User Guide

version 1.0[2]

The Container & Algorithm Library
User Guide

version 1.0$^{\beta 2}$

X3M

http://www.x3m.com/

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

With the approval of STL as a part of the Standard C++ specification, the concept of generic programming was introduced to the masses of C++ programmers. The essence of generic programming, is to separate the algorithms from the data structures, without sacrificing efficiency. This is possible because many algorithms do not depend on some particular implementation of a data structure, but on a couple of semantic properties of the data structures. This was the basis for Alexander Stepanov, the author of STL [SL95].

CAL is an attempt to implement the most important features of STL in Java. As a consequence, a fair amount of the code is derived directly from the code in STL. This is especially evident if one takes a look at the code in the algorithm package. However, the transition from STL is not quite that easy. C++ have some features that the Java language is missing. Among them are templates and pointers[1], both of which STL rely heavily on. The lack of pointers was not too big a concern, there is not the same need for pointers in Java, as references are used exclusively instead. However, templates are one of the main reasons for STL's outstanding performance. In addition, they allow the algorithms and data structures to work with any types, whether they are primitive types like int or float, or user defined objects.

Since Java does not currently provide any equivalent feature, it was necessary make the Object the basic element that every algorithm and data structure in CAL use. In addition, all the container classes are derived from a common base class, Container, and there is a complete hierarchy of Containers, including arrays, sets, maps and search trees.

1.1 Who Should Use CAL

CAL is intended for the serious Java developer needing high performance algorithms and data structures that work, as well as the hobby programmer looking for a more functionality than what is provided in the JDK. CAL is also an ex-

[1] Or rather pointer arithmetic

cellent source for educational purposes. Hopefully both experienced STL users and non-STL users will find CAL a valuable addition to their set of Java tools.

Note that because of the limited bandwidth available on the internet today, it is not recommended to use this library for applets. Its main target was for stand alone applications written in Java. However, in the future, when the bandwidth increases, using it in applets should be feasible. If there is a great need to use the library in an applet, one should strip away all the classes that are neither used nor needed. Depending on how much is used by the applet, one should be able to put together a package that isn't too large.

1.2 How To Use This Document

This document is an attempt to describe the functionality of CAL with the help of a large number of examples together with text explaining the different parts of the library.

Chapter 2 explains how to get started, setting up environment variables and running the examples. Chapter 3 gives a quick overview of the library and its functionality, while the rest of the document describe the various parts of the library in greater detail.

Certain conventions are used throughout the document. First off, text appearing in **boldface** signify new concepts and warnings. *Italics* are used to emphasize certain parts in the text. Finally, `monospaced` text is used for all source code included.

In addition to these conventions, you will find comments in the margin in several places. These will each have an icon attached giving a hint as to what kind of comment it is. A ✍ is just a note. Among other things it is used to tell that some feature require JDK 1.1. Also, it is used whenever there is a good hint or tip for the user.

For those readers not familiar with the $\boldsymbol{O}$-notation or time-complexity analysis, or for those a bit rusty, it might be useful to have a look at Appendix B.

1.3 CAL Compared To Other Libraries

At the time of writing, there are two other libraries for Java based on STL. They are SJL, written by Finn Bock, and JGL, written by ObjectSpace. As a consequence of the fact that all are based on STL, you will find a lot of the same features. Naturally we are inclined to believe that CAL is the better, but we suggest you make up your own mind.

There are other libraries out there on the net, but they will not be discussed here.

1.3.1 Structure

`Yet to be written.`

1.3.2 Extensibility

`Yet to be written.`

1.3.3 Performance

A very important criteria for choosing what library to use, is its performance. As a consequence, we have included an appendix (See Appendix A) that show the result of a series of benchmarks that were designed to compare the libraries as objectively as possible. Please have a look at the source code for the benchmarking as well as run it for yourself.

CHAPTER 2

Getting Started

This chapter describes how to get started with CAL.

2.1 Requirements

CAL is based on the language Java [Fla96]. As it uses some of the latest features that are only available in Java™ JDK 1.1, you will need a development platform supporting this. Suns distribution can be found at `http://www.javasoft.com`. Sadly, the 1.1 version is so new that it has not yet been ported to all platforms. Therefore we have provided a version that will compile under Java™ JDK 1.0.2.

2.2 Installation

First choose a place to install the software on your local computer. Then uncompress the file downloaded into this directory. See the `INSTALL.*` file on our site for instructions on how to do this.

The next step is to include `cal.zip` in the `CLASSPATH` so that `java` and `javac` will find it. Consult the `INSTALL.*` file, or the documentation for your development tool, to find out how to set this variable.

2.3 Package Layout

The layout of the distribution is depicted in the tree shown in figure 2.1.

2.4 Compiling the Library

If there should be a need to recompile the library, please use the `Makefile` provided if you have make installed. If not, enter the directory where you have installed the sources and type (on one line):

```
CAL_10
  ├─ README                General information and latest release notes
  ├─ CHANGES               Changes from previous versions
  ├─ COPYRIGHT             Copyright notice
  ├─ LICENSE               License agreement and warranty
  ├─ INSTALL.unix          Installation guide for UNIX
  ├─ INSTALL.win           Installation guide for WINDOWS-95/NT
  ├─ benchmark.zip         Benchmark binaries (DO NOT UNZIP)
  ├─ cal.zip               Library binaries (DO NOT UNZIP)
  ├─ examples.zip          Example binaries (DO NOT UNZIP)
  ├─ Makefile              GNU Makefile for compiling the library
  ├─ benchmark             Benchmark source files
  │   ├─ info
  │   ├─ modules
  │   └─ suites
  ├─ doc                   Library documentation
  │   └─ api               Generated API doucmentation
  ├─ examples              Example source files
  └─ src                   Library source files
      └─ COM
          └─ X3M
              └─ cal
                  ├─ adaptor
                  ├─ algorithm
                  ├─ function
                  ├─ iterator
                  ├─ matcher
                  └─ misc
```

Figure 2.1: Package layout

```
javac -O -depend -d output-directory *.java
        function/*.java iterator/*.java
        adaptor/*.java algorithm/*.java
        misc/*.java matcher/*.java
```

Likewise for the examples, in the `examples` directory type:

```
javac -O -depend -d output-directory *.java
```

2.5 Running Examples

Each and every example included in the distribution belongs to the package `COM.X3M.cal.examples`. Thus, to run the example `Container1`, type the following at the command prompt:

```
java COM.X3M.cal.examples.Container1
```

This requires that you have `examples.zip` in the `CLASSPATH`.

2.6 Submitting Bug Reports

Before submitting a bug, you should check the latest bug-list, found at the URL `http://www.x3m.com/products/cal/bugs.html` This page will contain a list of any known bugs, as well as info on patches available.

Should you however find a new bug, we would greatly appreciate it if you would take the time to fill out a bug report with as many details as possible, so that we can fix it as quickly as possible. To find information on how to do this, please go to `http://www.x3m.com/products/cal/bugreport.html`.

2.7 Finding Answers to Your Questions

To answer the most common questions regarding CAL, we will put together a constantly updated FAQ[1]. The latest FAQ can be found at the following URL: `http://www.x3m.com/products/cal/faq.html`. If your question is answered here, this will probably be the quickest route to finding your answer.

We welcome any questions about installing, using and licensing CAL.Just email your questions to `cal@x3m.com`. Please understand that since CAL support is free, requests will be handled as time permits.

2.8 Downloading Updates

To find the latest update, have a look at `http://www.x3m.com/products/cal`. All updates will be put there. In addition, you may register yourself to be notified whenever there is an update. Please check the URL for more information on registering yourself on our mailing list.

2.9 Changes from the First Beta

Many changes have been made in this second beta version. The major changes are the inclusion of `IndexedTree`, `GenericTree` and the pattern matchers. For a complete list, please read the `CHANGES` file in the distribution.

2.10 Plans for the Final Release

As this is the second public beta release, we expect there to be changes before the final release is out. We are very interested in getting some feedback on the current implementation, and we are open for suggestions on new features to be included in the next release.

Current plans for the next beta release are to include BTree, a fibonacci heap and one or more new hash tables. These are not definite, but they are fairly certain to be included.

[1] Frequently Asked Questions

CHAPTER 3

Overview of CAL Components

This chapter briefly describes the main components of CAL, what they are, and how they work together.

3.1 Containers

The containers are the objects that store your elements and ensure that they can be retrieved later. There are no restrictions on the elements that may be stored in a container, except that it must be derived from `Object`. In other words, the only kind of elements that may not be inserted in a container are the Java primitive types `boolean`, `byte`, `char`, `double`, `float`, `int`, `long` and `short`.

✍ Note that null values may be stored in containers, but certain algorithms do not work with null values and generate a `NullPointerException` when they encounter one.

The containers in CAL all share the same basic interface, namely `Container`. Furthermore they are subdivided into different classes depending on their characteristics. There's the sequences, the base for all linear containers. Sequences are further refined in the indexed sequences, which allow random access to any element. Then you have the associative containers providing fast access to elements based on keys. Finally the `GenericTree` makes it possible to build a tree structure with an arbitrary number of children in each node.

As an example of what a container is, have a look at the following example showing how one may insert and remove elements in an `Array`.

Example: `Overview1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;

public class Overview1
{
    public static void main(String[] args)
    {
        Array array = new Array();
```

```
        array.add("Earth");
        array.add("Jupiter");
        array.add("Neptune");
        array.add("Jupiter");
        array.add("Uranus");
        array.add("Pluto");

        System.out.println("array = " + array.stringList());

        array.remove("Jupiter");

        System.out.println("array = " + array.stringList());
    }
}
```

Output:

```
array = Earth, Jupiter, Neptune, Jupiter, Uranus, Pluto
array = Earth, Neptune, Uranus, Pluto
```

A brief description of each instantiable container is given below.

`SList`
: This is a singly linked list. Fast inserting and removing of elements. Can only be traversed in the forward direction.

`DList`
: This is a doubly linked list. Fast inserting and removing of elements. Can be traversed both forwards and backwards. Consumes more memory than the `DList`.

`IndexedTree`
: A tree with indexed nodes, functioning as any other indexed sequence. Operations like inserting and removing are faster than for an array, but finding an indexed element is slower.

`Array`
: This data structure stores its element in a plain array. Fast access elements at arbitary positions. It can add and remove elements at end in constant time if it does not need to resize the internal array.

`Deque`
: Same functionality as `Array`, but can also add and remove elements at the beginning in constant time.

`HAT`
: Same functionality as `Array`, but consumes less memory when resized.

`GenericTree`
: This stores the elements in a tree structure without automatic ordering.

`Set`
: This is a set of elements.

`OrderedSet`
: This is a ordered set of elements.

`Map`
: This is a set that associates a key to a value.

`OrderedMap`
: This is a set that associates a key to a value. The set is ordered on the value.

`BinarySearchTree`
: A basic binary search tree.

`RedBlackTree`
: An improved binary search tree that maintain a roughly balanced tree using color as its balancing condition.

`SplayTree`
: An improved binary search tree that tries to maintain a roughly balanced tree using the "splay" operation. In addition, it moves nodes that are referenced often nearer to the root.

3.2 Iterators

The iterators are objects that make it possible to iterate through a sequence of elements. Usually the iterators are used to traverse the elements of a container, but they don't have to be tied to a container. For example, there are iterators that iterate only on input and output streams.

There are five basic types of iterators defined: `OutputIterator`, `InputIterator`, `ForwardIterator`, `BidirectionalIterator` and `RandomAccessIterator`. The most general of these are the `InputIterator` and the `OutputIterator`, and they are specialized by their subclasses. Be aware that they should only be used with one pass algorithms. Consider an `InputStreamIterator` reading values from the console. It would not be possible to have an algorithm that reads all the values from the console, and then resets the iterator and starts all over again to do some more processing on the values read. For this one would have to store the elements in some container that supports a `ForwardIterator`. This makes more sense, as it is not the iterators responsibility to store the elements so that they may be used again in the future.

A brief description of each basic iterator is given below.

`OutputIterator`
: This is the base iterator that write one element at a time. It may only move in the forward direction.

`InputIterator`
: This is the base iterator that read one element at a time. It may only move in the forward direction.

`ForwardIterator`
: This is the base iterator that can both read and write one

element at a time. Like `InputIterator` and `OutputIterator` it may only move in the forward direction.

`BidirectionalIterator`
: This is the base iterator that, like the `ForwardIterator`, both read and write elements. In addition, it may move in both the forward and backward direction.

`RandomAccessIterator`
: This is the base iterator that in addition to the properties it inherits from BidirectionalIterator is capable of accessing any element in any direction.

A very simple example of an iterator in action is given below.

Example: `Overview2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;

public class Overview2
{
    public static void main(String[] args)
    {
        Array array = new Array();

        array.add("Earth");
        array.add("Jupiter");
        array.add("Neptune");
        array.add("Uranus");
        array.add("Pluto");

        System.out.println("array = " + array.stringList());

        ForwardIterator i = array.begin();
        i.inc();
        i.inc();

        System.out.println("Second element after array.begin() = "+
                           i.get());
    }
}
```

Output:

```
array = Earth, Jupiter, Neptune, Uranus, Pluto
Second element after array.begin() = Neptune
```

3.3 Generic Algorithms

CAL provide a rich set of fast and effiecient algorithms to ease the burden of the programmer. All the algorithms work on a range of elements defined by two iterators, `[first, last)`. This results in the algorithms being generic, meaning that they will work on a set of elements, independent of the underlying data structure. All they depend upon are the properties of the iterators used. For example, a quick sort algorithm only depends upon having random access to any element. Thus, the sort operation take two `RandomAccessIterator`s as arguments.

