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：

```
1  class Person{
2      public void eat(Apple apple){
3          Apple peeled = apple.getPeeled();
4          System.out.println("Yummy");
5      }
6  }
7
8  class Peeler{
9      static Apple peel(Apple apple){
10         //...remove peel
11         return apple;
12     }
13 }
14
15 class Apple{
16     Apple getPeeled(){return Peeler.peel(this);}
17 }
18
```

```
19 public class PassingThis{
20     public static void main(String[] args){
21         new Person().eat(new Apple());
22     }
23 }
```

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：

```
1 import static net.mindview.util.Print.*;
2
3 public class  InitialValues{
4     boolean t;
5     char c;
6     byte b;
7     short s;
```

```
8       int i;
9       long l;
10      float  f;
11      double d;
12      InitialValues reference;
13      void printInitiaValues(){
14          print("Data␣type␣␣␣␣␣␣␣␣␣initial␣values");
15          print("boolean␣␣␣␣␣␣␣␣␣␣␣␣"+t);
16          print("char␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+c);
17          print("int␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+i);
18          print("long␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+l);
19          print("float␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+f);
20          print("double␣␣␣␣␣␣␣␣␣␣␣␣␣"+d);
21          print("InitialValues␣␣␣␣␣␣"+reference);
22      }
23      public static void main(String[] args){
24          InitialValues iv = new InitialValues();
25          iv.printInitiaValues();
26      }
27
28  }
```

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：

```
1   import static  net.mindview.util.Print.*;
2   class Window{
3       Window(int marker){
```


```
4           print("Window number: " + marker);
5       }
6   }
7   class House{
8       Window w1 = new Window(1);
9       House(){
10          print("House()");
11          w3 = new Window(33);
12      }
13      Window w2 = new Window(2);
14      void f(){
15          print("f()");
16      }
17      Window w3 = new Window(3);
18  }
19  public class OrderOfInitialization
20  {
21      public static void main(String args[])
22      {
23          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。

10 显示初始化

在 Java 中可以使用 static 语句初始化（有时候我们叫之中初始化方式为 static 块）。看代码：

```
1 public class Spoon{
2     static int i;
3     static{
4         i = 47;
5     }
6 }
```

看起来像是一个方法，但是注意这种初始化方法仅仅是一个 static 关键词跟着一个语句块。同其他 static 初始化语句一样，使用 static 块的初始化也仅仅执行一次。看代码：

```
1  class Cup{
2    Cup(int marker){
3      print("Cup(" + marker + ")");
4    }
5    void f(int marker){
6      print("f(" + marker + ")");
7    }
8  }
9
10 class Cups{
11   static Cup cup1;
12   static Cup cup2;
13   static{
14     cup1 = new Cup(1);
15     cup2 = new Cup(2);
16   }
17   Cups(){
18     print("Cups()");
19   }
20 }
21
22 public class ExplicitStatic{
23   public static void main(String[] args){
24     print("Inside␣main()");
25     Cups.cup1.f(99);                              (cupsrun)
26   }
27   static Cups cups1 = new Cups();                 (cupsrun1)
28 }
```

输出为：

```
Inside main()
```

```
Cup(1)
Cup(2)
f(99)
```

`static` 初始化语句在下列任一情况下初始化 `Cups`：

1. 第cupsrun 行执行。
2. 第cupsrun 被注释，第cupsrun1 行解注；

在情况1，我们访问了 `Cups` 的成员变量，这个时候出发了 `Cups` 类的初始化；在情况2，我们初始化了对象 `cups1` 。

当我们把第 cupsrun1行解注，第cupsrun 注释掉之后，输出为：

```
Cup(1)
Cup(2)
Cups()
Inside main()
```

可以看到在类 `ExplicitStatic` 中，也是 `static` 变量先初始化。我们看到 `Inside main()` 最后输出。说明程序先完成了静态成员 `cups1` 的初始化之后再调用的 `main` 函数。

如果我把第 cupsrun1行和第cupsrun 行都解注，则输出：

