第 5 讲：抽象类型

5.1. 简介

这一课里，我们来看一种特殊的依赖，这是一种关于类型表示的抽象类型客户，我们要看看如何避免依赖。我们也要简要的讨论指定抽象类型的规格说明属性的概念，操作分类和表示均衡。

5.2. 用户定义类型

在早期的计算中，一个程序设计语言提供内置的类型（比如整型，布尔型，字符串型等）和内置程序，比如输入和输出。用户可以定义自己的程序：大型程序就是这样建立的。

抽象类型的思想是软件开发的一个巨大进步：程序语言的设计也允许用户自定义类型。这个思想源自很多研究人员的工作，特别的 Dahl（Simula 语言的发明者），Hoare（开发了许多技术我们至今还用来推论抽象类型），Parnas（提出了术语‘信息隐藏’并且第一次用他们的压缩秘诀提出了组织程序模块的概念），还有 MIT 的 Barbara Liskov 和 John Guttag，他们在抽象类型标准中做了根基性的工作，并在程序语言中支持他们（还开发了 6170）。

数据抽取的主要思想是你可以通过能够执行的操作来表现一个类型的特征。一个数字就是你可以进行加和乘操作的类型；一个字符串就是可以连接和提取子串的类型；一个布尔型就是可以取反的类型，等等。在某种意义上说，用户在早期的程序语言中就已经定义了他们自己的类型：你可以创建一个日期记录类型：比如用整型域表示年、月、日。但是以操作为焦点使抽象类型更新和不同：类型的使用者不用考虑它的值使如何存储的，通过同样的方法程序员也可以忽略编译器如何存储整数的。只有操作是最重要的。

就像在许多现代程序设计语言一样，在 Java 中内置类型和用户自定义类型的分界是有些模糊的。比如，java.lang 类中的 Integer 和 Boolean 类型是内置的；你是否把所有 java.util 的集合当作内置是不清楚的（也是不重要的）。Java 使用非对象的原始类型使问题复杂化了。这些类型的设置，比如整型和布尔型，不能被用户扩展。

5.3. 类型和操作分类

不管是内置类型还是用户自定义的类型，都可以用不变和可变分类。可变类型的对象能够更改：就是说，他们提供的操作被执行时产生一种结果而其他作用在相同对象上的操作却产生不同的结果。所以向量类型是可变的，因为你可以调用 *addElement* 操作并用 *size* 操作来观测改变。但字符串类型就是不变的，因为它的操作是创建新串而不是改变存在的串。有时一个类型被规定为可变和不变两种形式。比如 *StringBuffer* 是 *String* 的一种可变形式（尽管这两个在 Java 类型中并不完全相同，也不可以相互转换）。

通常的不可变类型的更容易分辨。混淆不是问题，因为共享不会被观察到。有时候使用不变类型更有效，因为更多的共享是可能的。但是使用可变类型使许多问题更自然的表达，而当大型结构需要本地改变时，他们变得更高效。

抽象类型的操作是按下面分类的：

- 创建这种类型新对象的*构造者*(constructors)。一个构造器可能把一个对象当作一个参数而不是一个该类型的对象。
- 根据旧对象创建新对象的*生产者*(producers)，条件是相同的。比如字符串连接方法，就是一个生产者：它根据两个字符串生产一个新串表示它们的连接。
- 改变对象的*改变者*(mutators)。比如向量类的 *addElement* 方法通过在向量的高端添加元素来改变一个向量。
- 提取抽象类型的对象并返回一个不同类型的对象的*观察者*(observers)。比如*向量*类的 *size* 方法返回整型。

我们可以总结这些不同点如下：

constructor: t -> T
producer: T, t -> T
mutator: T, t -> void
observer: T, t -> t

这些非正式的表示了在各种类里操作符号的的形态。每个T表示抽象类型自己，每个t表示其他类型。总的来说，当一个类型出现在左边，他可以出现不止一次。比如一个生产者可以取两个抽象类型值；字符串连接取了两个字符串。左边出现的t也可以忽略；一些观察者不采取非抽象结构（比如size），也有的采用几个。