The algorithms have been put in separate classes according to their classification, and are implemented as static methods. The following list shows the different classes of algorithms, and gives a short description of each.

`HeapOperations`
: Defines certain methods that allow a range to be treated as a heap. A heap is a data structure that make it possible to insert an element and extract the largest element in $O(\log n)$.

`Mutating`
: Defines certain mutative algorithms, ie. algorithms that change the elements in the range upon which they operate.

`NonMutating`
: Defines certain algorithms that leave the elements in the range they operate upon intact and unchanged.

`Numeric`
: Defines algorithms especially intended for operating on numeric elements.

`Searching`
: Defines algorithms that rely on the elements to be sorted to find elements in $O(\log n)$ time.

`SetOperations`
: Defines certain algorithms commonly associated with sets. Requires the elements to be sorted.

`Sorting`
: Defines a couple of different sorting algorithms.

The following example shows how the algorithms are used, and in particular, it shows the difference between merge and union.

Example: `Overview3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.adaptor.*;

public class Overview3
{
```

```
        public static void main(String[] args)
        {
            Array array1 = new Array();
            Array array2 = new Array();

            array1.add("Jupiter");
            array1.add("Neptune");
            array1.add("Uranus");
            array1.add("Pluto");

            array1.sort(new LessString());

            array2.add("Phobos");
            array2.add("Deimos");

            array2.sort(new LessString());

            System.out.println("array1 = " + array1.stringList());
            System.out.println("array2 = " + array2.stringList());

            System.out.print("Result of merge = ");
            SetOperations.merge(array1.begin(), array1.end(),
                                array2.begin(), array2.end(),
                                new OutputStreamIterator(" "),
                                new LessString());

            System.out.print("\nResult of union = ");
            SetOperations.union(array1.begin(), array1.end(),
                                array2.begin(), array2.end(),
                                new OutputStreamIterator(" "),
                                new LessString());
            System.out.println();
        }
    }
```

Output:

```
array1 = Jupiter, Neptune, Pluto, Uranus
array2 = Deimos, Phobos
Result of merge = Deimos Jupiter Neptune Phobos Pluto Uranus
Result of union = Deimos Jupiter Neptune Phobos Pluto Uranus
```

3.4 Function and Predicate Objects

A function object is an object that generates a new object from the one you put into it. Functions come as `UnaryFunction`, which take one argument, and `BinaryFunction`, which takes two arguments. Examples of the `UnaryFunction` object are the `StringLength`, `ToLowerCase`, `ToUpperCase` and `Negate`, while examples of the `BinaryFunction` object are the `Plus` and `Minus`.

A predicate object will, in contrast to a function object, return a boolean value rather than a new object. They are useful to check whether the argument object(s) satisfy a set of conditions. As with function objects, there is both a unary and a binary version, `UnaryPredicate` and `BinaryPredicate`. Examples of the `UnaryPredicate` object is the `InstanceOf`. Examples of the `BinaryPredicates` are `Greater`, `Less` and `Equals`.

The following example converts all string elements in a container to upper case.

Example: `Overview4.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import java.io.*;

public class Overview4
{
    public static void main(String[] args)
    {
        String[] str = {"Cordelia", "Ophelia", "Bianca",
                        "Cressida", "Desdemona", "Juliet"};
        Sequence uranusMoons = new Array(str);

        System.out.println("before = " + uranusMoons.stringList());
        uranusMoons.transform(new ToUpperCase());
        System.out.println("after  = " + uranusMoons.stringList());
    }
}
```

Output:

```
before = Cordelia, Ophelia, Bianca, Cressida, Desdemona, Juliet
after  = CORDELIA, OPHELIA, BIANCA, CRESSIDA, DESDEMONA, JULIET
```

3.5 Generators

A generator object is a very simple object that will return some object whenever it's generate method is called. Still, it turns out to be fairly useful for many different purposes. `RandomNumberGenerator` is one example of how one might make a generator.

The following example shows one use of the generator.

Example: `Overview5.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.adaptor.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.misc.*;
```

```
public class Overview5
{
    public static void main(String[] args)
    {
        Array array = new Array();

        Mutating.generateN(new BackInsertIterator(array),
                           5, new RandomNumberGenerator(10));

        System.out.println("array = " + array.stringList());
    }
}
```

Output:

```
array = 1, 1, 2, 0, 5
```

3.6 Pattern Matchers

There are two available pattern matchers in CALİt is the `WildcardMatcher` and the `RegExpMatcher`. The first one is a very simple pattern matcher based on the UNIX wildcards. The second is a full featured Perl-like regular expression matcher.

The following example demonstrates the power of regular expression matching.

Example: `Overview6.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.matcher.*;

public class Overview6
{
    public static void main(String[] args)
    {
        String str[] = {"Smith,    John K.",
                        "Nelson,   Cliff",
                        "Collins,  Joseph A."};
        String pattern = "^([^,]*),\\s*(.*)$";
        Array names = new Array(str);

        System.out.println("before replace =\n  " +
                           names.stringList("\n  "));

        RegExpMatcher matcher = new RegExpMatcher(pattern);
        names.transform(new ReplacePattern(matcher, "\\2 \\1"));

        System.out.println("after replace  =\n  " +
                           names.stringList("\n  "));
    }
}
```

Output:

```
before replace =
  Smith,    John K.
  Nelson,   Cliff
  Collins,  Joseph A.
after replace  =
  John K. Smith
  Cliff Nelson
  Joseph A. Collins
```

3.7 Adaptors

An adaptor is a device for connecting two parts that were not originally designed to be connected. CAL have three different categories of adaptors, the container adaptors, the iterator adaptors and the function adaptors.

3.7.1 Container Adaptors

The following are the container adaptors defined by CAL.

`Stack`
: This is a last in first out (LIFO) queue. It can use any class that implement the `BackStackable` or `FrontStackable` interface to store its elements.

`Queue`
: This is a first in first out (FIFO) queue. It can use any class that implement the `BackQueueable` or `FrontQueueable` interface to store its elements.

`PriorityQueue`
: This class will always return the greatest element from the queue when calling dequeue.

`<type>Array`
: These container adaptors make it possible to use a native Java array just like one might use the CAL-`Array`.

The following example shows the use of a container adaptor.

Example: `Overview6.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.matcher.*;

public class Overview6
{
    public static void main(String[] args)
    {
```

```
        String str[] = {"Smith,    John K.",
                        "Nelson,   Cliff",
                        "Collins,  Joseph A."};
        String pattern = "^([^,]*),\\s*(.*)$";
        Array names = new Array(str);

        System.out.println("before replace =\n  " +
                           names.stringList("\n  "));

        RegExpMatcher matcher = new RegExpMatcher(pattern);
        names.transform(new ReplacePattern(matcher, "\\2 \\1"));

        System.out.println("after replace  =\n  " +
                           names.stringList("\n  "));
    }
}
```

Output:

```
before replace =
  Smith,    John K.
  Nelson,   Cliff
  Collins,  Joseph A.
after replace  =
  John K. Smith
  Cliff Nelson
  Joseph A. Collins
```

3.7.2 Iterator Adaptors

The different iterator adaptors defined by CAL is given in the list below.

`Reverse Iterators`
: These iterators iterate in the opposite direction as that of the original iterator.

`Insert Iterators`
: These iterators are output iterators that will insert elements into a container rather than replacing an element when set is called.

`<type>ArrayIterator`
: These iterators make it possible to iterate through a native Java array just like one might iterate through the CAL-`Array`.

The following example shows the use of an iterator adaptor.

Example: `Overview7.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.adaptor.*;
import COM.X3M.cal.function.*;
```

```
public class Overview7
{
    public static void main(String[] args)
    {
        PriorityQueue pqueue =
            new PriorityQueue(new LessNumber());
        pqueue.enqueue(new Integer(4));
        pqueue.enqueue(new Integer(7));
        pqueue.enqueue(new Integer(3));
        pqueue.enqueue(new Integer(5));

        System.out.print("output = " + pqueue.dequeue());
        while (!pqueue.isEmpty())
            System.out.print(", " + pqueue.dequeue());
        System.out.println();
    }
}
```

Output:

```
output = 7, 5, 4, 3
```

3.7.3 Function Adaptors

To further enhance the usefulness of all the function objects defined, a couple of function adaptors have been supplied. They make it possible to combine existing function objects in new ways to end up with entirely new function objects.

The following example shows the use of function adaptors.

Example: `Overview8.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.adaptor.*;
import COM.X3M.cal.iterator.*;

public class Overview8
{
    public static void main(String[] args)
    {
        Sequence slist = new SList();
        slist.add(new Integer(1));
        slist.add(new Integer(2));
        slist.add(new Integer(3));

        ForwardIterator f = slist.begin();
        f.inc();

        InsertIterator i = new InsertIterator(slist, f);
        i.set("a");
```

```
            i.set("b");
            i.set("c");

            System.out.println("result = " + slist.stringList());
        }
    }
```

Output:

