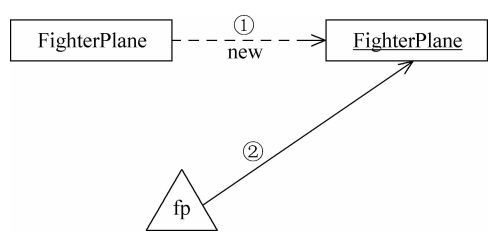

图 4.6

注意:引用从某种角度上讲,就好比对象的名片。

如果"FighterPlane fp = new FighterPlane();"这句代码处在某个类的方法当中,例如静态 main()方法,fp 和其引用的对象在内存中的实际分布又是什么样呢?

内存空间分为堆和栈(堆在应用程序生命周期内一直存在,而栈在方法调用完毕后就释放),类和对象被分配到堆中,而方法执行时用的局部变量和形式参数则放到栈空间当中。如果代码"FighterPlane fp = new FighterPlane();"处于静态 main 方法中,则代码中的 fp 处于静态 main 的栈空间当中,而产生的对象则被分配到堆当中。fp 作为引用能操控对象的原因在于:它在栈中保留了堆中对象的实际地址,图 4.7 表示的正是这种情况。

注意:C++中用关键字 new 方式产生的对象在堆中,而用"FighterPlane fp"方式产生的对象在方法栈中。

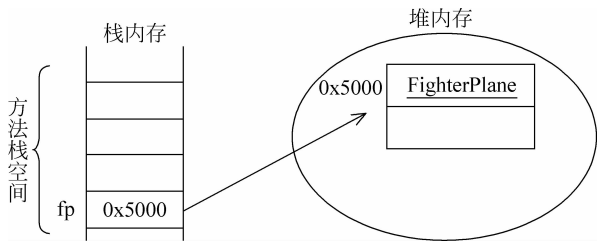

图　4.7

4.4.2　对象作为参数的特点

方法中的参数可以为基本数据类型,也可以为对象,它们有不同的特点。

基本数据类型作为参数在方法中的传递是值传递。

【例 4.3】

```
public class PassPara {
   private static int a;
   public static void main(String [] args) {
       modify(a);
       System.out.println(a);
  }
  public static void modify(int a) {
    a++;
  }
}
```

本程序的输出为 0。

程序说明

main()方法中的 a 是类 PassPara 的静态变量,初值为 0。当调用 modify()方法时,发生了参数传递,赋值给 modify 方法的形式参数 a,此 a 位于 modify 方法的栈空间当中。因为 a++是对形式参数进行自增,而不是对类静态变量 a 进行自增,当 modify 方法调用完毕后,栈空间被释放,形式参数 a 也被释放,无法回传给静态变量 a。

对象是引用传递,当对象作为参数传递时,传递的是对象的地址。也就是说,对象只有一个,例 4.4 和例 4.5 从不同角度体现了这一特点。

【例 4.4】

```
class IntClass{
  int value;
}
public class RunIntClass{
  public static void modifyValue(IntClass s,int val){
    s.value = val;
  }
  public static void main(String args[]){
    IntClass  a = new IntClass();
    modifyValue(a,8);
```

```
    System.out.println(a.value);
  }
}
```

程序说明

上面的程序在 main 方法中产生对象 a,在 modifyValue 中对 a 引用的对象进行属性赋值,之后又在 main 方法中显示 a 对象的属性。

当程序执行到"IntClass a = new IntClass()"时,产生了一个 IntClass 的对象,被 a 所引用,a 在 main 方法栈中,对象放到堆中,如图 4.8(a)所示。

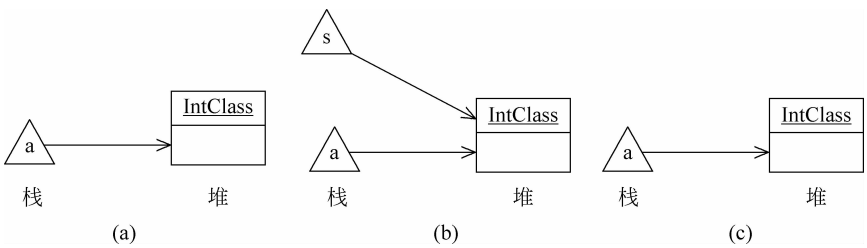

图 4.8

程序执行到"modifyValue(a,8)"时,在方法 modifyValue 的栈空间内分配了两个变量 s 和 val,其中 val 通过值传递的方式获得值。但 s 则不同,"s=a"的含义是将 a 的地址传递给 s,而不是产生了新的对象,这样 s 和 a 所指的对象是同一个(如图 4.8(b)所示)。此时,在方法 modifyValue 内部,通过 s 进行对象的赋值,改变了堆中引用对象的属性值。方法调用完毕后,modifyValue 方法所在的栈空间释放,此时 s、val 不复存在,但堆中的对象属性已经改变,虽然 s 不存在了,但 a 仍然引用了该对象(如图 4.8(c)所示)。

当程序执行到"System.out.println(a.value)"时,此时在 main 方法中,通过 a 操纵堆中的对象,显示其属性值。整个过程体现了对象作为参数传递时的引用特点。

【例 4.5】

```
class IntClass{
  int value;
}
public class RunIntClass{
   public static IntClass getInstance(){
      //在方法中产生对象
      IntClass s = new IntClass ();        //2
      s.value = 8;
      return  s;                           //引用返回
   }
   public static void main(String args[]){
      IntClass a ;                         //1
      a   = getInstance();                 //3
      System.out.println(a.value);
   }
}
```