Java 基础：集合基础（1）

数组和第一类对象

无论使用的数组属于什么类型，数组标识符实际都是指向真实对象的一个句柄。那些对象本身是在内存“堆”里创建的。堆对象既可“隐式”创建（即默认产生），亦可“显式”创建（即明确指定，用一个 new 表达式）。堆对象的一部分（实际是我们能访问的唯一字段或方法）是只读的 length（长度）成员，它告诉我们那个数组对象里最多能容纳多少元素。对于数组对象，“ []”语法是我们能采用的唯一另类访问方法。

对象数组和基本数据类型数组在使用方法上几乎是完全一致的。唯一的差别在于对象数组容纳的是句柄，而基本数据类型数组容纳的是具体的数值。

```
public class ArraySize {
    public static void main(String[] args) {
        // Arrays of objects:
        Weeble[] a; // Null handle
        Weeble[] b = new Weeble[5]; // Null handles
        Weeble[] c = new Weeble[4];
        for (int i = 0; i < c.length; i++)
            c[i] = new Weeble();
        Weeble[] d = { new Weeble(), new Weeble(), new Weeble() };
        // Compile error: variable a not initialized:
        // !System.out.println("a.length=" + a.length);
        System.out.println("b.length = " + b.length);
        // The handles inside the array are
        // automatically initialized to null:
        for (int i = 0; i < b.length; i++)
            System.out.println("b[" + i + "]=" + b[i]);
        System.out.println("c.length = " + c.length);
        System.out.println("d.length = " + d.length);
        a = d;
        System.out.println("a.length = " + a.length);
        // Java 1.1 initialization syntax:
        a = new Weeble[] { new Weeble(), new Weeble() };
        System.out.println("a.length = " + a.length);
        // Arrays of primitives:
        int[] e; // Null handle
```

```
        int[] f = new int[5];
        int[] g = new int[4];
        for (int i = 0; i < g.length; i++)
            g[i] = i * i;
        int[] h = { 11, 47, 93 };
        // Compile error: variable e not initialized:
        // !System.out.println("e.length=" + e.length);
        System.out.println("f.length = " + f.length);
        // The primitives inside the array are
        // automatically initialized to zero:
        for (int i = 0; i < f.length; i++)
            System.out.println("f[" + i + "]=" + f[i]);
        System.out.println("g.length = " + g.length);
        System.out.println("h.length = " + h.length);
        e = h;
        System.out.println("e.length = " + e.length);
        // Java 1.1 initialization syntax:
        e = new int[] { 1, 2 };
        System.out.println("e.length = " + e.length);
    }
}
```

输出如下：

```
b.length = 5
b[0]=null
b[1]=null
b[2]=null
b[3]=null
b[4]=null
c.length = 4
d.length = 3
a.length = 3
a.length = 2
f.length = 5
f[0]=0
f[1]=0
f[2]=0
f[3]=0
f[4]=0
g.length = 4
```

```
h.length = 3
e.length = 3
e.length = 2
```

其中，数组 a 只是初始化成一个 null 句柄。此时，编译器会禁止我们对这个句柄作任何实际操作，除非已正确地初始化了它。数组 b 被初始化成指向由 Weeble 句柄构成的一个数组，但那个数组里实际并未放置任何 Weeble 对象。然而，我们仍然可以查询那个数组的大小，因为 b 指向的是一个合法对象。

换言之，我们只知道数组对象的大小或容量，不知其实际容纳了多少个元素。

尽管如此，由于数组对象在创建之初会自动初始化成 null，所以可检查它是否为 null，判断一个特定的数组“空位”是否容纳一个对象。类似地，**由基本数据类型构成的数组会自动初始化成零（针对数值类型）、 null（字符类型）或者 false（布尔类型）。**

数组 c 显示出我们首先创建一个数组对象，再将 Weeble 对象赋给那个数组的所有“空位”。数组 d 揭示出“集合初始化”语法，从而创建数组对象（用 new 命令明确进行，类似于数组 c），然后用 Weeble 对象进行初始化，全部工作在一条语句里完成。