```
Cup(1)
Cup(2)
Cups()
Inside main()
f(99)
```

可以看出还是 `static` 变量优先，通过第 cupsrun1行对 `Cups` 类的静态变量进行了初始化。然后再执行 `main` 函数。

问题 *Create a class with a* `static String` *field that is initialized at the point of definition, and another one that is initialized by the* `static` *block. Add a* `static` *method that prints both fields and demonstrates that they are both initialized before they are used.*

解答： 首先编写代码，如下

```
1  import static net.mindview.util.Print.*;
2
3  class String1{
4      static String str1 = "string in String1";
5      String1(){
6          print("" + str1);
7      }
8  }
9  class String2{
10     static String str2;
11     static{
12         str2 = "string in string2";
13     }
14     String2(){
15         print("" + str2);
16     }
17 }
18
19 public class Exercise0614{
20     public static void main(String[] args){
21         print("Inside main()");
22         String1 str1 = new String1();
23         String2 str2 = new String2();
24     }
25 }
```

输出为：

```
Inside main()
string in String1
string in string2
```

之所以按顺序初始化 *str1* 和 *str2*，是因为在 *Exercise0614* 的类中，这两个变量不是 *static* ,把这两个类改成 *static* 的有：

```
1  import static net.mindview.util.Print.*;
2
3  class String1{
4      static String str1 = "string in String1";
5      String1(){
6          print("" + str1);
7      }
8  }
9  class String2{
10     static String str2;
11     static{
12         str2 = "string in string2";
13     }
14     String2(){
15         print("" + str2);
16     }
17 }
```

```
18
19 public class Exercise0614{
20     public static void main(String[] args){
21         print("Inside main()");
22     }
23     static String1 str1 = new String1();
24     static String2 str2 = new String2();
25
26 }
```

输出为：

```
string in String1
string in string2
Inside main()
```

然后修改代码，为其添加静态函数（静态函数可以在不定义对象的时候调用，而我们调用静态函数的同时，对这个静态函数所属的类初始化，主要是初始化其 *static* 变量）。

```
1  import static net.mindview.util.Print.*;
2
3  class String1{
4      static String str1 = "string in String1";
5      String1(){
6          print("construct 1" + str1);
7      }
8      static void f1(){
9          print("static method" + str1);
10     }
11 }
12 class String2{
13     static String str2;
14     static{
15         str2 = "string in string2";
16     }
17     String2(){
18         print("constructor 2" + str2);
19     }
20     static void f2(){
21         print("static method" + str2);
22     }
23 }
24
25 public class Exercise0614{
26     public static void main(String[] args){
27         print("Inside main()");
28         String2.f2();
29         String1.f1();
30     }
31     static String2 str2 = new String2();
32 }
```

输出为：

```
constructor 2string in string2
Inside main()
static methodstring in string2
static methodstring in String1
```

11 非静态实例初始化

对于非静态变量， java提供了类似于 static 块的初始化方式。看代码：

```
1  import static net.mindview.util.Print.*;
2
3  class Mug{
4      Mug(int marker){
5          print("Mug("  + marker +")");
6      }
7      void f(int marker){
8          print("f(" + marker + ")");
9      }
10 }
11 public class Mugs{
12     Mug mug1;
13     Mug mug2;
14     {
15         mug1 = new Mug(1);
16         mug2 = new Mug(2);
17         print("mug1 & mug2 initialized");
18     }
19     Mugs(){
20         print("Mugs()");
21     }
22     Mugs(int i){
23         print("Mugs(int)");
24     }
25     public static void main(String[] args){
26         print("Inside main()");
27         new Mugs();
28         print("new Mugs() completed");
29         new Mugs(1);
30         print("new Mugs(1) completed");
31     }
32 }
```

输出为：

```
Inside main()
Mug(1)
Mug(2)
```

```
mug1 & mug2 initialized
Mugs()
new Mugs() completed
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs(int)
new Mugs(1) completed
```

从这个输出可以看出，实例可以按照顺序初始化，没有像 static 一样执行顺序和代码出现顺序不一样的情况。

12 数组初始化

数组是一些相同类型元素的集合。这些元素可以是基础类型也可以是某个类（这么说可能有些不严谨，基础类型也是类，在Java中一切都是类。）。

定义一个整型数组可以使用 int[] a1; 也可以用 int a1[] 。在 Java 中，数组的初始化可以通过大括号实现，也可以不初始化。 Java 中，数组名的赋值，复制的是引用。看代码：