```
result = 1, a, b, c, 2, 3
```

CHAPTER 4

Basic Components

CAL depend upon a few basic components to simplify the use of the library. They are depicted in the diagram below.

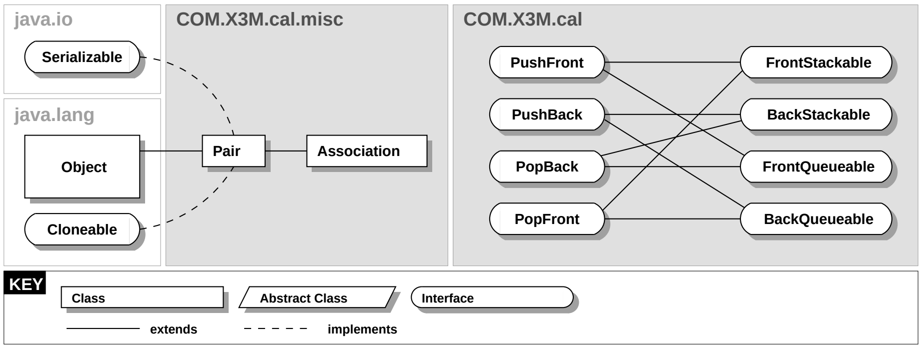

Figure 4.1: Basic Components Hierarchy

4.1 Pair and Association

The `Pair` wraps up two arbitrary objects into one, allowing both objects to be stored as a single item in any container. You may use the two fields `first` and `second` to access the objects at any time. In addition, this class overrides the `hashCode` and `equals` method in `Object`, taking into account both objects in the `Pair`.

The `Association` should be used rather than `Pair` when the first object is the value that identifies the `Pair`. In other words, two `Association` objects whose `first` fields are equal, will also be equal, regardless of whether the `second` fields are equal.

4.2 Miscellaneous Interfaces

The following is a short list of interfaces along with a short description. Note that they should only be implemented for classes that can support them in constant time.

- `PushFront` defines a method `pushFront` that insert an element to the front of the container.
- `PushBack` defines a method `pushBack` that insert an element to the back of the container.
- `PopFront` defines two methods `front` and `popFront`. `front` returns the first element in the container, while `popFront` will remove the same element.
- `PopBack` defines two methods `back` and `popBack`. `back` returns the last element in the container, while `popBack` will remove the same element.
- `FrontStackable` extends the `PushFront` and `PopFront` interfaces. In addition, it defines the two methods `size` (returns the size of the container) and `isEmpty` (returns true if the container is empty).
- `BackStackable` extends the `PushBack` and `PopBack` interfaces. In addition, it defines the two methods `size` (returns the size of the container) and `isEmpty` (returns true if the container is empty).
- `FrontQueueable` extends the `PushFront` and `PopBack` interfaces. In addition, it defines the two methods `size` and `isEmpty` whose function should be clear from their names.
- `BackQueueable` extends the `PushBack` and `PopFront` interfaces. In addition, it defines the two methods `size` and `isEmpty` whose function should be clear from their names.

The `PushFront` and `PushBack` interfaces are used by the `FrontInsertIterator` and the `BackInsertIterator`, while the `*Stackable` and `*Queueable` interfaces are used by the `Stack` and the `Queue` respectively.

CHAPTER 5

Containers

The `Container` class is the core component of CAL. Every data structure in the library, except some container adaptors, extends the `Container` class. It defines a set of methods that are common to all containers.

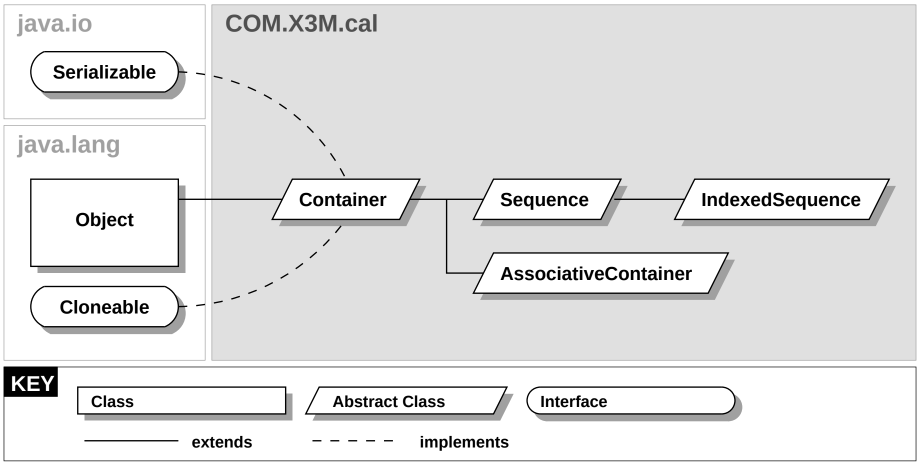

Figure 5.1: Container Hierarchy

Not all the functionality is defined in the base `Container` class, but is extended in the subclasses further down in the hierarchy. For example, `Sequence` has common functionality for lists, `IndexedSequence` has common functionality for arrays, and `AssociativeContainer` has common functionality for associatives.

All containers classes has a number of common methods that can be used:

`add:`	Adds an element to the container.
`begin:`	Returns an iterator positioned at the first element of the container.
`clear:`	Removes every element from the container.
`clone:`	Returns a shallow copy of the container.

`elements:`	Returns an `java.util.Enumeration` positioned at the first element of the container.
`end:`	Returns an iterator positioned at the last element of the container.
`equals:`	Returns true if the container contents.
`insert:`	Inserts an element at any position in the container.
`isEmpty:`	Returns true if the container is empty.
`list:`	Lists all elements in the container.
`size:`	Returns the number of elements in the container.
`stringList:`	Returns a string with all elements in the container.
`swap:`	Swaps the container contents with another.
`toString:`	Returns a string that describes the container.

All the above methods are described in the following sections.

5.1 Storing

Every container allows[1] you to store objects. This is either done with the `add` or `insert` method.

5.1.1 Adding and Inserting Elements

Below is a simple examples of how to add elements in a container.

Example: `Container1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;

public class Container1
{
    public static void main(String[] args)
    {
        Container container = new Array();

        container.add("Venus");
        container.add("Jupiter");
        container.add("Pluto");

        System.out.println("contents = " + container.stringList());
    }
}
```

Output:

```
contents = Venus, Jupiter, Pluto
```

[1] Of course, this is one of the *most* useful features.

To insert an element into a container at any position, the `insert` method does the job. This method requires an iterator which is discussed in more detail in chapter 10. The following example shows how to insert an element at the beginning of a list.

Example: `Container2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;

public class Container2
{
    public static void main(String[] args)
    {
        Container container = new DList();

        container.add("Venus");
        container.add("Jupiter");
        container.add("Pluto");

        System.out.println("Before insert = " +
                           container.stringList());
        container.insert(container.begin(), "Mars");
        System.out.println("After insert  = " +
                           container.stringList());
    }
}
```

Output:

```
Before insert = Venus, Jupiter, Pluto
After insert  = Mars, Venus, Jupiter, Pluto
```

A container can only store instances of Java classes, not primitives. For example, you can store `String`, a `Date`, and an `Integer` in the same container. When retrieving an object, it must be type-castedback to the required type before applying a type-specific method.

✍ Unlike STL containers, it is not possible to create a JDK or CAL container that can only hold instances of a specific type of object.

✍ Java's dynamic cast-checking mechanism ensures that any attempt to cast an object to the wrong type causes a runtime exception.

5.1.2 Storing Java Primitives

When storing Java primitives it must be wrapped in their object equivalents. Table 5.1 shows a list of primitives and their object equivalents.

The following example shows how a singly linked list can be used to store variety of primitives.

Example: `Container3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
```

Table 5.1: Object Equivalents of Java Primitives

Primitive	*Object Equivalent*	*Notes*
`boolean`	`java.lang.Boolean`	
`byte`	`java.lang.Byte`	Requires JDK1.1
`char`	`java.lang.Character`	
`double`	`java.lang.Double`	
`float`	`java.lang.Float`	
`int`	`java.lang.Integer`	
`long`	`java.lang.Long`	
`short`	`java.lang.Short`	Requires JDK1.1

```
public class Container3
{
    public static void main(String[] args)
    {
        Container container = new SList();

        container.add(new Double(3.14));
        container.add(new Character('A'));
        container.add(new Boolean(true));

        System.out.println("contents = " + container.stringList());
    }
}
```

Output:

```
contents = 3.14, A, true
```

5.1.3 Storing User-Defined Objects

There are three specific methods that are used by containers on the objects they contain.

`boolean equals(Object object)`
: Return true if this object is equal to the specified object. By default, CAL container and algorithms use this function when comparing elements.

`hashCode()`
: Return the hash code of this object. `Map` and `Set` use this value to store and locate objects. `OrderedMap` and `OrderedSet` uses this value by default to order their elements. If two objects match using `equals()`, their hash code must be identical.

`String toString()`
: Return a string that describes this object in readable format. Containers use this method to describe their objects.

Each of the listed methods has a default implementation in `java.lang.Object`. When storing used defined objects in a container, these methods should be overridden if necessary.

The following class, named `Planet`, overrides each of these methods. If two different `Planet` objects has the same name, they are considered equal.

Example: `Planet.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;

public class Planet
{
    String name;

    public Planet(String name)
    {
        this.name = name;
    }

    public int hashCode()
    {
        return this.name.hashCode();
    }

    public boolean equals(Object object)
    {
        return (object instanceof Planet) &&
            this.name.equals(((Planet)object).name);
    }

    public String toString()
    {
        return this.name;
    }
}
```

The following example stores instances of `Planet` into a `Map` as a key.

Example: `Container4.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;

public class Container4
{
    public static void main(String[] args)
    {
        Planet planet1 = new Planet("Uranus");
        Planet planet2 = new Planet("Pluto");
```

```
            Map map = new Map();
            map.add(planet1, "Umbriel");
            map.add(planet2, "Charon");

            System.out.println(planet2 + "'s moon is " +
                               map.get(planet2));
        }
    }
```

Output:

```
Pluto's moon is Charon
```

5.2 Iterating

All containers support two methods of iterating trough elements. The first method is using the JDK `java.util.Enumeration` interface, and the other preferred method is using CAL iterators. Iterators are more powerful than the JDK Enumeration type. For example, a `ReverseIterator` allows backward iterating through a container. Iterators will be covered in more details in chapter 10.

When traversing elements you should use the CAL iterator types instead of the JDK Enumeration type.

The following example demonstrates the use of the JDK Enumeration type to iterator through the container.

Example: `Container5.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import java.util.Enumeration;

public class Container5
{
    public static void main(String[] args)
    {
        Container container = new Array();

        container.add("Jupiter");
        container.add("Mars");
        container.add("Venus");

        Enumeration iterator = container.elements();
        while (iterator.hasMoreElements())
            System.out.println(iterator.nextElement());
    }
}
```

Output:

```
Jupiter
Mars
Venus
```

The following example demonstrates the more powerful way of iterating through the container.

Example: `Container6.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;

public class Container6
{
    public static void main(String[] args)
    {
        Array array = new Array();

        array.add("Jupiter");
        array.add("Mars");
        array.add("Venus");

        System.out.println("Forward iterating:");
        ForwardIterator iterator = array.begin();
        while (!iterator.atEnd())
            System.out.println("  " + iterator.nextElement());

        System.out.println("Backward iterating:");
        iterator = array.reverseBegin();
        while (!iterator.atEnd())
            System.out.println("  " + iterator.nextElement());
    }
}
```

Output:

```
Forward iterating:
  Jupiter
  Mars
  Venus
Backward iterating:
  Venus
  Mars
  Jupiter
```

5.3 Clearing and Size Measuring

Every container can remove all elements with the `clear` method. It can also return its size with the `size` method. `isEmpty` returns true if the container is empty. The following example demonstrates these three methods.

Example: `Container7.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;

public class Container7
{
```

```
    public static void main(String[] args)
    {
        Container container = new Deque();
        container.add("Uranus");
        container.add("Pluto");
        container.add("Mars");

        System.out.println("container contents  = " +
                           container.stringList());
        System.out.println("container.isEmpty() = " +
                           container.isEmpty());
        System.out.println("container.size()    = " +
                           container.size());

        container.clear();

        System.out.println("Clearing container...");
        System.out.println("container.isEmpty() = " +
                           container.isEmpty());
        System.out.println("container.size()    = " +
                           container.size());
    }
}
```

Output:

```
container contents  = Uranus, Pluto, Mars
container.isEmpty() = false
container.size()    = 3
Clearing container...
container.isEmpty() = true
container.size()    = 0
```

5.4 Cloning and Swapping

A *deep copy* is not possible since the `clone` method in the `java.lang.Object` class is protected by default.

All containers can be cloned using the `clone` method. This returns a *shallow copy*, which means that the original container and the copy share references to the same objects. This is a very fast way to copy a container and is sufficient for most programming purposes.

Two containers can swap their contents using the `swap` method. All the elements is swapped efficiently between two containers.

The following example demonstrates cloning and swapping.

Example: `Container8.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;

public class Container8
{
    public static void main(String[] args)
```

```
    {
        Container containerA = new DList();
        containerA.add("Mars");
        containerA.add("Venus");
        System.out.println("containerA = " + containerA);

        Container containerB = (Container)containerA.clone();
        System.out.println("containerB = containerA.clone() = " +
                           containerB);

        containerB.insert(containerB.begin(), "Hyperion");
        containerB.add("Pluto");
        System.out.println("containerB = " + containerB);

        System.out.println("Swapping containerA <-> containerB");
        containerA.swap(containerB);
        System.out.println("containerA = " + containerA);
        System.out.println("containerB = " + containerB);
    }
}
```

Output:

```
containerA = COM.X3M.cal.DList[ Mars, Venus ]
containerB = containerA.clone() = COM.X3M.cal.DList[ Mars, Venus ]
containerB = COM.X3M.cal.DList[ Hyperion, Mars, Venus, Pluto ]
Swapping containerA <-> containerB
containerA = COM.X3M.cal.DList[ Hyperion, Mars, Venus, Pluto ]
containerB = COM.X3M.cal.DList[ Mars, Venus ]
```

5.5 Comparing

All containers can be compared for equality using `equals`. It can also be compared using `less`, `lessEqual`, `greater` and `greaterEqual` which is discussed in Chapter 12. The following example shows the use of the `equals` method.

Example: `Container9.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;

public class Container9
{
    public static void main(String[] args)
    {
        Container containerA = new Array();
        containerA.add(new Integer(4));
        containerA.add(new Integer(2));
        containerA.add(new Integer(7));
        Container containerB = (Container)containerA.clone();

        System.out.println(containerA + " == " + containerB +
```

```
                            " is " + containerA.equals(containerB));

        containerB.begin().set(new Integer(3));

        System.out.println(containerA + " == " + containerB +
                            " is " + containerA.equals(containerB));
    }
}
```

Output:

```
COM.X3M.cal.Array[ 4, 2, 7 ] == COM.X3M.cal.Array[ 4, 2, 7 ] is true
COM.X3M.cal.Array[ 4, 2, 7 ] == COM.X3M.cal.Array[ 3, 2, 7 ] is false
```

5.6 Printing

In the previous examples it is only used the `toString` method for printing. CAL has many other printing methods. These methods are summarized below.

`void list()`
: Prints a comma-separated list of the elements to `System.out` stream.

`void list(PrintStream)`
: Same as `list()`, but uses the specified stream.

`void list(PrintStream, String)`
: Same as `list()`, but uses the specified stream and separation.

`void list(String)`
: Same as `list()`, but uses the specified separation.

`String stringList()`
: Returns a comma-separated list of the elements.

`String stringList(String)`
: Same as `stringList()`, but uses the specified separation.

`String toString()`
: Returns a string representation.

The following example uses some of the printing methods.

Example: `Container10.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;

public class Container10
{
    public static void main(String[] args)
    {
        Container container = new Array();
        container.add("Mercury");
```

```
        container.add("Venus");
        container.add("Earth");
        container.add("Mars");

        System.out.print("Output using list()    : ");
        container.list();
        System.out.print("\nOutput using list(\":\") : ");
        container.list(":");

        System.out.println("\nOutput using stringList()    : " +
                            container.stringList());
        System.out.println("Output using stringList(\":\") : " +
                            container.stringList(":"));
    }
}
```

Output:

```
Output using list()    : Mercury, Venus, Earth, Mars
Output using list(":") : Mercury:Venus:Earth:Mars
Output using stringList()    : Mercury, Venus, Earth, Mars
Output using stringList(":") : Mercury:Venus:Earth:Mars
```

5.7 Serialization

The `Container` class implements the `java.io.Serialization` interface. This means that all containers the ability to be serialized. This means that the container and it's contents can be streamed out over a socket or into a persistent file. To be able to do this all the container data must be serializable.

