%% %% This is file ‘zhwindowsfonts.tex’,%%generatedwiththedocstriputility.%%%%Theoriginalsourcefileswer

Thinking in Java chapter6 笔记和习题

emacsun

目录

1 简介

Initialization and Cleanup 这一章首先讲述类初始化的一些操作，顺带初始化操作阐述了方法重载。然后讲述了Java的Garbage Collector机制。其中关于 `static` 初始化的例子最为令人印象深刻，我把那个“橱柜”的代码做了逐行解析，学完之后感觉非常顺畅。

2 默认的 constructor 和带参数的 constructor

首先默认的 constructor ，看代码：

```
import java.util.*;
import static net.mindview.util.Print.*;
class Rock{
    Rock(){
        System.out.println("Rock");
    }
}
public class SimpleConstructor
{
    public static void main(String args[])
    {
        for (int i=0; i < 10 ;i++) {
```

```
            new Rock();
        }
    }
}
```

注意: constructor 是一个函数，其名称和 class 的名称必须相同(没有为什么，这是规定)。但是没有说明这个函数可不可以携带参数，实际上是可以的，继续看代码：

```
import java.util.*;
import static net.mindview.util.Print.*;
class Rock{
    Rock(int i){
        System.out.print("Rock" + i);
    }
}
public class SimpleConstructor2
{
    public static void main(String args[])
    {
        for (int i=0; i < 10 ;i++) {
            new Rock(i);
        }
    }
}
```

这段代码的输出是:

```
Rock 0 Rock 1 Rock 2 Rock 3 Rock 4 Rock 5 Rock 6 Rock 7 Rock 8 Rock 9
```

注意在这段代码中使用了 print 而不是 println . print 输出默认不带回车； println 输出默认带回车。

constructor 函数没有返回值，注意这里的没有返回值和 void 函数是两回事。

String 对象初始化值是 null, 看代码：

```
import java.util.*;
import static net.mindview.util.Print.*;
class Rock{
    String str;
}
public class Exercise0601
{
    public static void main(String args[])
    {
        Rock rcok = new Rock();
        print("" + rcok.str);
    }
}
```

其输出为

```
null
```

3 从 constructor 的定义引入函数重载

作者从 constructor 过度到另一个知识点 overload，平滑自然。对于重载，值得注意的是 primitive 类型的重载。看代码：

```
import static net.mindview.util.Print.*;
public class PrimitiveOverloading{
    void f1(char x){printnb("f1(char) ");}
    void f1(byte x){printnb("f1(byte) ");}
    void f1(short x){printnb("f1(short) ");}
    void f1(int x){printnb("f1(int) ");}
    void f1(long x){printnb("f1(long) ");}
    void f1(float x){printnb("f1(float) ");}
    void f1(double x){printnb("f1(double) ");}

    void f2(byte x){printnb("f2(byte) ");}
    void f2(short x){printnb("f2(short) ");}
    void f2(int x){printnb("f2(int) ");}
    void f2(long x){printnb("f2(long) ");}
    void f2(float x){printnb("f2(float) ");}
    void f2(double x){printnb("f2(double) ");}

    void f3(short x){printnb("f3(short) ");}
    void f3(int x){printnb("f3(int) ");}
    void f3(long x){printnb("f3(long) ");}
    void f3(float x){printnb("f3(float) ");}
    void f3(double x){printnb("f3(double) ");}

    void f4(int x){printnb("f4(int) ");}
    void f4(long x){printnb("f4(long) ");}
    void f4(float x){printnb("f4(float) ");}
    void f4(double x){printnb("f4(double) ");}

    void f5(long x){printnb("f5(long) ");}
    void f5(float x){printnb("f5(float) ");}
    void f5(double x){printnb("f5(double) ");}

    void f6(float x){printnb("f6(float) ");}
    void f6(double x){printnb("f6(double) ");}

    void f7(double x){printnb("f7(double) ");}

    void testConstVal(){
        printnb("5: ");
        f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);print();
    }

    void testChar(){
```

```
        char x = 'x';
        printnb("char: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
    }

    void testByte(){
        byte x = 0;
        printnb("byte: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
    }

    void testShort(){
        short x = 0;
        printnb("short: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
    }

    void testInt(){
        int x = 0;
        printnb("int: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
    }

    void testLong(){
        long x = 0;
        printnb("long: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
    }

    void testFloat(){
        float x = 0;
        printnb("float: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
    }

    void testDouble(){
        double x = 0;
        printnb("double: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);print();
    }

    public static void main(String[] args){
        PrimitiveOverloading p = new PrimitiveOverloading();
        p.testConstVal();
        p.testChar();
        p.testByte();
        p.testShort();
```

```
        p.testInt();
        p.testLong();
        p.testFloat();
        p.testDouble();
    }
}
```