```
1  import static net.mindview.util.Print.*;
2
3  public class ArrayOfPrimitives{
4      public static void main(String[] args){
5          int[] a1 = {1,2,3,4,5};
6          int[] a2;
7          a2 = a1;
8          for (int i = 0; i < a1.length; i++) {
9              a2[i] = a2[i]*2;
10         }
11         for (int i = 0; i < a1.length; i++) {
12             print("a1[" + i + "]= " + a1[i]);
13         }
14     }
15 }
```

输出是：

```
a1[0]= 2
a1[1]= 4
a1[2]= 6
a1[3]= 8
```

```
a1[4]= 10
```

可以看到执行了 a2=a1 之后， a2 指向的内容和 a1 指向的内容是一样的。这时候即使对 a2 进行修改， a1 的内容也同时发生了改变。

关于数组越界的问题在 C/C++ 和 Java 中有不同的处理方法： C/C++ 不会对数组越界进行约束，而 Java 会。在 Java 中，一旦数组越界就会报错。虽然适时的检查会不会报错是一件很低效的事情，但是为了开发效率， Java 的设计者认为这是值得的。

对于事先不知道长度的数组，需要用 new 来为其分配空间，看代码：

```
1  import java.util.*;
2  import static net.mindview.util.Print.*;
3
4  public class ArrayNew{
5      public static void main(String[] args){
6          int [] a;
7          Random rand = new Random(47);
8          a = new int[rand.nextInt(20)];
9          print("length␣of␣a␣=␣" + a.length );
10         print(Arrays.toString(a));
11     }
12 }
```

输出为：

```
length of a = 18
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

之前创建了 primitive 的数组。现在创建对象数组，当创建对象数组时，数组元素是对象的索引。现在考虑创建 Integer 数组。（回忆一下 Integer 是类不是primitive类型。基础类型和类的区别是，基础类型保存在栈上，而对象保存在堆上。）

```
1  import java.util.*;
2  import static net.mindview.util.Print.*;
3
4  public class ArrayClassObj{
5      public static void main(String[] args){
6          Random rand = new Random(47);
7          Integer[] a = new Integer[rand.nextInt(10)];
8          print(Arrays.toString(a));
9          print("length␣of␣a␣=␣" + a.length);
10         for (int i = 0; i < a.length; i++) {
11             a[i] = rand.nextInt(500);
12         }
13         print(Arrays.toString(a));
14     }
```

```
15 }
```

输出是：

```
[null, null, null, null, null, null, null, null]
length of a = 8
[55, 193, 361, 461, 429, 368, 200, 22]
```

从代码可以看出，没有初始化之前打印输出的都是 null 而不是像基础类型那样初始化为 0 ，只有初始化之后才会有数据打印。在创建对象数组时，即使使用 new 也没有初始化这个数组，必须一个一个来。

对于对象数组也可以用大括号进行初始化，看代码：

```
1  import java.util.*;
2
3  public class ArrayInit{
4      public static void main(String[] args){
5          Integer[] a = {
6              new Integer(1),
7              new Integer(2),
8              3,
9          };
10         Integer[] b = new Integer[]{
11             new Integer(1),
12             new Integer(2),
13             3,
14         };
15         System.out.println(Arrays.toString(a));
16         System.out.println(Arrays.toString(b));
17     }
18 }
```

输出为：

```
[1, 2, 3]
[1, 2, 3]
```

注意初始化对象数组时，最后一个元素后面的逗号。使用这个逗号写出的代码便于维护，尤其是维护长数组。可以用这种初始化方法传递变长参数。看代码：

```
1  class A{}
2
3  public class VarArgs{
4      static void printArray(Object[] args){
5          for (Object obj: args)
6              System.out.print(obj + "␣");
7          System.out.println();
```

```
8       }
9       public static void main(String[] args){
10          printArray(new Object[]{
11                  new Integer(47), new Float(3.14),new Double(11.11),
12              });
13          printArray(new Object[]{"one","two","three"});
14          printArray(new Object[]{new A(),new A(),new A()});
15      }
16  }
```