Requires JDK1.1

The following example demonstrates how to serialize a container to a persistent file, and then retrieve it.

Example: `Container11.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import java.io.*;
import java.util.*;

public class Container11
{
    static void write()
    {
        Container container = new Array();
        container.add("\n  Hello World!");
        container.add(new Date());
        container.add(new Boolean(true));
        System.out.println("Container = " + container);

        System.out.println("Serializing array...");
        try
        {
```

```
            FileOutputStream fileStream =
                new FileOutputStream("Array12");
            ObjectOutputStream objStream =
                new ObjectOutputStream(fileStream);
            objStream.writeObject(container);
        }
        catch(IOException e)
        {
            System.out.println("Got exception while " +
                               "serializing.\n" + e);
            System.exit(1);
        }
        System.out.println("Completed.");
    }

    static void read()
    {
        System.out.println("Loading serialized array...");
        try
        {
            FileInputStream fileStream =
                new FileInputStream("Array12");
            ObjectInputStream objStream =
                new ObjectInputStream(fileStream);
            Container container =
                (Container)objStream.readObject();
            System.out.println("Completed.");
            System.out.println("Container = " + container);
        }
        catch(Exception e)
        {
            System.out.println("Got exception while restoring " +
                               "array.\n" + e);
            System.exit(1);
        }
    }

    public static void main(String[] args)
    {
        write();

        read();
    }
}
```

Output:

```
Container = COM.X3M.cal.Array[
  Hello World!, Thu Mar 20 19:23:43 PST 1997, true ]
Serializing array...
Completed.
Loading serialized array...
Completed.
Container = COM.X3M.cal.Array[
  Hello World!, Thu Mar 20 19:23:43 PST 1997, true ]
```

5.8 Exception Handling

All CAL errors causes a Java exception to be thrown. Methods that can cause an exception are commented in the API reference. CAL throws standard JDK exceptions that are defined in `java.lang`.

`java.lang.IllegalArgumentException`
: An illegal argument was supplied to a method.

`java.lang.IndexOutOfBoundsException`
: An illegal index was supplied to a method.

`java.lang.NullPointerException`
: Attempt to add a null key or value.

The following example catches all of the exceptions.

Example: `Container12.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;

public class Container12
{
    public static void main(String[] args)
    {
        Container container = new DList();
        container.add("Mercury");
        container.add("Venus");
        container.add("Earth");
        System.out.print("Parsing elements: ");
        ForwardIterator iterator = container.begin();
        try
        {
            while(true)
            {
                System.out.print(iterator.nextElement() + ", ");
            }
        }
        catch(Exception e)
        {
            System.out.println("\n ** " + e);
        }
    }
}
```

Output:

```
Parsing elements: Mercury, Venus, Earth,
 ** java.lang.IndexOutOfBoundsException: Iterator is past-the-end
```

CHAPTER 6

Sequences

A `Sequence` is a container that organizes its elements in a strictly linear fashion. There are tree concrete classes derived directly from `Sequence`, namely `SList`, `DList` and `IndexedTree`. `SList` implements a singly linked list, `DList` implements a doubly linked list, and `IndexedTree` implements an indexed based tree.

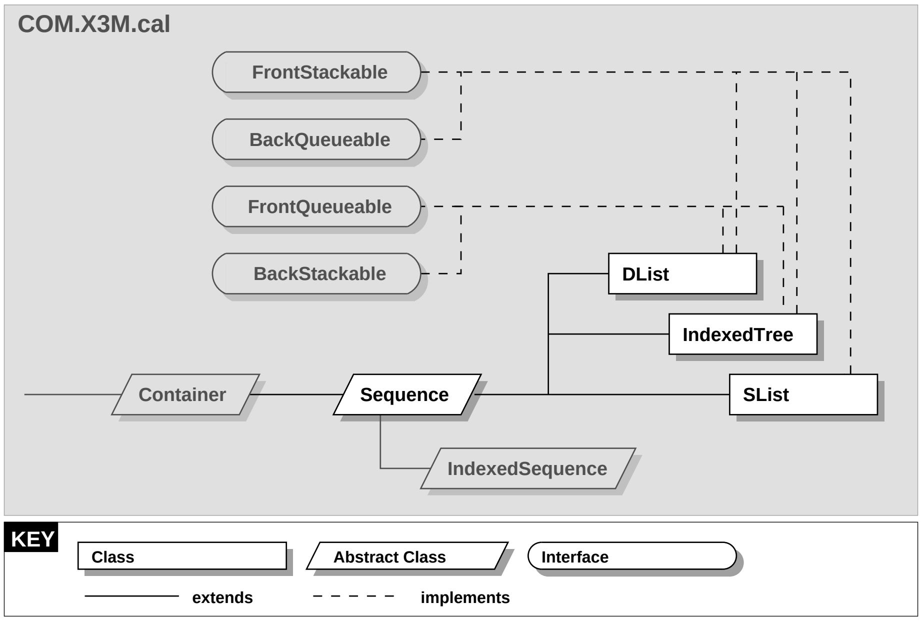

Figure 6.1: Sequence Hierarchy

The new methods introduced by `Sequence` are:

`insert:`	Inserts one or more elements before the specified position.
`erase:`	Erases the element or range of elements at the specified position(s).
`findIndex:`	Returns an iterator positioned at the nth element.

`remove:`	Removes and erases the elements matching the specified criteria.
`unique:`	Removes and erases all but the first element in every consecutive range of equal elements.
`sort:`	Sorts the elements.
`replace:`	Replaces certain elements with another element.
`transform:`	Transforms all the elements in the sequence with a function.
`reverse:`	Reverses the order of the elements in the sequence.
`rotate:`	Rotates the elements in the sequence a number of positions.
`count:`	Returns the number of elements in the sequence that match a certain criteria.
`front:`	Returns the first element in the sequence.
`back:`	Returns the last element in the sequence.
`pushBack:`	Appends an element to the end of the sequence.

6.1 Storing elements

There are basically two ways of storing elements in a `Sequence`. Either the elements may be added without specifying where in the `Sequence` they should be added. This way, the elements will be inserted at the end.

Another way of adding elements is by specifying where to insert the elements with an iterator. This allows a more controlled way of inserting the elements. The following example illustrates these two ways of inserting elements.

Example: `Sequence1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;

public class Sequence1
{
    public static void main(String[] args)
    {
        Sequence sequence = new SList();

        sequence.add("Venus");
        sequence.add("Jupiter");
        sequence.add("Pluto");

        System.out.println("before = " + sequence.stringList());

        ForwardIterator i = sequence.find("Jupiter");
        // inserting an element before the iterator i,
        // positioned at "Jupiter"
```

```
            sequence.insert(i, "Mars");
            i.inc();
            i.inc();
            // inserting an element before the iterator i,
            // now positioned at "Pluto"
            sequence.insert(i, "Uranus");
            System.out.println("after  = " + sequence.stringList());
        }
    }
```

Output:

```
before = Venus, Jupiter, Pluto
after  = Venus, Mars, Jupiter, Uranus, Pluto
```

6.2 Removing elements

At times it might be useful to be able to remove some of the elements stored in a `Sequence`. Three ways of doing this has been provided to cope with such needs.

The first way allows specific elements to be removed by supplying an iterator, or two iterators if specifying a range. The methods that do it this way are named `erase`.

Example: `Sequence2a.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;

public class Sequence2a
{
    public static void main(String[] args)
    {
        Sequence sequence = new DList();

        sequence.add("Venus");
        sequence.add("Mars");
        sequence.add("Io");
        sequence.add("Mars");
        sequence.add("Mars");
        sequence.add("Mars");

        System.out.println("before             = " +
                           sequence.stringList());
        sequence.erase(sequence.find("Mars"));
        System.out.println("after first erase  = " +
                           sequence.stringList());
        sequence.erase(sequence.findIndex(3), sequence.end());
        System.out.println("after second erase = " +
                           sequence.stringList());
    }
```

```
}
```

Output:

```
before             = Venus, Mars, Io, Mars, Mars, Mars
after first erase  = Venus, Io, Mars, Mars, Mars
after second erase = Venus, Io, Mars
```

Secondly, it is possible to remove all elements that match a certain criteria. These methods are named `remove`.

Example: `Sequence2b.java`

```
        ...
        System.out.println("before = " + sequence.stringList());
        sequence.remove("Mars");
        System.out.println("after  = " + sequence.stringList());
        ...
```

Output:

```
before = Venus, Mars, Io, Mars, Mars, Mars
after  = Venus, Io
```

Finally, one may remove all but the first element from every consecutive range of equal elements. For this, use the methods named `unique`.

Example: `Sequence2c.java`

```
        ...
        System.out.println("before = " + sequence.stringList());
        sequence.unique();
        System.out.println("after  = " + sequence.stringList());
        ....
```

Output:

```
before = Venus, Mars, Io, Mars, Mars, Mars
after  = Venus, Mars, Io, Mars
```

6.3 Miscellaneous Operations

In addition to the basic operations of adding and removing elements, the abstract class `Sequence` also define a number of other operations that are quite useful at occasions.

6.3.1 Counting, Replacing and Transforming

The `count` method returns the number of elements that match a certain criteria. The operation replace replaces every element satisfying a certain criteria with a new object. The criteria may be an `Object` or a `UnaryPredicate`. The `transform` method applies a function to every element in the `Sequence`, and replaces that element with the value returned from the function.

Example: `Sequence3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;
```

```
import COM.X3M.cal.*;
import COM.X3M.cal.function.*;

public class Sequence3
{
    public static void main(String[] args)
    {
        Sequence sequence = new DList();

        sequence.add("Venus");
        sequence.add("Jupiter");
        sequence.add("Jupiter");
        sequence.add("Pluto");

        System.out.println("before                    = " +
                           sequence.stringList());
        System.out.println("sequence.count(\"Jupiter\") = " +
                           sequence.count("Jupiter"));
        sequence.replace("Jupiter", "Saturn");
        System.out.println("after replace             = " +
                           sequence.stringList());
        sequence.transform(new ToLowerCase());
        System.out.println("after transform           = " +
                           sequence.stringList());
    }
}
```

Output:

```
before                    = Venus, Jupiter, Jupiter, Pluto
sequence.count("Jupiter") = 2
after replace             = Venus, Saturn, Saturn, Pluto
after transform           = venus, saturn, saturn, pluto
```

6.3.2 Reversing and Rotating

The `reverse` method reverses the order of the elements in the `Sequence`, while `rotate` rotates the elements so that the first element is the supplied iterator.

Example: `Sequence4.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;

public class Sequence4
{
    public static void main(String[] args)
    {
        Sequence sequence = new SList();

        sequence.add("Mars");
```

```
        sequence.add("Venus");
        sequence.add("Pluto");
        sequence.add("Jupiter");
        sequence.add("Earth");

        System.out.println("before        = " +
                           sequence.stringList());
        sequence.rotate(sequence.find("Pluto"));
        System.out.println("after rotate  = " +
                           sequence.stringList());
        sequence.reverse();
        System.out.println("after reverse = " +
                           sequence.stringList());
    }
}
```

Output:

```
before        = Mars, Venus, Pluto, Jupiter, Earth
after rotate  = Pluto, Jupiter, Earth, Mars, Venus
after reverse = Venus, Mars, Earth, Jupiter, Pluto
```

6.3.3 Sorting

Sometimes it is useful to rearrange the order of the elements in a `Sequence` with respect to some criteria. This may be done in an efficient manner for all sequences, and as a consequence, all subclasses of `Sequence` implement a sort method. By default it will sort the elements in ascending order based on the hash value of the elements. This may be overridden by supplying a `BinaryPredicate` that compares the relative order of two elements, and returns true if the first element is to be put before the second element.

