Java 基础：集合基础（2）

集合的类型

Vector

Java 标准集合里包含了 toString()方法，所以它们能生成自己的 String 表达方式，包括它们容纳的对象。

例如在 Vector 中， toString()会在 Vector 的各个元素中步进和遍历，并为每个元素调用 toString()。假定我们现在想打印出自己类的地址。看起来似乎简单地引用 this 即可（特别是 C++程序员有这样做的倾向）：

```
public class CrashJava {
    public String toString() {
        return "CrashJava address: " + this + "\n";
    }
    public static void main(String[] args) {
        Vector v = new Vector();
        for (int i = 0; i < 10; i++)
            v.addElement(new CrashJava());
        System.out.println(v);
    }
}
```

此时发生的是字串的自动类型转换。当我们使用下述语句时：

```
“CrashJava address: ” + this
```

编译器就在一个字串后面发现了一个“ +”以及好象并非字串的其他东西，所以它会试图将 this 转换成一个字串。转换时调用的是 toString()，后者会产生一个递归调用。若在一个 Vector 内出现这种事情，看起来堆栈就会溢出，同时违例控制机制根本没有机会作出响应。

若确实想在这种情况下打印出对象的地址，解决方案就是调用 Object 的 toString 方法。此时就不必加入 this，只需使用 super.toString()。当然，采取这种做法也有一个前提：我们必须从 Object 直接继承，或者没有一个父类覆盖了 toString 方法。

BitSet

BitSet 实际是由“二进制位”构成的一个 Vector。如果希望高效率地保存大量“开-关”信息，就应使用 BitSet。它只有从尺寸的角度看才有意义；如果希望的高效率的访问，那么它的速度会比使用一些固有类型的数组慢一些。

BitSet 的最小长度是一个长整数（ Long ）的长度： 64 位。这意味着假如我们准备保存比这更小的数据，如 8 位数据，那么 BitSet 就显得浪费了。所以最好创建自己的类，用它容纳自己的标志位。

Stack

Stack 有时也可以称为“后入先出”（ LIFO)集合。换言之，我们在堆栈里最后“压入”的东西将是以后第一个“弹出”的。和其他所有 Java 集合一样，我们压入和弹出的都是“对象”，所以必须对自己弹出的东西进行“造型”。

下面是一个简单的堆栈示例，它能读入数组的每一行，同时将其作为字串压入堆栈。

```
public class Stacks {
    static String[] months = { "January", "February", "March", "April", "May",
            "June", "July", "August", "September", "October", "November",
            "December" };
    public static void main(String[] args) {
        Stack stk = new Stack();
        for (int i = 0; i < months.length; i++)
            stk.push(months[i] + " ");
        System.out.println("stk = " + stk);
        // Treating a stack as a Vector:
        stk.addElement("The last line");
        System.out.println("element 5 = " + stk.elementAt(5));
        System.out.println("popping elements:");
        while (!stk.empty())
            System.out.println(stk.pop());
    }
}
```

months 数组的每一行都通过 push()继承进入堆栈，稍后用 pop()从堆栈的顶部将其取出。要声明的一点是，Vector 操作亦可针对 Stack 对象进行。这可能是由继承的特质决定的——Stack“属于”一种 Vector。因此，能对 Vector 进行的操作亦可针对 Stack 进行，例如 elementAt()方法。

Hashtable

Vector 允许我们用一个数字从一系列对象中作出选择，所以它实际是将数字同对象关联起来了。但假如我们想根据其他标准选择一系列对象呢？堆栈就是这样的一个例子：它的选择标准是“最后压入堆栈的东西”。

这种“从一系列对象中选择”的概念亦可叫作一个“映射”、“字典”或者“关联数组”。从概念上讲，它看起来象一个 Vector，但却不是通过数字来查找对象，而是用另一个对象来查找它们！这通常都属于一个程序中的重要进程。