这个分类给出了一些有用的术语，但它不准确。在复杂数据类型中，比如可能存在同时是生产者和改变者的操作。有些人用‘生产者’表示没有改变发生。

你们应该知道的另一个短语是*迭代*。一个迭代经常表示一类特殊的方法（Java中不允许），每次返回一系列对象收集中的一个——比如一系列元素中的一个。在Java中，一个迭代是这样一个*类*，它提供了用来获得一系列对象收集的方法。许多收集类提供了一个名为*iterator*的方法并返回一个迭代。

5.4. 举例：List

我们来看一个关于抽象类型的例子：列表（*list*）。在Java中，一个列表就像一个数组。它提供了在某个特定位置提取和替换元素的方法。但和数组不同的是，它也有在特定位置插入和删除元素的方法。在Java中，*List*是一个具有很多方法的借口，但是现在，我们假设它是一个具有下面方法的简单类：

```
public class List {
    public List ();
    public void add (int i, Object e);
    public void set (int i, Object e);
    public void remove (int i);
    public int size ();
    public Object get (int i);
    }
```

其中add、set和remove方法是改变者；size和get方法是观察者。一个可变类型没有生

产者是很普通的（当然一个不变类也可以没有改变者）。

为了说明这些方法，我们要从几个方面来讨论一下列表应该是什么样子的。我们通过*规格说明属性*的概念来处理这个问题。你可以想象类里的一个对象有这些属性，但记住它们在实际实现中不需要作为属性，而且没有要求一个规格说明属性的值是一些方法可以得到的。在这种情况下我们用一个简单的规格说明属性描述列表，

```
seq [Object]        elems;
```

对一个列表*I*，表达式*I.elems*表示存储在列表中的对象序列，从零开始标识。现在我们可以详细说明一些方法：

```
public void get (int i);
// throws
//   IndexOutOfBoundsException if i < 0 or i > length (this.elems)
// returns
//    this.elems [i]
public void add (int i, Object e);
// modifies this
// effects
//   throws IndexOutOfBoundsException if i < 0 or i > length (this.elems)
//   else this.elems' = this.elems [0..i-1] ^ <e> ^ this.elems [i..]
public void set (int i, Object e);
// modifies this
// effects
//    throws IndexOutOfBoundsException if i < 0 or i >= length (this.elems)
//    else this.elems' [i] = e and this.elems unchanged elsewhere
```

在*add*的后置条件中，我用了*s[i..j]*来表示s的下标从 *i* 到 *j* 的子序列，而*s[i..]*表示从 *i* 开始的后缀。符号^表示序列串联。所以后置条件说明，当下标在边界或一个在上面，一个新元素就在指定下标处接入。

5.5. 设计一个抽象类型

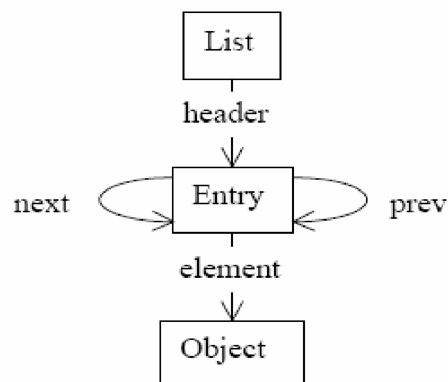
设计一个抽象类型包括选择合适的操作和决定它们的行为。一些要点规则：

- 设计一些可以通过有力的方式联合的简单操作好于许多复杂操作。
- 每一个操作应该有良好定义的目标，而且应该有连贯的行为而不是繁琐的特殊条件。
- 提供的操作应该是足够的；必须满足的客户要做的各种计算。一个好的测试就是检查该类型的对象的每个属性都可以提取。比如，如果没有*get*操作，我们就不能够找到列表中的元素。基本的信息不应该很难获得。*size*方法不是严格必需的，因为我们可以在增加变量时我们可以使用*get*，但是这样是低效且不方便的。
- 类型可以是一般的：比如一个列表或序列，或者一个图。也可以是指定范围的：一个街道图，一个雇员数据库，一个电话本等等。但不能混和一般个特殊范围的性质。