Example: `Sequence5.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;

public class Sequence5
{
    public static void main(String[] args)
    {
        Sequence sequence = new DList();

        sequence.add("Mars");
        sequence.add("Venus");
        sequence.add("Pluto");
        sequence.add("Jupiter");
        sequence.add("Earth");

        System.out.println("before = " + sequence.stringList());
        sequence.sort(new LessString());
```

```
        System.out.println("after  = " + sequence.stringList());
    }
}
```

Output:

```
before = Mars, Venus, Pluto, Jupiter, Earth
after  = Earth, Jupiter, Mars, Pluto, Venus
```

6.4 Lists

CAL comes with two types of lists, singly linked lists (`SList`), and doubly linked lists (`DList`). Lists allow constant time insertion at any point in the list. Only sequential access to the elements are possible however. Contrast this with `IndexedSequence`, that allow random access to all the elements, but insertion and removal of elements is only done in linear time.

Both `SList` and `DList` introduce a new operation splice. This is an operation that destructively moves one or more elements from one list to another. This is a very efficient operation compared to a normal insert operation, as it is constant time, no matter how many elements are being spliced over.

Example: `Sequence6.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;

public class Sequence6
{
    public static void main(String[] args)
    {
        SList list1 = new SList();
        SList list2 = new SList();

        list1.add("Earth");
        list1.add("Mars");
        list1.add("Jupiter");

        list2.add("Io");
        list2.add("Phobos");

        System.out.println("list1 before = " + list1.stringList());
        System.out.println("list2 before = " + list2.stringList());

        ForwardIterator i = list1.find("Jupiter");
        i.inc();

        list1.splice(i, list2);
        System.out.println("list1 after  = " + list1.stringList());
        System.out.println("list2 after  = " + list2.stringList());
    }
```

```
}
```

Output:

```
list1 before = Earth, Mars, Jupiter
list2 before = Io, Phobos
list1 after  = Earth, Mars, Jupiter, Io, Phobos
list2 after  =
```

6.4.1 SList

An `SList` is a singly linked list, and thus uses less storage than a `DList`. In addition, insertion and removal of elements are slightly faster, due to the fact that there are fewer references to keep track of. The downside is that an `SList` may only be traversed in the forward direction, but in many cases this is all that is needed anyway. An `SList` implements the `FrontStackable` and `BackQueueable` interfaces described in chapter 4.

The `SList` has the following additional features:

- `splice:` Destructively moves one or more elements from one list into this list.
- `merge:` Merges the elements of another list into this list. Both lists are assumed to be sorted.
- `pushFront:` Adds an element to the head of the `SList`.
- `popFront:` Removes the head of the `SList`.

6.4.2 DList

A `DList` is a doubly linked list. Insertion and removal of elements is slightly slower than for an `SList`, and needs more memory, but the elements may be traversed in both directions. It has the same set of operations as an `SList`, plus a few more operations made possible by the fact that it has bidirectional traversial. A `DList` implements the `FrontQueueable`, `BackQueueable`, `FrontStackable` and `BackStackable` interfaces described in chapter 4.

The `SList` has the following additional features:

- `splice:` Destructively moves one or more elements from one list into this list.
- `merge:` Merges the elements of another list into this list. Both lists are assumed to be sorted.
- `pushFront:` Adds an element to the head of the `DList`.
- `popFront:` Removes the head of the `DList`.
- `popBack:` Removes the tail of the `DList`.
- `reverseBegin:` Returns a reverse iterator treating the `DList` as if it had been reversed.
- `reverseEnd:` Returns a reverse iterator treating the `DList` as if it had been reversed.

`partition:` Partitions the `DList` so that every element matching a certain criteria is moved to the front of the list.

`stablePartition:` Partitions the `DList` so that every element matching a certain criteria is moved to the front of the list.

`nextPermutation:` Generates the next lexicographical permutation.

`prevPermutation:` Generates the previous lexicographical permutation.

6.4.3 IndexedTree

`IndexedTree` is a type sequence with indexed access just like the `IndexedSequence`. Unlike the `IndexedSequence` however, it cannot deliver constant time random access. Instead, it gives you O(log n) time for all insert, erase and indexed access operations. This makes the `IndexedTree` interesting for applications where elements have to be inserted at random positions in the sequence.

`set:` Sets the element at a specific index.

`get:` Gets the element at a specific index.

`indexOf:` Returns the index of the first element that matches the specified criteria.

`popBack:` Removes the last element.

`capacity:` Returns the current capacity.

`trimToSize:` Optimizes storage use with respect to the current number of elements.

`ensureCapacity:` Ensures that no reallocation of memory is needed to accomodate the specified number of elements.

`reverseBegin:` Returns a reverse iterator treating the `IndexedTree` as if it had been reversed.

`reverseEnd:` Returns a reverse iterator treating the IndexedTree as if it had been reversed.

`randomShuffle:` Shuffles the elements around in a random order.

`stableSort:` Sorts the element with respect to a certain criteria with time-complexity $\boldsymbol{O}(n \log n)$.

`partition:` Partitions the IndexedTree moving to the front of the `IndexedTree` every element matching a certain criteria.

`stablePartition:` Partitions the `IndexedTree` so that every element matching a certain criteria is moved to the front of the `IndexedTree`.

`nextPermutation:` Generates the next lexicographical permutation.

`prevPermutation:` Generates the previous lexicographical permutation.

`pushHeap:` Assumes the `IndexedTree` is a valid heap, and pushes an element into it.

`popHeap:` Assumes the `IndexedTree` is a valid heap, and returns the greatest value from it.

`makeHeap:` Makes a heap out of the `IndexedTree`.

`sortHeap:` Sorts the elements in the heap.

CHAPTER 7

Indexed Sequences

`IndexedSequence` is the abstract class that provides the common denominator for all random access sequence containers, such as `Array`, `Deque` and `BoundedArray`.

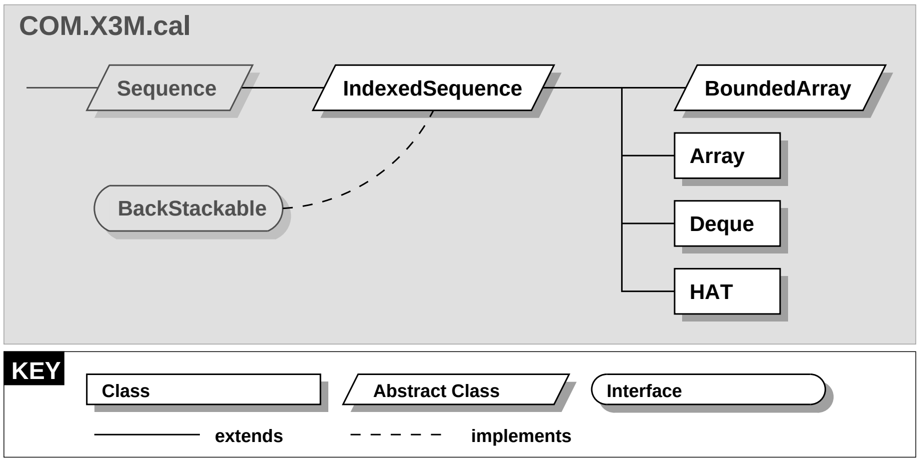

Figure 7.1: Indexed Sequence Hierarchy

`IndexedSequence` expands the `Sequence` container with the concept of random access to all the elements in constant time. However, insertion of elements at an arbitrary position may only be done in linear time. `IndexedSequence` provide versions using integer indices for most of the operations in `Sequence`. Versions of these operations that work on a subrange defined by two indices are also provided.

The following is features that are introduced by `IndexedSequence`.

`set:`	Sets the element at a specific index.
`get:`	Gets the element at a specific index.
`indexOf:`	Returns the index of the first element that matches the specified criteria.

`popBack`: Removes the last element.

`capacity`: Returns the current capacity.

`trimToSize`: Optimizes storage use with respect to the current number of elements.

`ensureCapacity`: Ensures that no reallocation of memory is needed to accomodate the specified number of elements.

`reverseBegin`: Returns a reverse iterator treating the `IndexedSequence` as if it had been reversed.

`reverseEnd`: Returns a reverse iterator treating the IndexedSequence as if it had been reversed.

`randomShuffle`: Shuffles the elements around in a random order.

`stableSort`: Sorts the element with respect to a certain criteria with time-complexity $O(n \log n)$

`partition`: Partitions the IndexedSequence moving to the front of the `IndexedSequence` every element matching a certain criteria.

`stablePartition`: Partitions the `IndexedSequence` so that every element matching a certain criteria is moved to the front of the `IndexedSequence`.

`nextPermutation`: Generates the next lexicographical permutation.

`prevPermutation`: Generates the previous lexicographical permutation.

`pushHeap`: Assumes the `IndexedSequence` is a valid heap, and pushes an element into it.

`popHeap`: Assumes the `IndexedSequence` is a valid heap, and returns the greatest value from it.

`makeHeap`: Makes a heap out of the `IndexedSequence`.

`sortHeap`: Sorts the elements in the heap.

7.1 Random Access

As mentioned, an `IndexedSequence` allows random access to all its elements in constant time. The most common way of accessing the elements is by using an index. For this, the `set` and `get` methods should be used.

Example: `IndexedSequence1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;

public class IndexedSequence1
{
    public static void main(String[] args)
    {
        IndexedSequence sequence = new Array();
```

```
            sequence.add("Sun");
            sequence.add("Mercury");
            sequence.add("Venus");
            sequence.add("Jupiter");
            sequence.add("Pluto");

            System.out.println("contents = " + sequence);
            System.out.println("Third planet from the sun is " +
                               sequence.get(3) + " ... NOT!");
            sequence.set(3, "Earth");
            System.out.println("Third planet from the sun is " +
                               sequence.get(3));
        }
    }
```

Output:

```
contents = COM.X3M.cal.Array[ Sun, Mercury, Venus, Jupiter, Pluto ]
Third planet from the sun is Jupiter ... NOT!
Third planet from the sun is Earth
```

Most of the methods from `Sequence` have been overloaded to take parameters that are indices inplace of iterators. In addition, some of the methods now operate on a subrange of the elemenets indicated by two indices.

Example: `IndexedSequence2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;

public class IndexedSequence2
{
    public static void main(String[] args)
    {
        IndexedSequence sequence = new Deque();

        sequence.add("Sun");
        sequence.add("Mercury");
        sequence.add("Venus");
        sequence.add("Jupiter");
        sequence.add("Pluto");

        System.out.println("before = " + sequence.stringList());
        sequence.transform(1, sequence.size(), new ToLowerCase());
        System.out.println("after  = " + sequence.stringList());
    }
}
```

Output:

```
before = Sun, Mercury, Venus, Jupiter, Pluto
after  = Sun, mercury, venus, jupiter, pluto
```

7.2 Managing Storage

Because `IndexedSequences` all use some sort of array to keep track of its elements, they sometimes need to reallocate memory[1] to provide storage for more elements. This is done automatically, and will work without any help from the programmer. However, at times it might be useful to give some hints as to how much storage is actually needed. For this, two methods have been provided, `trimToSize` and `ensureCapacity`. The first one optimizes the amount of space allocated for the current number of elements stored in the `IndexedSequence`. The second method tells the `IndexedSequence` that it should make sure there is enough space to hold the specified number of elements. In addition, the method capacity has been provided. The method `capacity` returns the number of elements that the `IndexedSequence` currently may hold without a reallocation.

Example: `IndexedSequence3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.adaptor.*;

public class IndexedSequence3
{
    public static void main(String[] args)
    {
        Array sequence = new Array();

        sequence.add("Sun");
        sequence.add("Mercury");
        sequence.add("Earth");

        System.out.println("before       = " +
                           sequence.stringList() +
                           ", size = " + sequence.size() +
                           ", capacity = " + sequence.capacity());
        sequence.trimToSize();
        System.out.println("after trim   = " +
                           sequence.stringList() +
                           ", size = " + sequence.size() +
                           ", capacity = " + sequence.capacity());
        sequence.insert(2, "Venus");
        System.out.println("after insert = " +
                           sequence.stringList() +
                           ", size = " + sequence.size() +
                           ", capacity = " + sequence.capacity());
    }
}
```

Output:

```
before       = Sun, Mercury, Earth, size = 3, capacity = 4
after trim   = Sun, Mercury, Earth, size = 3, capacity = 3
```

[1] the exception is `BoundedArray` which have a fixed capacity.