在 Java 中，这个概念具体反映到抽象类 Dictionary 身上。该类的接口是非常直观的 size()告诉我们其中包含了多少元素； isEmpty()判断是否包含了元素（是则为 true）；put(Object key, Object value)添加一个值（我们希望的东西），并将其同一个键关联起来（想用于搜索它的东西）； get(Object key)获得与某个键对应的值；而 remove(Object Key)用于从列表中删除“键 - 值”对。还可以使用枚举技术： keys()产生对键的一个枚举（ Enumeration）；而 elements()产生对所有值的一个枚举。这便是一个 Dictionary（字典）的全部。

```
public class AssocArray extends Dictionary {
    private Vector keys = new Vector();
    private Vector values = new Vector();
    public int size() {
        return keys.size();
    }
    public boolean isEmpty() {
        return keys.isEmpty();
    }
    public Object put(Object key, Object value) {
        keys.addElement(key);
        values.addElement(value);
        return key;
    }
    public Object get(Object key) {
        int index = keys.indexOf(key);
        // indexOf() Returns -1 if key not found:
        if (index == -1)
            return null;
        return values.elementAt(index);
    }
    public Object remove(Object key) {
```

```
        int index = keys.indexOf(key);
        if (index == -1)
            return null;
        keys.removeElementAt(index);
        Object returnval = values.elementAt(index);
        values.removeElementAt(index);
        return returnval;
    }
    public Enumeration keys() {
        return keys.elements();
    }
    public Enumeration elements() {
        return values.elements();
    }
    // Test it:
    public static void main(String[] args) {
        AssocArray aa = new AssocArray();
        for (char c = 'a'; c <= 'z'; c++)
            aa.put(String.valueOf(c), String.valueOf(c).toUpperCase());
        char[] ca = { 'a', 'e', 'i', 'o', 'u' };
        for (int i = 0; i < ca.length; i++)
            System.out.println("Uppercase: " + aa.get(String.valueOf(ca[i])));
    }
}
```

在对 AssocArray 的定义中，我们注意到的第一个问题是它“扩展”了字典。这意味着 AssocArray 属于 Dictionary 的一种类型，所以可对其发出与 Dictionary 一样的请求。如果想生成自己的 Dictionary，而且就在这里进行，那么要做的全部事情只是填充位于 Dictionary 内的所有方法(而且必须覆盖所有方法，因为它们—— 除构建器外—— 都是抽象的)。

标准 Java 库只包含 Dictionary 的一个变种，名为 Hashtable(散列表，注释③)。Java 的散列表具有与 AssocArray 相同的接口 (因为两者都是从 Dictionary 继承来的)。但有一个方面却反映出了差别：执行效率。若仔细想想必须为一个 get()做的事情，就会发现在一个 Vector 里搜索键的速度要慢得多。但此时用散列表却可以加快不少速度。不必用冗长的线性搜索技术来查找一个键，而是用一个特殊的值，名为“散列码”。散列码可以获取对象中的信息，然后将其转换成那个对象“相对唯一”的整数 (int)。所有对象都有一个散列码，而 hashCode()是根类 Object 的一个方法。Hashtable 获取对象的 hashCode()，然后用它快速查找键。

```
class Counter {
    int i = 1;
    public String toString() {
        return Integer.toString(i);
    }
}
class Statistics {
    public static void main(String[] args) {
        Hashtable ht = new Hashtable();
        for (int i = 0; i < 10000; i++) {
            // Produce a number between 0 and 20:
            Integer r = new Integer((int) (Math.random() * 20));
            if (ht.containsKey(r))
                ((Counter) ht.get(r)).i++;
            else
                ht.put(r, new Counter());
        }
        System.out.println(ht);
    }
}
```

- 创建“关键”类

但在使用散列表的时候，一旦我们创建自己的类作为键使用，就会遇到一个很常见的问题。例如，假设一套天气预报系统将 Groundhog(土拨鼠)对象匹配成 Prediction(预报) 。这看起来非常直观：我们创建两个类，然后将 Groundhog 作为键使用，而将 Prediction 作为值使用。如下所示：

```
class Groundhog {
    int ghNumber;
    Groundhog(int n) {
        ghNumber = n;
    }
}
class Prediction {
    boolean shadow = Math.random() > 0.5;
    public String toString() {
        if (shadow)
            return "Six more weeks of Winter!";
        else
```

```
            return "Early Spring!";
    }
}
public class SpringDetector {
    public static void main(String[] args) {
        Hashtable ht = new Hashtable();
        for (int i = 0; i < 10; i++)
            ht.put(new Groundhog(i), new Prediction());
        System.out.println("ht = " + ht + "\n");
        System.out.println("Looking up prediction for groundhog #3:");
        Groundhog gh = new Groundhog(3);
        if (ht.containsKey(gh))
            System.out.println((Prediction) ht.get(gh));
    }
}
```

问题在于 Groundhog 是从通用的 Object 根类继承的（若当初未指定基础类，则所有类最终都是从 Object 继承的）。事实上是用 Object 的 hashCode()方法生成每个对象的散列码，而且默认情况下只使用它的对象的地址。所以，Groundhog(3)的第一个实例并不会产生与 Groundhog(3)第二个实例相等的散列码，而我们用第二个实例进行检索或许认为此时要做的全部事情就是正确地覆盖 hashCode()。但这样做依然行不能，除非再做另一件事情：覆盖也属于 Object 一部分的 equals()。当散列表试图判断我们的键是否等于表内的某个键时，就会用到这个方法。同样地，默认的 Object.equals()只是简单地比较对象地址，所以一个 Groundhog(3)并不等于另一个 Groundhog(3)。