这段代码的输出是:

```
5: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
char: f1(char) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
byte: f1(byte) f2(byte) f3(short) f4(int) f5(long) f6(float) f7(double)
short: f1(short) f2(short) f3(short) f4(int) f5(long) f6(float) f7(double)
int: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
long: f1(long) f2(long) f3(long) f4(long) f5(long) f6(float) f7(double)
float: f1(float) f2(float) f3(float) f4(float) f5(float) f6(float) f7(double)
double: f1(double) f2(double) f3(double) f4(double) f5(double) f6(double) f7(double)
```

4 默认的 constructor

默认的 constructor 是没有参数的。如果定义类时没有指定构造函数，那么编译器会生成一个默认的构造函数。如果在定义类时指定了构造函数，那么在创建该类的对象时就需要指定该对象实用的构造函数，而不能使用默认构造函数，否则就会报错。看代码：

```
1  class Bird{
2      Bird(int i){}
3      Bird(double d){}
4  }
5  public class NoSynthesis{
6      public static void main(String[] args){
7          Bird b2 = new Bird(1);
8          Bird b3 = new Bird(1.0);
9          Bird b4 = new Bird();
10     }
11 }
```

这个代码会报错：

```
error: no suitable constructor found for Bird(no arguments)
        Bird b4 = new Bird();
                  ^
    constructor Bird.Bird(int) is not applicable
      (actual and formal argument lists differ in length)
    constructor Bird.Bird(double) is not applicable
      (actual and formal argument lists differ in length)
1 error
```

编译器会认为没有无参数的构造函数定义。对代码进行修改：

```
1  class Bird{
2      Bird(int i){}
3      Bird(double d){}
4  }
5  public class NoSynthesis{
6      public static void main(String[] args){
7          Bird b2 = new Bird(1);
```

```
8           Bird b3 = new Bird(1.0);
9       }
10  }
```

就没有报错。但是，看代码：

```
1   class Bird{
2       Bird(int i){}
3       Bird(double d){}
4   }
5   public class NoSynthesis{
6       public static void main(String[] args){
7           Bird b2 = new Bird(1);
8           Bird b3 = new Bird(1.0);
9           Bird b4;
10      }
11  }
```

这个代码也没有报错，但是我不知道 b4 调用了那个构造函数。为了确认一下，对代码做如下修改：

```
1   class Bird{
2       Bird(int i){
3           System.out.println("with␣int␣i");
4       }
5       Bird(double d){
6           System.out.println("with␣double␣d");
7       }
8   }
9   public class NoSynthesis{
10      public static void main(String[] args){
11          Bird b2 = new Bird(1);
12          Bird b3 = new Bird(1.0);
13          Bird b4;
14      }
15  }
```