5.6. 表示选择

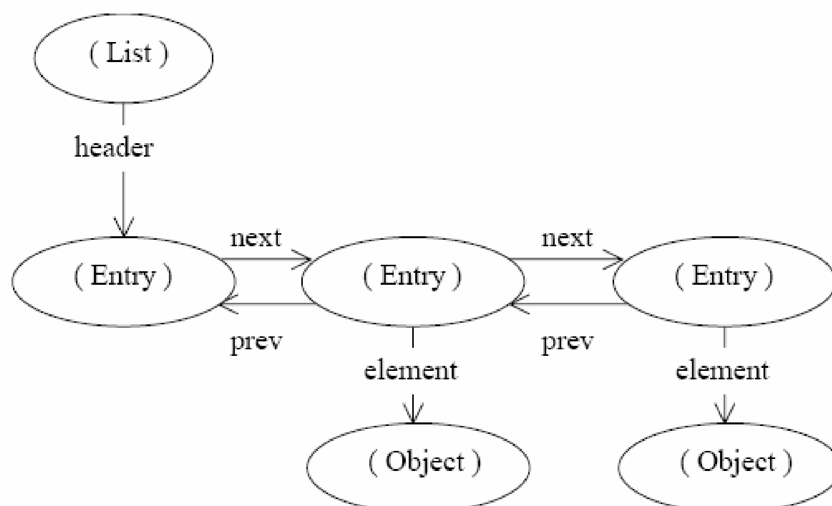
迄今为止，我们都通过操作把重点放在抽象类型的性质上。在代码中，一个实现抽象类型的类提供了一个*表示（representation）*：实际支持操作的数据结构。表示是一些属性的集合，每个属性中都包含了其他的Java类型；在一个递归实现中，一个属性可以有抽象类型但在Java中很少这样做。

比如链表是一个很常用的列表表示。下面的对象模型表示了一个链表的实现，它类似于（但不完全相同）标准Java库中的*LinkedList*类：

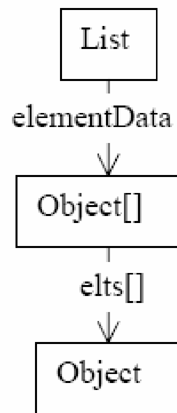


这个列表对象有一个*header*属性和*Entry*对象相连接。*Entry*对象是一个有三个属性的记录：*next*和*prev*可以控制和其它*Entry*对象的连接，*element*控制和元素对象的连接。*Next*和*prev*属性是指向列表中前后的链接。在列表的中间，紧跟*next*然后*prev*就会把你带回到你开始时的对象。让我们假设链接列表没有把空指向作为元素存储。在列表的开始经常存在一个*Entry*哑元，它的元素属性为空，但这并不作为元素解释。

下面的图表对象表示一个包含两个元素的列表：

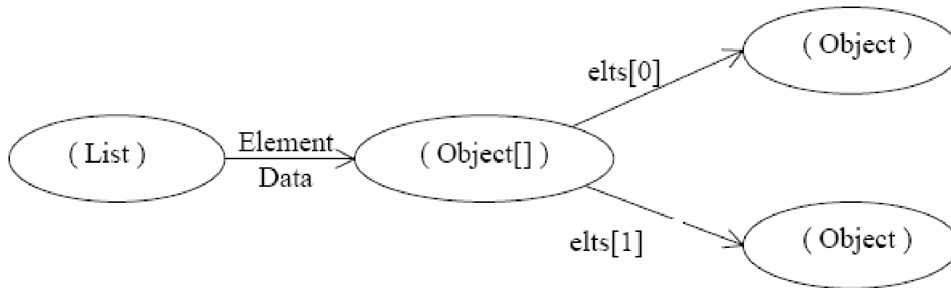


另外，可以用数组表示列表。下面的对象模型



表示如何用标准的Java库中的ArrayList类表示列表。

这里是用它表示的含有两个元素的列表：



这些表示有这不同的优点。链表表示在有较多表前插入操作时会更高效，因为它可以接入一个元素而且只改动两个指针。数组表示需要将所有插入点之上的元素上移，而且如果数组太小，可能还要分配一个新的更大的数组并拷贝所有相应的元素。如果有许多的*get*和*set*操作，数组列表表示就会更好一些，因为在固定的时间内它提供随机入口，而链表需要进行顺序查找。