因此，为了在散列表中将自己的类作为键使用，必须同时覆盖 hashCode()和 equals()，就象下面展示的那样：

```
class Groundhog {
    int ghNumber;
    Groundhog(int n) {
        ghNumber = n;
    }
}
class Prediction {
    boolean shadow = Math.random() > 0.5;
    public String toString() {
        if (shadow)
            return "Six more weeks of Winter!";
        else
```

```
            return "Early Spring!";
    }
}
public class SpringDetector {
    public static void main(String[] args) {
        Hashtable ht = new Hashtable();
        for (int i = 0; i < 10; i++)
            ht.put(new Groundhog(i), new Prediction());
        System.out.println("ht = " + ht + "\n");
        System.out.println("Looking up prediction for groundhog #3:");
        Groundhog gh = new Groundhog(3);
        if (ht.containsKey(gh))
            System.out.println((Prediction) ht.get(gh));
    }
}
```

Groundhog2.hashCode()将土拔鼠号码作为一个标识符返回（在这个例子中，程序员需要保证没有两个土拔鼠用同样的 ID 号码并存）。为了返回一个独一无二的标识符，并不需要 hashCode()，equals()方法必须能够严格判断两个对象是否相等。

equals()方法要进行两种检查：检查对象是否为 null；若不为 null ，则继续检查是否为 Groundhog2 的一个实例（要用到 instanceof 关键字）。即使为了继续执行 equals()，它也应该是一个 Groundhog2。正如大家看到的那样，这种比较建立在实际 ghNumber 的基础上。这一次一旦我们运行程序，就会看到它终于产生了正确的输出（许多 Java 库的类都覆盖了 hashcode() 和 equals()方法，以便与自己提供的内容适应）。

再论枚举器

将穿越一个序列的操作与那个序列的基础结构分隔开。在下面的例子里， PrintData 类用一个 Enumeration 在一个序列中移动，并为每个对象都调用 toString()方法。此时创建了两个不同类型的集合：一个 Vector 和一个 Hashtable。并且在它们里面分别填充 Mouse 和 Hamster 对象，由于 Enumeration 隐藏了基层集合的结构，所以 PrintData 不知道或者不关心 Enumeration 来自于什么类型的集合：

```
class PrintData {
    static void print(Enumeration e) {
        while (e.hasMoreElements())
            System.out.println(e.nextElement().toString());
    }
}
```

```
class Enumerators2 {
    public static void main(String[] args) {
        Vector v = new Vector();
        for (int i = 0; i < 5; i++)
            v.addElement(new Mouse(i));
        Hashtable h = new Hashtable();
        for (int i = 0; i < 5; i++)
            h.put(new Integer(i), new Hamster(i));
        System.out.println("Vector");
        PrintData.print(v.elements());
        System.out.println("Hashtable");
        PrintData.print(h.elements());
    }
}
```

注意 PrintData.print()利用了这些集合中的对象属于 Object 类这一事实，所以它调用了 toString()。但在解决自己的实际问题时，经常都要保证自己的 Enumeration 穿越某种特定类型的集合。例如，可能要求集合中的所有元素都是一个 Shape（几何形状），并含有 draw()方法。若出现这种情况，必须从 Enumeration.nextElement()返回的 Object 进行下溯造型，以便产生一个 Shape。

排序

编写通用的排序代码时，面临的一个问题是必须根据对象的实际类型来执行比较运算，从而实现正确的排序。当然，一个办法是为每种不同的类型都写一个不同的排序方法。然而，应认识到假若这样做，以后增加新类型时便不易实现代码的重复利用。

程序设计一个主要的目标就是“将发生变化的东西同保持不变的东西分隔开”。在这里，保持不变的代码是通用的排序算法，而每次使用时都要变化的是对象的实际比较方法。因此，我们不可将比较代码“硬编码”到多个不同的排序例程内，而是采用“回调”技术。

利用回调，经常发生变化的那部分代码会封装到它自己的类内，而总是保持相同的代码则“回调”发生变化的代码。这样一来，不同的对象就可以表达不同的比较方式，同时向它们传递相同的排序代码。

下面这个“接口”（ Interface ）展示了如何比较两个对象，它将那些“要发生变化的东西”封装在内：

```
interface Compare {
boolean lessThan(Object lhs, Object rhs);
boolean lessThanOrEqual(Object lhs, Object rhs);
}
```