输出为:

```
with int i
with double d
```

可见 Bird b4 没有调用给出的两个构造函数，而是用的默认的构造函数。

5 this 的作用

简而言之， this 用来代表当前的对象。在使用的过程中，你完全可以用 this 来替代当前的对象。但是 this 的实用也会有一些限制，比如只能在 non-static 的方法中使用。但是在一个类的多个方法中不需要显示的使用 this 来指示当前类。

this 的一个经常用到的地方是 return 语句返回一个对象。看代码：

```
1   public class Leaf{
2       int i=0;
3       Leaf increament(){
4           i++;
5           return this;
6       }
7       void print(){
8           System.out.println("␣i␣=␣" + i);
9       }
10      public static void main(String[] args){
11          Leaf x = new Leaf();
12          x.increament().increament().print();
13      }
14  }
```

结果输出为：

```
i = 2
```

this 也可以用来把当前的对象传递给另外的方法，看代码：

```
class Person{
    public void eat(Apple apple){
        Apple peeled = apple.getPeeled();
        System.out.println("Yummy");
    }
}

class Peeler{
    static Apple peel(Apple apple){
        //...remove peel
        return apple;
    }
}

class Apple{
    Apple getPeeled(){return Peeler.peel(this);}
}

public class PassingThis{
    public static void main(String[] args){
        new Person().eat(new Apple());
    }
}
```

this 还可以用来从一个 constructor 中调用另一个 constructor 。这一用途有两点需要注意：

1. 你不能在一个 constructor 中调用两次 this 初始化函数。
2. 在一个 constructor 中，如果要使用 this ，第一行有效代码就应该是使用 this 的代码。

6 理解 static

有了 this 我们现在可以更深刻的理解 static 。我们可以从 non-static 函数里调用 static 函数，但是不能从 static 函数里调用 non-static 函数。为什么？因为在 static 函数里没有对象的概念。 static 函数依赖于 class 的定义存在，而不依赖于对象的存在。所以 static 看起来就像是一个全局方法，不依赖于对象存在。但是 Java 中是没有全局函数的，所以通过 static 可以实现类似的效果。

正是因为 static 的这个特性，人们诟病 Java 的 static 方法不是面向对象的。因为在 static 方法中，无法像一个对象发消息，因为根本没有 this 。因此当你的代码中有很多 static 的时候，你要重新审视一下你的代码结构。但是在很多时候 static 又是一个不得不存在的特性，在以后的章节中就会看到。

7 class 成员初始化

Java 中对于 class 的基础类型成员都做了默认初始化。这说明，每一个基础类成员都有一个默认的构造函数，为其赋初值。看代码：

```
import static net.mindview.util.Print.*;

public class  InitialValues{
    boolean t;
    char c;
    byte b;
    short s;
    int i;
    long l;
    float  f;
```

```
    double d;
    InitialValues reference;
    void printInitiaValues(){
        print("Data␣type␣␣␣␣␣␣␣␣␣initial␣values");
        print("boolean␣␣␣␣␣␣␣␣␣␣␣␣"+t);
        print("char␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+c);
        print("int␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+i);
        print("long␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+l);
        print("float␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+f);
        print("double␣␣␣␣␣␣␣␣␣␣␣␣␣"+d);
        print("InitialValues␣␣␣␣␣␣"+reference);
    }
    public static void main(String[] args){
        InitialValues iv = new InitialValues();
        iv.printInitiaValues();
    }

}
```

其输出为：

```
Data type          initial values
boolean            false
char               [ ]
int                0
long               0
float              0.0
double             0.0
InitialValues      null
```

char 的初始值是0，打印出来是一个空格。另外需要注意的是：如果在 class 中定义了一个对象而没有为其赋初值，则这个对象的 reference 会被赋值 null