当我们用列表写代码时我们可能不知道哪些操作占多数。那么，重要的问题就是如何保证在以后能够很容易的改变表示。

5.7. 表示独立性

表示的独立性是要保证抽象结构独立于它的表示，所有改变表示对抽象结构之外的代码没有影响。我们来检测一下如果没有独立性哪里会出错，然后看一看帮助保证独立性的一些语言机制。

假定我们知道我们的列表是由一组数组元素来完成的。我们打算使用一些代码来创建一系列对象，但很不幸的是，它创建了一个向量而不是列表。我们的列表类型没有提供可以转

换的构造器。我们发现向量有一个copyInto方法可以把向量的元素拷贝到一个数组。这里是我们现在写的：

```
List I = new List ();
v.copyInto (I.elementData);
```

多聪明的一个窍门（hack）！像许多窍门一样它只工作一小会儿。加入列表类的实现者现在改变了表示方法，把数组型改成了链表类型。现在列表*l*将不再有elementData属性，而编译器会拒绝这个程序。这是一个失败的表示独立性例子：我们需要更改代码中所有这样做了的地方。

遇到一个编译错误还不算太大的问题，更糟糕的是如果执行成功所做的改变仍然会破坏程序。下面列举了这个问题的一种可能情况：

总的来说，数组的大小应该比列表中元素的数目大，否则每次添加或移除元素都要创建新数组。所以一定有某种方式标志包含数组元素的段的结束。假设列表的实现者按照惯例设计了它，用段执行到的第一个空引用，或者到数组的最后，无论哪个在先。值得庆幸的是（实际是不幸的），我们的hack在这样的环境下能够起作用。

现在我们的执行者发现者是一个错误的决定，因为检测列表的大小需要一个线性查找一直到找到第一个空引用。所以他增加了一个*size*属性并且每当进行了改变列表的操作时就更新它。这样更好一些，因为查找大小现在采取的是固定时间。它也很自然的把空引用当作列表元素处理（这就是Java *LinkedList* 的实现也这样做的原因）。

现在我们聪明的hack好像生产了一些令人讨厌的行为，而它们产生的原因很难被捕捉到。我们创建的列表有个糟糕的size属性：无论列表中有多少元素它始终保持为零（除非我们单独更新数组）。*Get*和*set*操作将起作用，但第一个对size的调用会神秘的失败。

下面是另外一个例子。假设我们有一个链表实现，而且包括一个操作它返回指定位置的*Entry*对象。

```
public Entry getEntry (int i)
```

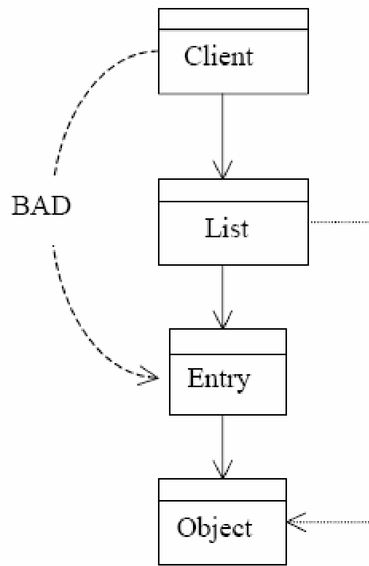
我们的基本原理是如果存在对一个位置的元素多次调用*set*，将会按照线性查找到的元素重复的保存。现在我们把

```
I.set (i, x); ... ; I.set (i, y)
```

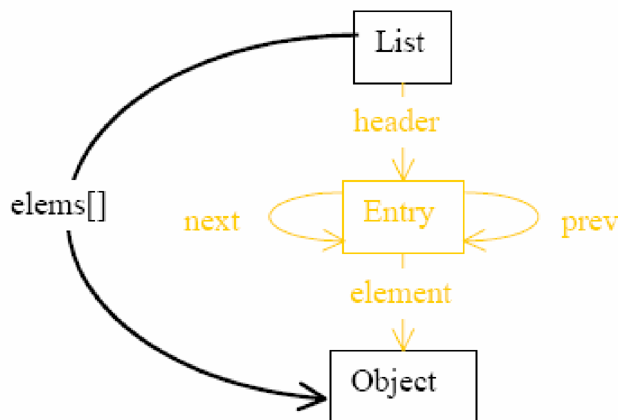
写成：

```
Entry e = I.getEntry (i);
e.element = x;
...
e.element = y;
```