下面这个表达式：

```
a = d;
```

向我们展示了如何取得同一个数组对象连接的句柄，然后将其赋给另一个数组对象，向我们展示了如何取得同一个数组对象连接的句柄，然后将其赋给另一个数组对象。

集合类只能容纳对象句柄。但对一个数组，却既可令其直接容纳基本类型的数据，亦可容纳指向对象的句柄。利用象 Integer、 Double 之类的“ 封装器”类，可将基本数据类型的值置入一个集合里。

无论将基本类型的数据置入数组，还是将其封装进入位于集合的一个类内，都涉及到执行效率的问题。显然，若能创建和访问一个基本数据类型数组，那么比起访问一个封装数据的集合，前者的效率会高出许多。

数组的返回

假定我们现在想写一个方法，同时不希望它仅仅返回一样东西，而是想返回一系列东西。此时，像 C 和 C++这样的语言会使问题复杂化，因为我们不能返回一个数组，只能返回指向数组的一个指针。这样就非常麻烦，因为很难控制数组的“存在时间”，它很容易造成内存“漏洞”的出现。

Java 采用的是类似的方法，但我们能“返回一个数组”。当然，此时返回的实际仍是指向数组的指针。但在 Java 里，我们永远不必担心那个数组的是否可用——只要需要，它就会自动存在。而且垃圾收集器会在我们完成后自动将其清除。

```
public class IceCream {
    static String[] flav = { "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
            "Mint Chip", "Mocha Almond Fudge", "Rum Raisin", "Praline Cream",
            "Mud Pie" };
```

```
    static String[] flavorSet(int n) {
        // Force it to be positive & within bounds:
        n = Math.abs(n) % (flav.length + 1);
        String[] results = new String[n];
        int[] picks = new int[n];
        for(int i = 0; i < picks.length; i++)
        picks[i] = -1;
        for(int i = 0; i < picks.length; i++) {
        retry:
        while(true) {
            int t =(int)(Math.random() * flav.length);
            for(int j = 0; j < i; j++)213
            if(picks[j] == t) continue retry;
            picks[i] = t;
            results[i] = flav[t];
            break;
            }
        }
        return results;
    }
    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {
            System.out.println("flavorSet(" + i + ") = ");
            String[] fl = flavorSet(flav.length);
            for (int j = 0; j < fl.length; j++)
                System.out.println("\t" + fl[j]);
        }
    }
}
```

flavorSet()方法创建了一个名为 results 的 String 数组。该数组的大小为 n—— 具体数值取决于我们传递给方法的自变量。随后，它从数组 flav 里随机挑选一些“香料”（ Flavor ），并将它们置入 results 里，并最终返回 results。返回数组与返回其他任何对象没什么区别—— 最终返回的都是一个句柄。

另一方面，注意当 flavorSet()随机挑选香料的时候，它需要保证以前出现过的一次随机选择不会再次出现。为达到这个目的，它使用了一个无限 while 循环，不断地作出随机选择，直到发现未在 picks 数组里出现过的一个元素为止（当然，也可以进行字串比较，检查随机选择是否在 results 数组里出现过，但字串比较的效率比较低）。若成功，就添加这个元素，并中断循环（ break ），再查找下一个（ i 值会递增）。但假若 t 是一个已在 picks 里出现过的数组，就用标签式的 continue 往回跳两级，强制选择一个新 t。 用一个调试程序可以很

清楚地看到这个过程。

集合

为容纳一组对象，最适宜的选择应当是数组。而且假如容纳的是一系列基本数据类型，更是必须采用数组。

缺点：类型未知

使用 Java 集合的“缺点”是在将对象置入一个集合时丢失了类型信息。之所以会发生这种情况，是由于当初编写集合时，那个集合的程序员根本不知道用户到底想把什么类型置入集合。若指示某个集合只允许特定的类型，会妨碍它成为一个“常规用途”的工具，为用户带来麻烦。为解决这个问题，集合实际容纳的是类型为 Object 的一些对象的句柄。

