Chapter 11 Generalizing Our Array and Linked-List Containers

We have now implemented two kinds of array container, `Vector` and `Heap`, and two kinds of linear linked container, `Stack` and `Queue`. In this chapter we will generalize our array containers and our linear linked containers with abstract classes `ArrayContainer` and `LinearLinkedContainer`, respectively. We will also add an abstract class called `LinkedContainer` between classes `Container` and `LinearLinkedContainer`, allowing for the future expansion of our software design to include nonlinear linked containers such as trees.

11.1 Our Current Model

Our current software model is shown below in Figure 11.1. The solid arrows with crosshair ends denote the *nesting* relationship. A UML nesting relationship corresponds to a Java *nested type*. We will implement nested types `Node` and `SLNode`. Implementing `DLNode` is left as an exercise.

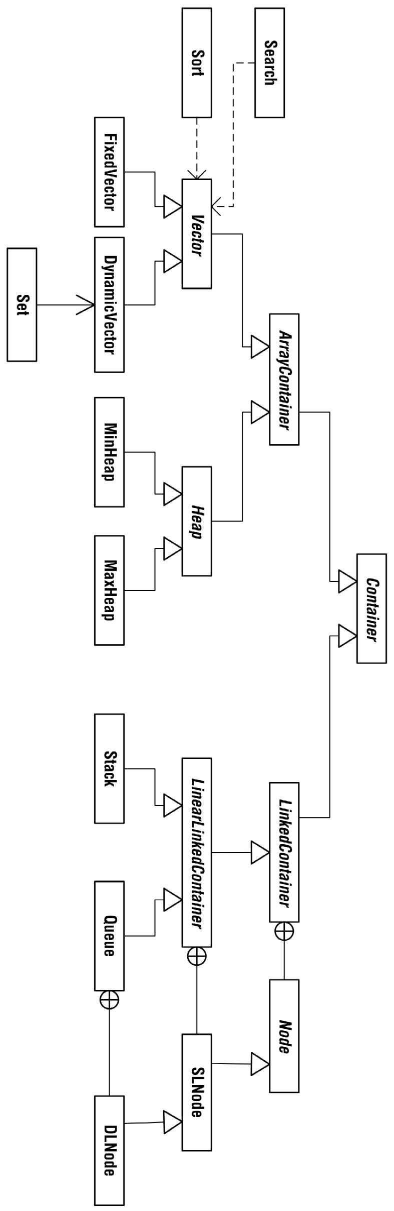

Fig. 11.1. Our current UML class diagram, which generalizes array and linked-list containers and makes room for the addition of nonlinear linked containers

11.2 Generalizing Our Array Containers

We have already generalized our containers somewhat by moving field `numItems` and methods `clear`, `isEmpty`, and `size` into class `Container`. But classes `Vector` and `Heap` also have an array, `data`, in common. We can move the array into an abstract class, thereby relating `Vector` and `Heap` and providing a general class that may be specialized by additional kinds of array containers.

11.2.1 Class `ArrayContainer`

The code for abstract class `ArrayContainer` appears below.

```
public abstract class ArrayContainer extends Container
{
    protected static final int DEFAULT_CAPACITY = 1000;
    protected Object[] data;

    public ArrayContainer()
    {
        data = new Object[DEFAULT_CAPACITY];
    }

    public int capacity()
    {
        return data.length;
    }

    public void clear()
    {
        for (int j = 0; j < numItems; j++)
            data[j] = null;
        super.clear();
    }

    protected boolean isFull()
    {
        return numItems == capacity();
    }
}
```

11.2.2 Updating Classes `Vector` and `Heap`

Changing class `Vector` so that it inherits from `ArrayContainer` is straightforward and is left as an exercise. Updating our `Heap` implementation presents more of a challenge because class `Heap` needs an array of `Comparable` elements. We can give `Heap` what it needs by *shadowing* the `data` field of class `ArrayContainer`.

Field Shadowing

To shadow a superclass field is to declare a field of the same name in a subclass. Methods of the subclass use that name to access the new field, not the shadowed superclass field. The shadowed field is still present, but it must be accessed by prepending `super` to its name.