输出是：

```
47 3.14 11.11
one two three
A@15db9742 A@6d06d69c A@7852e922
```

上面代码的 Object 共有的根类，关于这个类，我们后面还会学到更多。因为所有类都是从 Object 继承而来，所以可以把 Object 数组传入一个方法。

另外我们还可以看到，print 函数接收一个 Object 的数组，然后用 foreach 的循环语法打印每个 Object 数组中 reference 对应的内容。打印标准的 Java 库类输出具有较强的可读性（看输出的前两行），打印自定义的类的输出（输出的第三行）就有点不知所云。不过从输出的第三行可以看出打印自定义类，其输出具有固定的格式，都是类名跟上 @ 然后是一串十六进制数（以后我们会了解到，这串十六进制数是对象地址）。

上面的这段代码在 Java SE5之前比较常见，但是自从 Java SE5之后，就有更方便的写法了：可以像 printArray() 那样打印数组（我现在写这个笔记的时候用的是SE8，所以浪费了一段时间学习了一个历史技术。）。

看代码：

```
1   public class NewVarArgs{
2       static void printArray(Object... args){
3           for (Object obj : args) {
4               System.out.print(obj + "␣");
5           }
6           System.out.println();
7       }
8       public static void main(String[] args){
9           printArray(new Integer(47),new Float(3.14),new Double(11.11));
10          printArray(47,3.14F,11.11);
11          printArray("one","two","three");
12          printArray(new A(),new A(),new A());
13          printArray((Object[])new Integer[]{1,2,3,4});
    (NewVarArgsline2)
14          printArray();//Empth list is also OK;
15      }
16  }
```

输出为：

```
47 3.14 11.11
47 3.14 11.11
one two three
A@15db9742 A@6d06d69c A@7852e922
1 2 3 4
```

开心的服下这颗语法糖。

当你使用 varargs 时，不在需要写明数组信息，编译器会自动帮你填写。编译器会给 print() 函数一个 array 。但是注意这里不仅仅是从一个元素列表到一个数组的转换。注意代码中第NewVarArgsline2 行，我们可以看到一个 Integer 数组转换成了一个 Object 数组。 *在编译过程中，编译器看到这是一个数组，并不会做转换* （对这句话我还没有深刻的理解，需要以后回头看）。

最后一行代码告诉我们，可以对一个 vararg 的列表传送零个参数。这个语法很有用，尤其是当函数的尾随参数个数可变时。关于尾随参数，看代码：

```
1  public class OptionalTrailingArguments{
2      static void f(int required, String... trailing){
3          System.out.print("required:␣" + required + "␣");
4
5          for(String s: trailing)
6              System.out.print(s + "␣");
7          System.out.println();
8      }
9      public static void main(String[] args){
10         f(1,"one");
11         f(2,"two","three");
12         f(0);
13     }
14 }
```

输出为：

```
required: 1 one
required: 2 two three
required: 0
```

从上面代码还可以看出，可以使用除了 Object 之外的类型作为 varargs 的类型（在这里例子中，我们实用的是 String ）。事实上，可以使用任何类型作为可变参数类型。看代码：

```
1  public class VarargType{
```


```
2       static void f(Character... args){
3           System.out.print(args.getClass());
4           System.out.println("␣length␣" + args.length);
5       }
6       static void g(int ... args){
7           System.out.print(args.getClass());
8           System.out.println("␣length␣" + args.length);
9       }
10      public static void main(String[] args){
11          f('a');
12          f();
13          g(1);
14          g();
15          System.out.println("int␣[]:␣" + new int[0].getClass());
16      }
17  }
```

输出为：

```
class [Ljava.lang.Character; length 1
class [Ljava.lang.Character; length 0
class [I length 1
class [I length 0
int []: class [I
```

从上面的代码可以看出，可以使用 Character 数组作为可变参数类型，也可以使用 int 这种基础类型作为可变参数类型。 getClass() 函数是 Object 的内置函数。