当然，也要注意集合并不包括基本数据类型，因为它们并不是从“任何东西”继承来的。

Java 不允许人们滥用置入集合的对象。假如将一条狗扔进一个猫的集合，那么仍会将集合内的所有东西都看作猫，所以在使用那条狗时会得到一个“违例”错误。在同样的意义上，假若试图将一条狗的句柄“造型”到一只猫，那么运行期间仍会得到一个“违例”错误。

```
class Cat {
    private int catNumber;
    Cat(int i) {
        catNumber = i;
    }
    void print() {
        System.out.println("Cat #" + catNumber);
    }
}
class Dog {
    private int dogNumber;
    Dog(int i) {
        dogNumber = i;
    }
    void print() {
        System.out.println("Dog #" + dogNumber);
    }
}
public class CatsAndDogs {
    public static void main(String[] args) {
        Vector cats = new Vector();
        for (int i = 0; i < 7; i++)
            cats.addElement(new Cat(i));
```

```
        // Not a problem to add a dog to cats:
        cats.addElement(new Dog(7));
        for (int i = 0; i < cats.size(); i++)
            ((Cat) cats.elementAt(i)).print();
        // Dog is detected only at run-time
    }
}
```

- 错误有时并不显露出来

在某些情况下，程序似乎正确地工作，不造型回我们原来的类型。第一种情况是相当特殊的： String 类从编译器获得了额外的帮助，使其能够正常工作。只要编译器期待的是一个 String 对象，但它没有得到一个，就会自动调用在 Object 里定义、并且能够由任何 Java 类覆盖的 toString()方法。这个方法能生成满足要求的 String 对象，然后在我们需要的时候使用。因此，为了让自己类的对象能显示出来，要做的全部事情就是覆盖 toString()方法。

```
class Mouse {
    private int mouseNumber;
    Mouse(int i) {
        mouseNumber = i;
    }
    // Magic method:
    public String toString() {
        return "This is Mouse #" + mouseNumber;
    }
    void print(String msg) {
        if (msg != null)
            System.out.println(msg);
        System.out.println("Mouse number " + mouseNumber);
    }
}
class MouseTrap {
    static void caughtYa(Object m) {
        Mouse mouse = (Mouse) m; // Cast from Object
        mouse.print("Caught one!");
    }
}
public class WorksAnyway {
    public static void main(String[] args) {
        Vector mice = new Vector();
        for(int i = 0; i < 3; i++)
            mice.addElement(new Mouse(i));
```

```
        for(int i = 0; i < mice.size(); i++) {
            // No cast necessary, automatic call
            // to Object.toString():
            System.out.println(
            "Free mouse: " + mice.elementAt(i));
            MouseTrap.caughtYa(mice.elementAt(i));
            }
        }
}
```

可在 Mouse 里看到对 toString()的重定义代码。在 main()的第二个 for 循环中，可发现下述语句：

```
System.out.println("Free mouse: " +
mice.elementAt(i));
```

在“ +”后 ,编译器预期看到的是一个 String 对象。elementAt()生成了一个 Object，所以为获得希望的 String ,编译器会默认调用 toString()。但不幸的是 ,只有针对 String 才能得到象这样的结果；其他任何类型都不会进行这样的转换。

隐藏造型的第二种方法已在 Mousetrap 里得到了应用。 caughtYa()方法接收的不是一个 Mouse，而是一个 Object。随后再将其造型为一个 Mouse。当然，这样做是非常冒失的 ,因为通过接收一个 Object ,任何东西都可以传递给方法。然而 ,假若造型不正确——如果我们传递了错误的类型—— 就会在运行期间得到一个违例错误。这当然没有在编译期进行检查好，但仍然能防止问题的发生。注意在使用这个方法时毋需进行造型：

```
 MouseTrap.caughtYa(mice.elementAt(i));
```