对这两种方法来说， lhs 代表本次比较中的“左手”对象，而 rhs 代表“右手”对象。

可创建 Vector 的一个子类，通过 Compare 实现“快速排序”。对于这种算法，包括它的速度以及原理等等。

```
public class SortVector extends Vector {
    private Compare compare; // To hold the callback
    public SortVector(Compare comp) {
        compare = comp;
    }
    public void sort() {
        quickSort(0, size() - 1);
    }
    private void quickSort(int left, int right) {
        if (right > left) {
            Object o1 = elementAt(right);
            int i = left - 1;
            int j = right;
            while (true) {
                while (compare.lessThan(elementAt(++i), o1))
                    ;
                while (j > 0)
                    if (compare.lessThanOrEqual(elementAt(--j), o1))
                        break; // out of while
                if (i >= j)
                    break;
                swap(i, j);
            }
            swap(i, right);
            quickSort(left, i - 1);
            quickSort(i + 1, right);
        }
    }
    private void swap(int loc1, int loc2) {
        Object tmp = elementAt(loc1);
        setElementAt(elementAt(loc2), loc1);
        setElementAt(tmp, loc2);
    }
}
```

为使用 SortVector，必须创建一个类，令其为我们准备排序的对象实现 Compare。此时内部类并不显得特别重要，但对于代码的组织却是有益的。下面是针对 String 对象的一个例子：

```
public class StringSortTest {
    static class StringCompare implements Compare {
        public boolean lessThan(Object l, Object r) {
            return ((String) l).toLowerCase().compareTo(
                    ((String) r).toLowerCase()) < 0;
        }
        public boolean lessThanOrEqual(Object l, Object r) {
            return ((String) l).toLowerCase().compareTo(
                    ((String) r).toLowerCase()) <= 0;
        }
    }
    public static void main(String[] args) {
        SortVector sv = new SortVector(new StringCompare());
        sv.addElement("d");
        sv.addElement("A");
        sv.addElement("C");
        sv.addElement("c");
        sv.addElement("b");
        sv.addElement("B");
        sv.addElement("D");
        sv.addElement("a");
        sv.sort();
        Enumeration e = sv.elements();
        while (e.hasMoreElements())
            System.out.println(e.nextElement());
    }
}
```

一旦设置好框架，就可以非常方便地重复使用象这样的一个设计——只需简单地写一个类，将“需要发生变化”的东西封装进去，然后将一个对象传给 SortVector 即可继承（ extends ）在这儿用于创建一种新类型的 Vector——也就是说， SortVector 属于一种 Vector，并带有一些附加的功能。继承在这里可发挥很大的作用，但了带来了问题。它使一些方法具有了 final 属性，所以不能覆盖它们。如果想创建一个排好序的 Vector，令其只接收和生成 String 对象，就会遇到麻烦。因为 addElement()和 elementAt()都具有 final 属性，而且它们都是我们必须覆盖的方法，否则便无法实现只能接收和产生 String 对象。

但在另一方面，请考虑采用“合成”方法：将一个对象置入一个新类的内部。此时，不是改写上述代码来达到这个目的，而是在新类里简单地使用一个 SortVector。在这种情况下，用于实现 Compare 接口的内部类就可以“匿名”地创建。

```
import java.util.*;
public class StrSortVector {
    private SortVector v = new SortVector(
    // Anonymous inner class:
            new Compare() {
                public boolean lessThan(Object l, Object r) {
                    return ((String) l).toLowerCase().compareTo(
                            ((String) r).toLowerCase()) < 0;
                }
                public boolean lessThanOrEqual(Object l, Object r) {
                    return ((String) l).toLowerCase().compareTo(
                            ((String) r).toLowerCase()) <= 0;
                }
            });
    private boolean sorted = false;
    public void addElement(String s) {
        v.addElement(s);
        sorted = false;
    }
    public String elementAt(int index) {
if(!sorted) {
v.sort();232
sorted = true;
}
return (String)v.elementAt(index);
}
    public Enumeration elements() {
        if (!sorted) {
            v.sort();
            sorted = true;
        }
        return v.elements();
    }
    // Test it:
```

```
    public static void main(String[] args) {
        StrSortVector sv = new StrSortVector();
        sv.addElement("d");
        sv.addElement("A");
        sv.addElement("C");
        sv.addElement("c");
        sv.addElement("b");
        sv.addElement("B");
        sv.addElement("D");
        sv.addElement("a");
        Enumeration e = sv.elements();
        while (e.hasMoreElements())
            System.out.println(e.nextElement());
    }
}
```