The Revised `Heap` *Implementation*

Our new `Heap` class is shown below.

```
public abstract class Heap extends ArrayContainer
{
    protected Comparable[] data;  // This array shadows the
                                  // superclass field.
    public Heap()
    {
        // The 'data' array of ArrayContainer is being shadowed
        // precisely because it is not needed here. Thus it should
        // be reclaimed by the garbage collector.

        super.data = null;
        data = new Comparable[DEFAULT_CAPACITY];
    }

    public Heap(int initCapacity)
    {
        super.data = null;
        if (initCapacity <= 0)
            data = new Comparable[DEFAULT_CAPACITY];
        else
            data = new Comparable[initCapacity];
    }

    // The capacity() method of class ArrayContainer must be
    // overridden here because the superclass method deals with
    // the shadowed array, not the array of Comparable.

    public int capacity()
    {
        return data.length;
    }

    // Method clear() must be overridden for the same reason.

    public void clear()
    {
        for (int j = 0; j < numItems; j++)
            data[j] = null;
        numItems = 0;
    }

    public void contract()
    {
        if (size() == data.length)
            return;
        Comparable[] newData = new Comparable[size()];
        for (int j = 0; j < size(); j++)
            newData[j] = data[j];
        data = newData;
    }
```

```
protected abstract void percolate();

public void insert(Comparable element)
{
    if (isFull())
    {
        Comparable[] newData =
            new Comparable[data.length * 2];
        for (int j = 0; j < numItems; j++)
            newData[j] = data[j];
        data = newData;
    }
    data[numItems++] = element;
    percolate();
}

protected boolean isLeaf(int pos)
{
    return 2 * pos + 1 >= size();
}

protected int leftChild(int pos)
{
    if (pos < 0)
        return -1;
    return 2 * pos + 1;
}

protected int rightChild(int pos)
{
    if (pos < 0)
        return -1;
    return 2 * pos + 2;
}

protected int parent(int pos)
{
    if (pos < 1)
        return -1;
    return (pos - 1) / 2;
}

protected Comparable peek()
{
    if (isEmpty())
        return null;
    return data[0];
}

protected abstract void sift();

protected Comparable remove()
{
    if (isEmpty())
        return null;
    swap(data, 0, size() - 1);  // Swap the root item and the
                                // last item,...
    Comparable root = data[size() - 1];  // then save the root
                                         // item.
    data[--numItems] = null; // Delete the root from the
                             // array.
    if (size() != 0)         // If the heap is non-empty, sift
        sift();              // to restore the heap ordering.
    return root;
}
```

```
    protected void swap(Comparable[] arr, int first, int second)
    {
        Comparable temp = arr[first];
        arr[first] = arr[second];
        arr[second] = temp;
    }
}
```

11.3 Generalizing Linked Containers and Linear Linked Containers

In Chapter 6 we implemented the heap ADT as an array container, despite the fact that a complete binary tree, or any tree, is conceptually a linked structure. Tree ADTs besides the heap are typically implemented as linked containers, and so it makes sense to make room in our software model for all linked containers, as opposed to merely accommodating the linear linked containers presented in this book. Thus we add class `LinkedContainer` to our model. The code for `LinkedContainer` appears below.

```
public abstract class LinkedContainer extends Container
{
    /*
    Any linked container is composed of nodes, but the particular
    form of the nodes depends on the type of container. Classes
    SLNode and DLNode are suitable for linear linked containers,
    for example, but not for, say, linked binary-tree containers
    because a binary-tree node typically has links to its parent,
    its left child, and its right child. Thus class LinkedCon-
    tainer provides a member class called Node, an instance of
    which has only a data portion. The links appropriate for a
    specific kind of linked container can be obtained by subclass-
    ing Node, as we will soon see.
    */

    protected static abstract class Node
    {
        public Object data;

        public Node(Object dat)
        {
            data = dat;
        }
    }
}
```

11.3.1 Nested Types

A Java nested type (class, enumerated type, or interface) is defined within some containing type, as opposed to being defined in a separate file. In this chapter, we will restrict our use of nested types to *inner classes*, of which there are several kinds: static and nonstatic *member classes*, *local classes*, and *anonymous classes*. All of our inner classes will take the form of static member classes, like class `Node`. Like fields and methods, the visibility of a member class is determined by its being declared private, protected, or public.

Member classes are typically helpers for their enclosing classes. A member class is usually implemented as such because it is intimately associated with its enclosing class and would be of no use outside the enclosing class.

11.3.2 Generalizing Our Linear Linked Containers by Specializing Class `LinkedContainer`

We are now in a position to extract the common elements of our linear linked containers into an abstract class that specializes `LinkedContainer`. Any linear linked container requires, at a minimum, a head reference and singly linked nodes. These requirements are met by class `LinearLinkedContainer`; class `SLNode` has been revised as a static member class that subclasses `LinkedContainer.Node`.

```
public abstract class LinearLinkedContainer
    extends LinkedContainer
{
    protected static class SLNode extends Node
    {
        public SLNode next;

        public SLNode(Object dat)
        {
            super(dat);
        }

        public SLNode(Object dat, SLNode nxt)
        {
            super(dat);
            next = nxt;
        }
    }

    protected SLNode head;

    public void clear()
    {
        head = null;
        super.clear();
    }
}
```

Exercises

1. Revise class `Vector` as a subclass of `ArrayContainer`.
2. Revise classes `Stack` and `Queue` as subclasses of `LinearLinked-Container`. Recall that class `Queue` needs a tail reference and doubly linked nodes; subclass `LinearLinkedContainer.SLNode` with static member class `Queue.DLNode`.
3. Add attributes and operations to the model of Figure 11.1.