8 初始化顺序

在 Java 中，类变量的初始化先于类方法调用。什么意思？就是说：在写代码的时候，即便你把类变量的定义放在了构造函数的后面， Java 依然会先初始化这些变量，然后再去调用函数（这个函数通常是构造函数）。看代码：

```
import static  net.mindview.util.Print.*;
class Window{
    Window(int marker){
        print("Window␣number:␣" + marker);
    }
}
class House{
    Window w1 = new Window(1);
    House(){
        print("House()");
        w3 = new Window(33);
    }
    Window w2 = new Window(2);
    void f(){
        print("f()");
    }
    Window w3 = new Window(3);
}
public class OrderOfInitialization
{
    public static void main(String args[])
    {
        House h = new House();
```

```
24          h.f();
25      }
26  }
```

输出为：

```
Window number: 1
Window number: 2
Window number: 3
House()
Window number: 33
f()
```

从代码中我们可以看到， House 的构造函数在 w1 和 w2 之间。但是我们执行 House h = new House() 这一行代码时，初始化的执行顺序是：

1. 初始化 w1
2. 初始化 w2
3. 初始化 w3
4. 调用 House() 对 w3 再次初始化。

9 static 类型的初始化

无论创建了多少个对象， static 类型的数据都只占用一份存储。只有 class 的域可以是 static ，一个本地变量不能是 static 类型的。接下来我们通过一个例子来查看 static 是如何初始化的, 看代码:

```
1  // specifying initial values in a class definition
2  import static net.mindview.util.Print.*;
3
4  class Bowl{
5      Bowl(int marker){
6          print("Bowl(" + marker + ")");
7      }
8      void f1(int marker){
9          print("f1(" + marker + ")");
10     }
11 }
12
13 class Table{
14     static Bowl bowl =new Bowl(1);
15     Table(){
16         print("Table()");
17         bowl2.f1(1);
18     }
19     void f2(int marker){
20         print("f2(" + marker + ")");
21     }
22     static Bowl bowl2 = new Bowl(2);
23 }
24
25 class Cupboard{
26     Bowl bowl3 = new Bowl(3);
27     static Bowl bowl4 = new Bowl(4);
28     Cupboard(){
29         print("Cupboard()");
30         bowl4.f1(2);
31     }
32     void f3(int marker){
33         print("f3(" + marker + ")");
```

```
34      }
35      static Bowl bowl5 =new Bowl(5);
36  }
37  public class StaticInitialization
38  {
39      public static void main(String args[])
40      {
41          print("Creating␣new␣Cupboard()␣in␣main");
42          new Cupboard();
43          print("Creating␣new␣Cupboard()␣in␣main");
44          new Cupboard();
45          table.f2(1);
46          cupboard.f3(1);
47      }
48      static Table table = new Table();
49      static Cupboard cupboard = new Cupboard();
50  }
```

这段代码是我目前敲过的最长的 Java代码，其输出也最长，看输出:

```
Bowl(1)
Bowl(2)
Table()
f1(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f1(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f1(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f1(2)
f2(1)
f3(1)
```

让我们来仔细分析一下每一行的输出是怎么来的。通过 Bowl 我们可以知道初始化的顺序。当我们调用 main() 时，首先初始化的是 StaticInitialization 中的 static 成员，然后是 non-static 成员。在这个代码中是先初始化最后两行的 table 和 cupboard 。

在初始化 table 过程中，生成了前四行输出。具体过程是：先初始化 Table 的 bowl1 和 bowl(2) 然后调用构造函数 Table() 生成第三行第四行输出。

在初始化 cupboard 过程中，生成了接下来的五行输出。具体过程是：先初始化 bowl(4) 和 bowl(5) 然后初始化 bowl3 最后调用 Cupboard() 构造函数。

table 和 cupboard 初始化结束后，接下来执行第 41 行打印了一句提示，然后执行第42 行，这个时候由于 bowl4 和 bowl5 已经被初始化了，所以只初始化了 bowl3 并调用了 Cupboard() 构造函数。

然后执行第 43 行，同样的只初始化了 bowl3 并调用了 Cupboard() 构造函数。

最后调用 table.f2(2) 和 cupboard.f3(1) 生成最后两行输出。