```
after insert = Sun, Mercury, Venus, Earth, size = 4, capacity = 6
```

7.3 Array

This type of `IndexedSequence` stores its elements in a plain array. It is the fastest of the different indexed sequences when it comes to accessing the elements, but it suffers from the fact that it can only add and remove elements at the end in constant time.

7.4 Deque

Like an `Array`, this type of `IndexedSequence` stores its elements in a plain array. Unlike an `Array` though, it can insert and remove elements both at the end and at the beginning in constant time. This results in slightly slower access to the elements though.

Additional features of `Deque`:

- `pushFront`: Adds an element to the head.
- `popFront`: Removes elements from the head.

7.5 HAT

This is an adaption of the hashed array tree found in [Sit96]. The problem with arrays is that they suffer from excessive copying when reallocating memory. Also they need an excessive amount of extra memory when reallocating. These problems have been addressed by this implementation.

7.6 BoundedArray

`BoundedArray` is an `IndexedSequence` that works exactly like the `Array`, except that it does not reallocate the underlying array if it runs out of space, but throws an exception instead. Thus the maximum size of the `Array` is bounded, and it can use the same array for storage all the time. This allows it to be a wrapper around native arrays in a predictable way. If it were growable, the array used for storage would be reallocated, and any changes to the elements would not be reflected in the original array. See Chapter 15.

Example: `IndexedSequence4.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.adaptor.*;
```

```
public class IndexedSequence4
{
    public static void main(String[] args)
    {
        int iarray[] = { 1, -4, 8, 0, 2 };
        IndexedSequence sequence = new IntArray(iarray);

        System.out.println("before = " + sequence.stringList());
        sequence.sort();
        System.out.println("after  = " + sequence.stringList());
        System.out.print("native = ");
        for (int i = 0; i < iarray.length - 1; i++)
            System.out.print(iarray[i] + ", ");
        System.out.println(iarray[iarray.length - 1]);
    }
}
```

Output:

```
before = 1, -4, 8, 0, 2
after  = -4, 0, 1, 2, 8
native = -4, 0, 1, 2, 8
```

CHAPTER 8

Generic Tree

The `GenericTree` is a container that stores its elements in a tree structure. There is no automatic ordering of the elements, the structure of the tree is entirely up to the builder of the tree. Also, there is no limit on the number of direct children in any node.

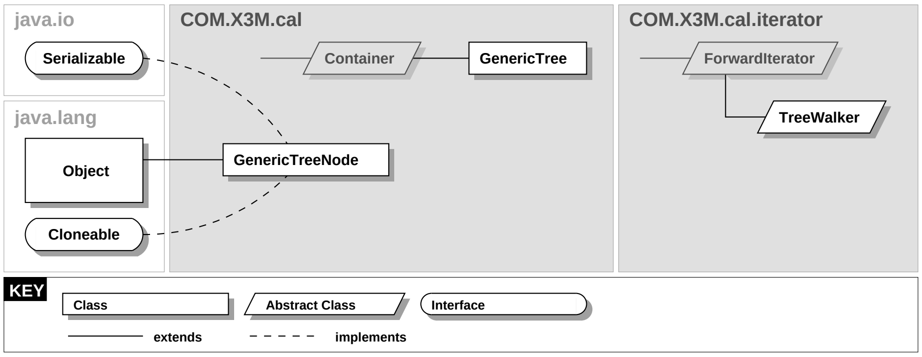

Figure 8.1: Generic Tree Hierarchy

The `GenericTree` have the following operations:

- `insert:` Inserts an element as the child of the specified node.
- `insertRight:` Inserts an element as the right sibling of the specified node.
- `insertLeft:` Inserts an element as the left sibling of the specified node.
- `erase:` Erases a node and it's subtree.
- `getWalker:` Returns a `TreeWalker` positioned at the root of the tree.
- `sort:` Sorts the direct children of the specified node.
- `sortRecursive:` Sorts the direct children of the specified node, and recursively does the same for all the children of this node.
- `printTree:` Prints the structure of the tree to an `OutputStream`.

8.1 Building a Generic Tree

The tree is built using the methods `insert`, `insertLeft` and `insertRight`. To walk around the tree while building it, use the `TreeWalker`. The following example shows how easy it is to build a tree using the insert method along with the iterators.

Example: `GenericTree1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;

public class GenericTree1
{
    public static void main(String[] args)
    {
        GenericTree tree = new GenericTree();

        tree.setRoot("Sun");

        TreeWalker i = tree.getWalker();
        ForwardIterator j;
        tree.insert(i, "Mercury");
        tree.insert(i, "Venus");
        j = tree.insert(i, "Earth");
        tree.insert(j, "Moon");
        j = tree.insert(i, "Mars");
        tree.insert(j, "Phobos");
        tree.insert(j, "Deimos");

        tree.printTree();
    }
}
```

Output:

```
Sun
    Mercury
    Venus
    Earth
        Moon
    Mars
        Phobos
        Deimos
```

8.2 Erasing Elements

When a spesific element is erased from the tree, all the children of that element will be erased as well, as shown in the next example.

Example: `GenericTree2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
```

```
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;

public class GenericTree2
{
    public static void main(String[] args)
    {
        GenericTree tree = new GenericTree();

        tree.setRoot("Sun");

        TreeWalker i = tree.getWalker();
        ForwardIterator j;
        tree.insert(i, "Mercury");
        tree.insert(i, "Venus");
        j = tree.insert(i, "Earth");
        tree.insert(j, "Moon");
        j = tree.insert(i, "Mars");
        tree.insert(j, "Phobos");
        tree.insert(j, "Deimos");

        System.out.println();
        ((TreeWalker) j).goLeftSibling();
        tree.erase(j);

        tree.printTree();
    }
}
```

Output:

```
Sun
    Mercury
    Venus
    Mars
        Phobos
        Deimos
```

8.3 Sorting a Generic Tree

Sometimes one may want to sort the elements in a `GenericTree` so that the children of a node are sorted with respect to each other. This is done as shown in the following example.

Example: `GenericTree3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.function.*;
```

```
public class GenericTree3
{
    public static void main(String[] args)
    {
        GenericTree tree = new GenericTree();

        tree.setRoot("Sun");

        TreeWalker i = tree.getWalker();
        ForwardIterator j;
        tree.insert(i, "Mercury");
        tree.insert(i, "Venus");
        j = tree.insert(i, "Earth");
        tree.insert(j, "Moon");
        j = tree.insert(i, "Mars");
        tree.insert(j, "Phobos");
        tree.insert(j, "Deimos");

        tree.sortRecursive(new LessString());

        tree.printTree();
    }
}
```

Output:

```
Sun
    Earth
        Moon
    Mars
        Deimos
        Phobos
    Mercury
    Venus
```

CHAPTER 9

Associative Containers

An associative container is a container that allows fast retrieval of its elements based on keys. Unlike sequences which organizes its elements in a linear order and guarantees that the relative ordering of the elements are the same before and after insertion and removal of elements, associative containers make no such guarantee. This is because the internal organization of the elements may have to be reorganized after such operations to still allow fast retrieval of the elements.

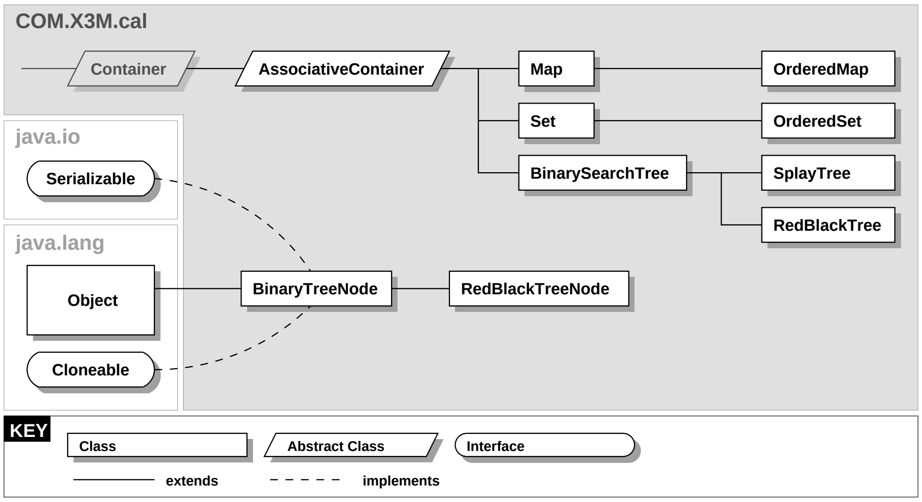

Figure 9.1: Associative Container Hierarchy

Associative containers include such data structures as hash tables and search-trees, in addition to maps and sets, which are just wrappers around either hash tables or search trees. Concrete classes derived from this abstract class are `BinarySearchTree`, `RedBlackTree`, `SplayTree`, `Map`, `OrderedMap`, `Set` and `OrderedSet`.

All associative containers have the following features in addition to the ones for `Container`.

`insert:` Inserts an element.

`erase:`	Erases an element.
`remove:`	Removes all matching elements.
`get:`	Gets the first element with matching key.
`lowerBound:`	Returns an iterator positioned at the first element with key not less than the supplied key.
`upperBound:`	Returns an iterator positioned at the first element with key equal or greater to the supplied key.
`equalRange:`	Equal to `new Pair(lowerBound, upperBound)`.

9.1 Storing Elements

To store an element in an associative container, one may use any of the insert methods, or the add method mentioned in the Chapter 5. For all associative containers, the add and insert methods will find an appropriate place for the new element. It is not necessary for the programmer to worry about this. The following example shows this.

Example: `AssociativeContainer1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;

public class AssociativeContainer1
{
    public static void main(String[] args)
    {
        AssociativeContainer assoc =
            new OrderedSet(new LessString());

        assoc.add("Sun");
        assoc.add("Mars");
        assoc.add("Venus");

        System.out.println("before = " + assoc.stringList());
        assoc.insert("Jupiter");
        System.out.println("after  = " + assoc.stringList());
    }
}
```

Output:

```
before = Mars, Sun, Venus
after  = Jupiter, Mars, Sun, Venus
```

9.2 Removing Elements

As for sequences there is more than one way to remove elements. One may either use the remove operation that removes all elements that match a specific key, or one may specific elements by using erase with an iterator.

Example: `AssociativeContainer2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;

public class AssociativeContainer2
{
    public static void main(String[] args)
    {
        AssociativeContainer assoc = new OrderedSet();

        assoc.add("Sun");
        assoc.add("Mars");
        assoc.add("Venus");

        System.out.println("before       = " + assoc.stringList());
        assoc.remove("Mars");
        System.out.println("after remove = " + assoc.stringList());
        assoc.erase(assoc.find("Venus"));
        System.out.println("after find   = " + assoc.stringList());
    }
}
```

Output:

```
before       = Sun, Mars, Venus
after remove = Sun, Venus
after find   = Sun
```

9.3 Search Trees

Search trees organizes the elements in a tree structure. This results in operations that generally work in $O(\log n)$ time. This running time assumes that the tree is roughly balanced, but unless steps are taken to ensure that the tree stays balanced, it may quickly become unbalanced. As a result, several different implementations of search trees have been included, each with certain benefits and drawbacks.

A property of a search tree is that the elements are kept in sorted order. The ordering is dictated by a `BinaryPredicate`, by default it uses the `Less` predicate, resulting in the elements being ordered in ascending order based on their `hashCode`. This ordering is guaranteed, however, if there are groups of "equal" elements, the internal order of these elements may change over time.

Note that iterators for search trees are constant, meaning that it is not possible to use the `set` method. This is because the internal organization of the elements might be invalidated if one changes the value of an element.

9.3.1 BinarySearchTree

This is the basic `BinarySearchTree`. Each node has up to two children, and insertion is done in $O(h)$ time where h is the height of the tree. For a balanced tree, the height will be $\log n$ where n is the number of elements. Nothing is done to ensure that the tree stays balanced when inserting and removing elements though, so the user of the tree will have to worry about that.

It defines only one additional feature:

`printTree`: Prints the tree to an output stream.

The following example uses the `BinarySearchTree`.

Example: `AssociativeContainer3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;

public class AssociativeContainer3
{
    public static void main(String[] args)
    {
        BinarySearchTree tree =
            new BinarySearchTree(new LessString());

        tree.add("Mars");
        tree.add("Venus");
        tree.add("Sun");
        tree.add("Neptune");
        tree.add("Pluto");
        tree.add("Uranus");

        tree.printTree();
    }
}
```

Output:

```
         O 5:Venus +
         |         |
         |         |       O 1:Uranus
         |         |       |
         |         O 4:Sun +
         |                 |
         |                 |           O 1:Pluto
         |                 |           |
         |                 O 2:Neptune +
         |
O 6:Mars +
```

9.3.2 RedBlackTree

Red-black trees improve on the basic binary search tree by imposing a balancing condition upon the organization of the elements. Exactly how this is done is out of scope for this document, but if interested, have a look at any good book on algorithms and data structures [THCR94]. The result however, is that the tree stays roughly balanced at all times, and all operations on red-black trees have a guaranteed worst case running time of $O(\log n)$.

Red-black trees uses color as its balancing condition. Each node may be either red or black, hence the name.