这仍然违反表示的独立性，因为当我们转换成数组表示时，就不再有*Entry*对象了。我们可以用一个模块依赖图表说明这个问题。



在*List*类中应该只有一个对客户类型*Client*的依赖（同样在元素类型里，这里是*Object*）。*Client*对*Entry*的依赖是我们问题的起因。回到我们的独立性对象模型，我们想要访问Entry类和它作为List内在的联合。我们可以通过把客户无法访问的部分用红色表示来非正式的解释一些这个问题（如果你是用的是黑白打印，我所指的是*Entry*和它所有的输入输出弧线），并添加标准属性*elems*隐藏表示：



在*Entry*示例中，我们已经揭示了表示。一个更模糊的说明，也是最普遍的，是从完成一个返回值为集合的方法中得到的。当表示已经包含了一系列合适类型的对象，直接返回它是很诱人的。比如，假如*List*有个*toArray*方法相对列表的元素返回一组元素。如果我们用数组实现列表，我们就可以直接返回数组本身。如果*size*属性是基于下标的（一个空引用先出现）一个对数组的修改可能破坏数组大小的计算。

a =I.toArray (); // exposes the rep
a[i] = null; //ouch!!

…

l.get (i); // now behaves unpredictably

一旦大小计算错误，所有地狱般的破坏就开始释放：接下来的操作可能会按任意的方式运行。

5.8. 语言机制

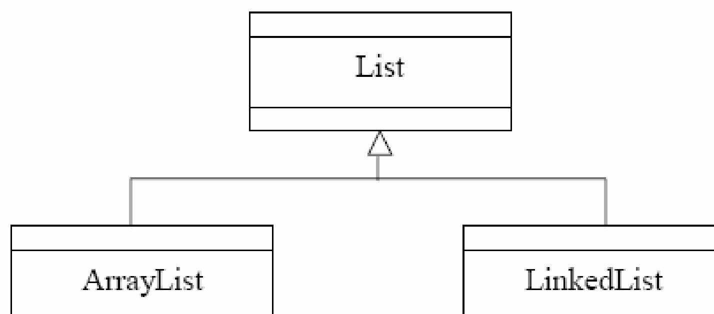
为了防止对表示的访问，我们可以设置私有属性。这就排除了数组hack，代码段

v.copyInto (l.elementData);

将会被编译器拒绝，因为表达式*l.elementData*将从一个私有属性所在类外引用它。

*Entry*的问题不是这么容易解决的。它没有对表示的直接访问。而是List类返回了一个属于表示的Entry对象。这叫作表示*暴露*，并且它不能只靠语言机制来预防。我们需要检查表示中对可变部分的引用没有传出到客户，而且表示的创建没有用到引进的可变对象。比如在数组表示中，我们不需要创建者用一个数组并把它赋值到内在属性。

接口提供了实现表示独立性的又一个方法。在Java标准库中，我们讨论的列表的两种表示实际上是独特的类，*ArrayList*和*LinkedList*。两者都被声明为*List*接口的扩展。在这种情况中，接口打破了客户和其它类之间的依赖关系。表示类：



这种途径是较好的因为接口不能有（非静态）属性，所有访问表示的问题永远不会出现。但是因为Java接口不能有创建者，它实际用起来是笨拙的，因而在实现类之间共享的构造者符号信息就不可以在接口中表达。此外，既然在某些点客户代码必须构造对象，具体的类之间肯定有依赖（我们将很明显的局部化）。我们将在后面课程里讨论的Factory式样，从事这个特殊问题的研究。

5.9. 总结

抽象类型的性质是由它们的操作表现的。表示独立性使改变一个类型的表示而保持客户不变成为可能。在Java中，访问控制机制和接口可以帮助保证独立性。但表示暴露是一个需要小心操作的地方，它仍需要通过仔细的程序员纪律来处理。