- 生成能自动判别类型的 Vector

一个更“健壮”的方案是用 Vector 创建一个新类，使其只接收我们指定的类型，也只生成我们希望的类型。

```
class Gopher {
    private int gopherNumber;
    Gopher(int i) {
        gopherNumber = i;
    }
    void print(String msg) {
        if (msg != null)
            System.out.println(msg);
        System.out.println("Gopher number " + gopherNumber);
    }
}
class GopherTrap {
    static void caughtYa(Gopher g) {
```

```
        g.print("Caught one!");
    }
}
class GopherVector {
    private Vector v = new Vector();
    public void addElement(Gopher m) {
        v.addElement(m);
    }
    public Gopher elementAt(int index) {
        return (Gopher) v.elementAt(index);
    }
    public int size() {
        return v.size();
    }
    public static void main(String[] args) {
        GopherVector gophers = new GopherVector();
        for (int i = 0; i < 3; i++)
            gophers.addElement(new Gopher(i));
        for (int i = 0; i < gophers.size(); i++)
            GopherTrap.caughtYa(gophers.elementAt(i));
    }
}
```

新的 GopherVector 类有一个类型为 Vector 的 private 成员（从 Vector 继承有些麻烦，理由稍后便知），而且方法也和 Vector 类似。然而，它不会接收和产生普通 Object，只对 Gopher 对象感兴趣。

由于 GopherVector 只接收一个 Gopher（地鼠），所以假如我们使用：

```
gophers.addElement(new Pigeon());
```

就会在编译期间获得一条出错消息。采用这种方式，尽管从编码的角度看显得更令人沉闷，但可以立即判断出是否使用了正确的类型。注意在使用 elementAt()时不必进行造型——它肯定是一个 Gopher。

枚举器

容纳各种各样的对象正是集合的首要任务。在 Vector 中，addElement()便是我们插入对象采用的方法，而 elementAt()是提取对象的唯一方法。 Vector 非常灵活，我们可在任何时候选择任何东西，并可使用不同的索引选择多个元素。

若从更高的角度看这个问题，就会发现它的一个缺陷：需要事先知道集合的准确类型，否则无法使用。乍看来，这一点似乎没什么关系。但假若最开始决定使用 Vector，后来在程

序中又决定（考虑执行效率的原因）改变成一个 List（属于 Java1.2 集合库的一部分），这时又该如何做呢？

我们通常认为反复器是一种“轻量级”对象；也就是说，创建它只需付出极少的代价。但也正是由于这个原因，我们常发现反复器存在一些似乎很奇怪的限制。例如，有些反复器只能朝一个方向移动。

Java 的 Enumeration（枚举，注释②）便是具有这些限制的一个反复器的例子。除下面这些外，不可再用它做其他任何事情：

1) 用一个名为 elements()的方法要求集合为我们提供一个 Enumeration。我们首次调用它的 nextElement() 时，这个 Enumeration 会返回序列中的第一个元素。
2) 用 nextElement() 获得下一个对象。
3) 用 hasMoreElements()检查序列中是否还有更多的对象

```
class Hamster {
    private int hamsterNumber;
    Hamster(int i) {
        hamsterNumber = i;
    }
    public String toString() {
        return "This is Hamster #" + hamsterNumber;
    }
}
class Printer {
    static void printAll(Enumeration e) {
        while (e.hasMoreElements())
            System.out.println(e.nextElement().toString());
    }
}
public class HamsterMaze {
    public static void main(String[] args) {
        Vector v = new Vector();
        for (int i = 0; i < 3; i++)
            v.addElement(new Hamster(i));
        Printer.printAll(v.elements());
    }
}
```

仔细研究一下打印方法：

```
static void printAll(Enumeration e) {
while(e.hasMoreElements())
System.out.println(
e.nextElement().toString());
```

```
}
```

注意其中没有与序列类型有关的信息。我们拥有的全部东西便是 Enumeration。为了解有关序列的情况，一个 Enumeration 便足够了：可取得下一个对象，亦可知道是否已抵达了末尾。取得一系列对象，然后在其中遍历，从而执行一个特定的操作—— 这是一个颇有价值的编程概念。