Example: `AssociativeContainer4.java`

```
...
BinarySearchTree tree = new RedBlackTree(new LessString());
...
```

Output:

```
          O B-2:Venus +
          |           |
          |           O R-1:Uranus
          |
O B-6:Sun +
          |
          |             O R-1:Pluto
          |             |
          O B-3:Neptune +
                        |
                        O R-1:Mars
```

9.3.3 SplayTree

Like red-black trees, splay trees try to maintain a roughly balanced tree, but without using any balancing conditions [ST85]. Instead they are self-adjusting, performing splay-operations within the tree every time an access is made. The amortized cost (see Appendix B) of each operation on an n-node tree is $O(\log n)$, but individual calls to an operation may take longer time.

Example: `AssociativeContainer5.java`

```
...
BinarySearchTree tree = new SplayTree(new LessString());
...
tree.printTree();
tree.find("Pluto");
tree.printTree();
...
```

Output:

```
           O 1:Venus
           |
O 6:Uranus +
           |
           |         O 1:Sun
           |         |
           O 4:Pluto +
```

```
                    |
                    O 2:Neptune +
                                |
                                O 1:Mars

                    O 1:Venus
                    |
          O 3:Uranus +
          |         |
          |         O 1:Sun
          |
O 6:Pluto +
          |
          O 2:Neptune +
                     |
                     O 1:Mars
```

9.4 HashTables

Currently, there are no public hash tables in CAL, however there is a private one used by Set and Map. It may be made public later if there is a need, or if variants of this `HashTable` is introduced. The current implementation is based on the proposal for hash tables for STL [BM95]

9.5 Set and OrderedSet

The basic `Set` is just a wrapper class around the `HashTable` just described. It may or may not allow duplicate elements, by default, it does not.

`OrderedSet` orders the elements in any order using a predicate to compare elements. Default this predicate compares the `hashCode`.

The following example shows the use of sets.

Example: `AssociativeContainer6.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.adaptor.*;
import COM.X3M.cal.function.*;

public class AssociativeContainer6
{
    public static void main(String[] args)
    {
        OrderedSet set1   = new OrderedSet();
```

```
        OrderedSet set2   = new OrderedSet();
        SList      result = new SList();

        set1.add("Earth");
        set1.add("Mars");
        set1.add("Uranus");
        set1.add("Venus");

        set2.add("Venus");
        set2.add("Mars");

        System.out.println("set1        = " + set1.stringList());
        System.out.println("set2        = " + set2.stringList());

        SetOperations.difference(set1.begin(), set1.end(),
                                 set2.begin(), set2.end(),
                                 new BackInsertIterator(result),
                                 new LessString());

        System.out.println("set1 - set2 = " + result.stringList());
    }
}
```

Output:

```
set1        = Mars, Earth, Venus, Uranus
set2        = Mars, Venus
set1 - set2 = Earth, Uranus
```

9.6 Map and OrderedMap

Maps work just like sets, except they associate a key to a value, and the key and the value are two separate objects. This is done by internally wrapping up the key and value in an `Association`.

New features introduced:

- `keysBegin`: Returns an iterator positioned at the first element. This iterator iterates through the keys instead of the values.
- `keysEnd`: Returns an iterator positioned at the last element. This iterator iterates through the keys instead of the values.

The **begin** and **end** methods for maps returns an iterator that iterates through the values, not the keys.

`OrderedMap` orders the values in any order using a predicate to compare elements. Default this predicate compares the `hashCode`.

The following example shows a simple use of maps.

Example: `AssociativeContainer7.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
```

```
public class AssociativeContainer7
{
    public static void main(String[] args)
    {
        Map map = new OrderedMap();

        map.add("Earth", "Moon");
        map.add("Mars", "Deimos");
        map.add("Uranus", "Oberon");

        System.out.print("keys    = ");
        for (ForwardIterator i = map.keysBegin(); !i.atEnd(); )
        {
            System.out.print(i.nextElement());
            if (!i.atEnd())
                System.out.print(", ");
            else
                System.out.println();
        }

        System.out.println("values  = " + map.stringList());
        System.out.println("Sattelite orbiting planet Earth: " +
                           map.get("Earth"));
    }
}
```

Output:

```
keys    = Mars, Earth, Uranus
values  = Deimos, Moon, Oberon
Sattelite orbiting planet Earth: Moon
```

CHAPTER 10

Iterators

Iterators provide the means to traverse the elements in a data structure in a common, uniform and consistent way. CAL iterators extend the concept of the `Enumeration` found in the Java™ JDK with a few simple, but useful operations, detailed later in this chapter. To differentiate between the possibilities inherent in some data structures, CAL provide five different kinds of iterators, shown in the diagram below.

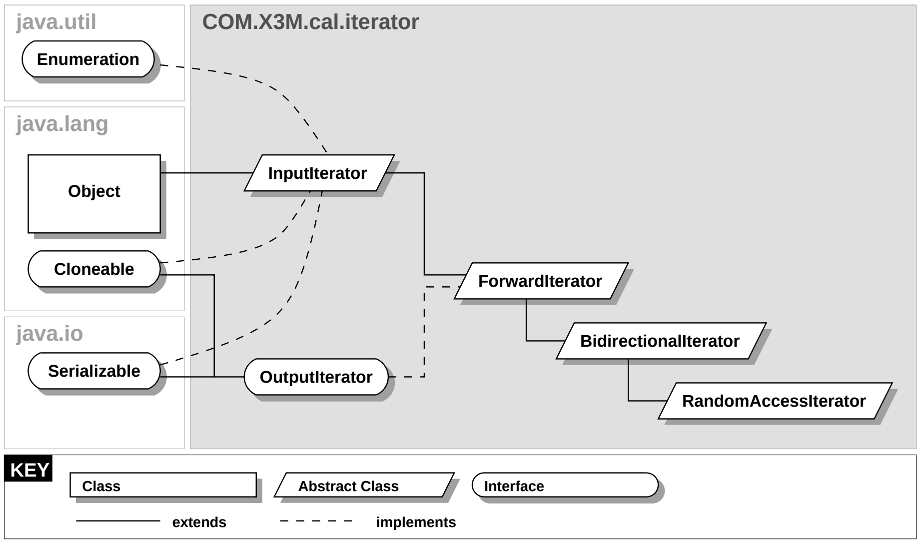

Figure 10.1: Iterator Hierarchy

10.1 InputIterator

An `InputIterator` is an iterator that may read one element at a time in the forward direction. Note that algorithms based on this iterator should be single pass, because for `InputIterator` `r` and `s`,

```
r.equals(s)
```

does not imply:

```
r.inc();
s.inc();
r.equals(s);
```

The following operations are defined for an `InputIterator`:

`atEnd`:	return true if the iterator is positioned after the last element.
`inc`:	increment the position of the iterator by one (or n).
`get`:	return the element at the current position.
`equals`:	compare two iterators and return true if they are equal.
`clone`:	return a clone of this iterator.

The `InputIterator` provides two ways of iterating through all the elements. First off, there's the Java™ JDK `Enumeration` style of doing it, using the two methods `hasMoreElements()` and `nextElement()`. Secondly, one may use the methods `inc()`, `get()` and either `hasMoreElements()` or `atEnd()`. The first is the faster, as there are only two method calls rather than three, but the second way provide more possibilities.

The following example iterates through the elements in three different ways.

Example: `Iterator1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import java.util.Enumeration;

public class Iterator1
{
    public static void main(String[] args)
    {
        Container container = new Array();

        container.add("Sun");
        container.add("Mercury");
        container.add("Venus");
        container.add("Mars");

        for (Enumeration e = container.elements();
             e.hasMoreElements();)
            System.out.print(e.nextElement() + ", ");
        System.out.println();
```

```
        for (InputIterator i = container.begin();
             !i.atEnd(); i.inc())
            System.out.print(i.get() + ", ");
        System.out.println();
        for (InputIterator i = container.begin();
             !i.equals(container.end()); i.inc())
            System.out.print(i.get() + ", ");
        System.out.println();
    }
}
```

Output:

```
Sun, Mercury, Venus, Mars,
Sun, Mercury, Venus, Mars,
Sun, Mercury, Venus, Mars,
```

10.2 OutputIterator

An `OuputIterator` is an iterator that may write one element at a time in the forward direction. As for `InputIterators`, algorithms based on this iterator should be single pass. Furthermore, values should be assigned before incrementing the iterator. These are the operations defined for the `OutputIterator`:

`inc:`	increment the position of the iterator by one (or n).
`set:`	set the element at the current position.
`clone:`	return a clone of this iterator.

The following example uses an `OutputStreamIterator` to display the elements.

Example: `Iterator2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.adaptor.*;
import java.util.Enumeration;

public class Iterator2
{
    public static void main(String[] args)
    {
        OutputStreamIterator out = new OutputStreamIterator();

        out.set("Sun");
        out.set(", Mercury");
        out.set(", Venus\n");
    }
}
```

Output:

```
Sun
```

```
, Mercury
, Venus
```

10.3 ForwardIterator

A `ForwardIterator` is an iterator combining the characteristics of `InputIterator` and `OutputIterator`. The restrictions regarding single-pass and assignments are removed, and multi-pass unidirectional algorithms may be applied to the iterator.

These are the operations defined for the `ForwardIterator`:

`belongsTo:`	return true belongs to the specified Object.
`distance:`	return the distance between two iterators.
`get:`	return the element positioned n places ahead.
`set:`	set the element positioned n places ahead.
`swap:`	swap the elements in two iterators.

The following example replaces "Venus" with "Earth":

Example: `Iterator3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import java.util.Enumeration;

public class Iterator3
{
    public static void main(String[] args)
    {
        Container container = new Array();

        container.add("Sun");
        container.add("Mercury");
        container.add("Venus");
        container.add("Mars");

        System.out.println("before = " + container.stringList());
        ForwardIterator i = container.begin();
        while (!i.atEnd())
        {
            if (i.get().equals("Venus"))
                i.set("Earth");
            i.inc();
        }
        System.out.println("after  = " + container.stringList());
    }
}
```

Output:
```
before = Sun, Mercury, Venus, Mars
after  = Sun, Mercury, Earth, Mars
```

10.4 BidirectionalIterator

The `BidirectionalIterator` extend the `ForwardIterator` by allowing algorithms to be bidirectional, ie. moving the iterator both forwards and backwards. These are the operations defined for the `BidirectionalIterator`:

`atBegin`:	return true if the iterator is positioned at the first element.
`dec`:	decrement the position of the iterator by one (or n).

The following example iterates through the elements in the reverse direction.

Example: `Iterator4.java`
```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import java.util.Enumeration;

public class Iterator4
{
    public static void main(String[] args)
    {
        Container container = new Array();

        container.add("Sun");
        container.add("Mercury");
        container.add("Venus");
        container.add("Mars");

        System.out.println("container = " +
                           container.stringList());
        BidirectionalIterator i =
            (BidirectionalIterator) container.end();
        System.out.print("Iterated: ");
        do
        {
            i.dec();
            System.out.print(i.get() + ", ");
        } while (!i.atBegin());
        System.out.println();
    }
}
```

Output:
```
container = Sun, Mercury, Venus, Mars
Iterated: Mars, Venus, Mercury, Sun,
```

10.5 RandomAccessIterator

A `RandomAccessIterator` has all the properties of `BidirectionalIterator`, in addition to allowing random access to all the elements. These are the operations defined for the `RandomAccessIterator`:

- `index:` return the index of the current position.
- `swap:` swap the elements at arbitrary positions.
- `before:` return true if this iterator is positioned before the other iterator.
- `after:` return true if this iterator is positioned after the other iterator.

The following example swaps two elements using the `RandomAccessIterator`.

Example: `Iterator5.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import java.util.Enumeration;

public class Iterator5
{
    public static void main(String[] args)
    {
        Container container = new Array();

        container.add("Sun");
        container.add("Mercury");
        container.add("Venus");
        container.add("Mars");

        System.out.println("before = " + container.stringList());
        ((RandomAccessIterator) container.begin()).swap(1, 3);
        System.out.println("after  = " + container.stringList());
    }
}
```

Output:

```
before = Sun, Mercury, Venus, Mars
after  = Sun, Mars, Venus, Mercury
```

CHAPTER 11

Generic Algorithms

If you want to do anything with your containers besides storing and retrieving, you need to use an *algorithm*. They include *non-mutating* algorithms, like finding an element or counting the number of elements matching a criteria, and *mutating* algorithms, like copying and reversing element order.

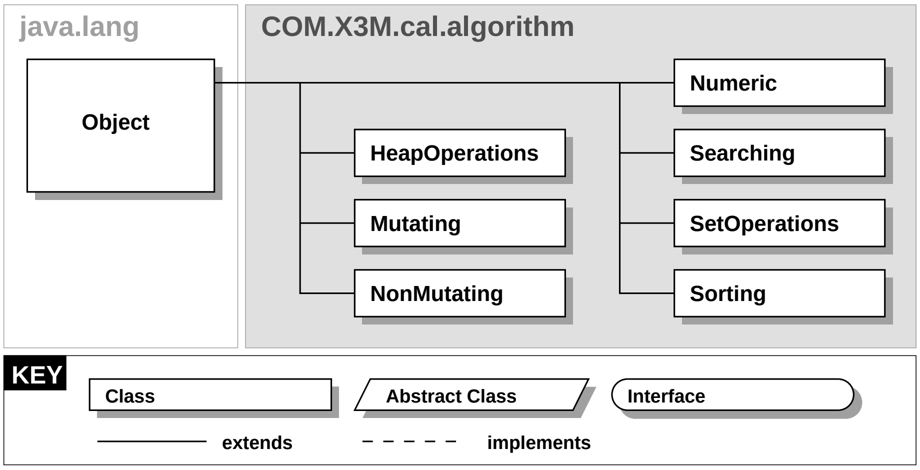

Figure 11.1: Algorithm hierarchy

The algorithms are implemented as static methods in wrapper classes. There are seven of these wrapper classes, each grouping a specific type of algorithms. These types are the general `Mutating` and `NonMutating` classes, as well as the more specific `Searching`, `Sorting`, `Numeric`, `SetOperation` and `HeapOperation` classes. There are also helper methods available at the container classes.

The following example demonstrate a simple algorithm, `find` that finds a specific element. This is a non-mutating operation taking two input-iterators, the beginning and the end of the search area, and an object matching the wanted object.

Note that for many of the algorithms, there are corresponding helper functions in the container classes that operate on all of the container.

Example: `Algorithms1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.algorithm.*;

public class Algorithms1
{
    public static void main(String[] args)
    {
        String[] jupiterMoons = {"Metis", "Adrastea", "Amalthea"};
        Container container = new Array(jupiterMoons);

        String find1 = (String)container.find("Adrastea").get();
        String find2 = (String)NonMutating.find(container.begin(),
                                                container.end(),
                                                "Amalthea").get();

        System.out.println("find1 = " + find1 +
                           ", find2 = " + find2);
    }
}
```

Output:

```
find1 = Adrastea, find2 = Amalthea
```

11.1 Nonmutating

The non-mutating algorithms are algorithms that do not change the iterators they operate upon. All methods accept a subrange specified by a first and last iterator. The subrange does not include the last element. The algorithms of the class `NonMutating` are:

- `forEach`: Traverses the container and applies the supplied function to every item.
- `find`: Searches for an item that matches the supplied value.
- `findIf`: Searches for an item that satisfies the supplied predicate.
- `count`: Returns the number of items matching the supplied value.
- `countIf`: Returns the number of items satisfying the supplied predicate.
- `mismatch`: Finds the first mismatch between the container subrange and the supplied corresponding range.
- `equal`: Returns true if the items in the subrange equals a supplied corresponding range.
- `adjacentFind`: Returns the first element that equals its successor.
- `adjacentFindIf`: Returns the first element that equals its successor and satisfies the supplied predictor.

`search:` Find a subrange with items equal to another supplied subrange.

The following example uses the `forEach` algorithm to print the elements in an array:

Example: `AlgorithmsNonmutating1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.function.*;

public class AlgorithmsNonmutating1
{
    public static void main(String[] args)
    {
        String[] saturnMoons = {"Pan", "Atlas", "Prometheus"};
        Container container = new Array(saturnMoons);

        NonMutating.forEach(container.begin(), container.end(),
                            new PrintString());
    }
}

class PrintString implements UnaryFunction
{
    public Object function(Object arg)
    {
        System.out.println(arg.toString());
        return null;
    }
}
```

Output:

```
Pan
Atlas
Prometheus
```

11.2 Mutating

The mutating algorithms change the iterators they operate on, or output the result to a different iterator.

`copy:` Copies all the elements (shallow copy) to the supplied output iterator.

`copyBackward:` Copies all the elements to the supplied output iterator. The iterators are decremented. Used if the region copied to are in the region copied from.

`swapRanges:` Swaps two ranges.

`transformCopy`: Transforms and copies all the elements to the supplied output iterator.

`replace`: Replaces all elements that matches an old value with a new value.

`replaceIf`: Replaces all elements satisfying a supplied predicate with a new value.

`replaceCopy`: Same as copy, but replaces all elements matching a supplied value with a new value.

`replaceCopyIf`: Same as copy, but replaces all elements satisfying a supplied predicate with a new value.

`fill`: Assigns a value to every item in the range.

`fillN`: Assigns a value to the next N items.

`generate`: Assigns the values generated by the supplied generator to every item in the range.

`generateN`: Assigns the values generated by the supplied generator to the next N items.

`nextPermutation`: Generates the next lexicographical permutation if one exists, and returns true. Otherwise it generates the smallest permutation, and returns false.

`prevPermutation`: Generates the previous lexicographical permutation if one exists, and returns true. Otherwise it generates the largest permutation, and returns false.

`remove`: Removes the items matching the supplied value.

`removeIf`: Removes the items satisfying the supplied predicate.

`removeCopy`: Same as copy, but removes all elements matching a supplied value.

`removeCopyIf`: Same as copy, but removes all elements matching a supplied predicate.

`unique`: Removes all but the first item in every consecutive range of equal items.

`uniqueCopy`: Same as `copy`, but copies only the first element from each consecutive group of equal elements.

`reverse`: Reverses the order of the items in the subrange.

`reverseCopy`: Copies the subrange to an output iterator in the reverse order.

`rotate`: Rotates the element in the subrange.

`rotateCopy`: Copies the subrange rotated to an output iterator.

`partition`: Partitions the subrange into two subranges, where the lower satisfies a supplied predicate, and the higher does not.

`stablePartition`: Same as `partition`, but the relative ordering of the items is preserved.

The following example shows some of the Mutating-algorithms in use. Note that the call to `unique` does not truncate the list but returns an iterator positioned after the last element in the resulting range.

Example: `AlgorithmsMutating1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.adaptor.*;

public class AlgorithmsMutating1
{
    public static void main(String[] args)
    {
        String[] saturnMoons = {"Helene", "Rhea", "Titan",
                                "Hyperion", "Iapetus"};
        Sequence container = new Array(saturnMoons);
        SList result = new SList();

        Mutating.copy(container.begin(), container.end(),
                      new FrontInsertIterator(result));

        System.out.println("result = " + result.stringList());

        result.clear();
        Mutating.transformCopy(container.begin(), container.end(),
                               new FrontInsertIterator(result),
                               new ToLowerCase());
        System.out.println("result = " + result.stringList());
        Mutating.replace(result.begin(), result.end(),
                         "rhea", "titan");
        System.out.println("result = " + result.stringList());
        ForwardIterator last = Mutating.unique(result.begin(),
                                               result.end());
        System.out.println("result = " + result.stringList());
        result.erase(last, result.end());
        System.out.println("result = " + result.stringList());

        System.out.println("container = " +
                           container.stringList());
        Mutating.nthElement(
            (RandomAccessIterator)container.begin(),
            (RandomAccessIterator)container.findIndex(3),
            (RandomAccessIterator)container.end(),
            new LessString());
        System.out.println("container = " +
                           container.stringList());
    }
}
```

Output:

```
result = Iapetus, Hyperion, Titan, Rhea, Helene
result = iapetus, hyperion, titan, rhea, helene
result = iapetus, hyperion, titan, titan, helene
result = iapetus, hyperion, titan, helene, helene
result = iapetus, hyperion, titan, helene
container = Helene, Rhea, Titan, Hyperion, Iapetus
container = Helene, Hyperion, Iapetus, Rhea, Titan
```

11.3 Searching

All the searching methods except maxElement and minElement require the supplied ranges to be sorted. The following algorithms are defined in the `Searching` class.

- `lexicographicalCompare:` Returns true if the first input range is lexicographical less than the other input range.
- `binarySearch:` Returns true if the value is contained in the supplied range.
- `lowerBound:` Returns the first position where the supplied value can be inserted without violating the ordering.
- `upperBound:` Returns the furthermost position where the supplied value can be inserted without violating the ordering.
- `equalRange:` Finds the greatest subrange such that for any iterator i in the subrange, value can be inserted without violating the ordering.
- `maxElement:` Returns an iterator pointing to the maximum value.
- `minElement:` Returns an iterator pointing to the minimum value.

11.4 Sorting

These are the algorithms defined in the `Sorting` class.

- `randomShuffle:` Shuffles the elements randomly.
- `sort:` Sorts the elements.
- `stableSort:` Sorts the elements preserving the relative order of equal elements.
- `partialSort:` Sorts the first elements of a subrange. The rest of the elements will be put in an undefined order.
- `nthElement:` Swaps the `nth` item and the item that would be in that position if the range was sorted.

An example showing randomShuffle and sort in action:

Example: `AlgorithmsSorting1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
```

```
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.iterator.*;

public class AlgorithmsSorting1
{
    public static void main(String[] args)
    {
        String[] neptuneMoons = {"Naiad", "Despina", "Proteus",
                                 "Triton", "Nereid"};
        Container container = new Array(neptuneMoons);

        System.out.println("container = " +
                           container.stringList());
        Sorting.randomShuffle(
            (RandomAccessIterator)container.begin(),
            (RandomAccessIterator)container.end());
        System.out.println("container = " +
                           container.stringList());
        Sorting.sort((RandomAccessIterator)container.begin(),
                     (RandomAccessIterator)container.end(),
                     new LessString());
        System.out.println("container = " +
                           container.stringList());
    }
}
```

Output:

```
container = Naiad, Despina, Proteus, Triton, Nereid
container = Naiad, Despina, Nereid, Triton, Proteus
container = Despina, Naiad, Nereid, Proteus, Triton
```

11.5 Numeric

The numeric algorithms are shown below:

`accumulate`: Returns the accumulated value after applying the supplied function to every item in the subrange.

`innerProduct`: Returns the inner product of the two supplied ranges, using the supplied accumulation function and combining function.

`partialSum`: Assigns all partial sums, defined by the supplied function, at the output iterator.

`adjacentDifference`: Assigns all differences, defined by the supplied function, to the output iterator.

An example of `partialSum` and `adjacentDifference`:

Example: `AlgorithmsNumeric1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.adaptor.*;

public class AlgorithmsNumeric1
{
    public static void main(String[] args)
    {
        int[] numbers = {778, 108, 58, 150, 228};
        Container container = new IntArray(numbers);
        BackStackable result1 = new DList();
        BackStackable result2 = new Deque();

        Numeric.partialSum(container.begin(), container.end(),
                           new BackInsertIterator(result1));
        Numeric.adjacentDifference(container.begin(),
                                   container.end(),
                           new BackInsertIterator(result2));

        System.out.println("result1 = " + result1);
        System.out.println("result2 = " + result2);
    }
}
```

Output:

```
result1 = COM.X3M.cal.DList[ 778, 886, 944, 1094, 1322 ]
result2 = COM.X3M.cal.Deque[ 778, -670, -50, 92, 78 ]
```

11.6 Set Operations

All the set operations require the supplied ranges to be sorted. These are the algorithms defined in `SetOperations`:

- `merge`: Merges two sorted iterators.
- `inplaceMerge`: Merges two consecutive ranges putting the result into the concatenated range. The merge is stable, that is, it conserves ordering of matching items.
- `includes`: Returns true if every element in the first range is included in the other.
- `union`: Returns the union of two sorted ranges, that is, all elements with multiplicity one.
- `intersection`: Returns the intersection of two ranges, that is, common elements.
- `difference`: Returns the difference of two sorted ranges.
- `symmetricDifference`: Returns the symmetric difference of two ranges.

11.7 Heap Operations

The heap operations are algorithms that operate on a range of RandomAccessIterators. The operations for retrieving the next element ans inserting an element into a heap is an $O(\lg n)$ operation. The following are the operations defined for the `Heap` class

- `pushHeap:` The subrange, not including the last item, is assumed to be a heap. This method will add the last item to that heap.
- `popHeap:` Swaps the first item on the heap with the last, and makes a valid heap out of the new subrange not including the last item.
- `makeHeap:` Constructs a heap out of the supplied range.
- `sortHeap:` Sorts the heap in a non-stable way.

CHAPTER 12

Function and Predicate Objects

To further enhance the power of the algorithms provided, CAL use function objects to alter their behavior. There are two basic types of function objects, the ones that return an `Object` and those that return a `boolean` value. They are further refined into taking either one or two arguments. Thus there are 4 basic types of function/predicate objects, `UnaryFunction`, `BinaryFunction`, `UnaryPredicate` and `BinaryPredicate`.

12.1 Function Objects

There are two versions of the function object, `UnaryFunction` and `BinaryFunction`. Both are interfaces, and they both define a method `function`.

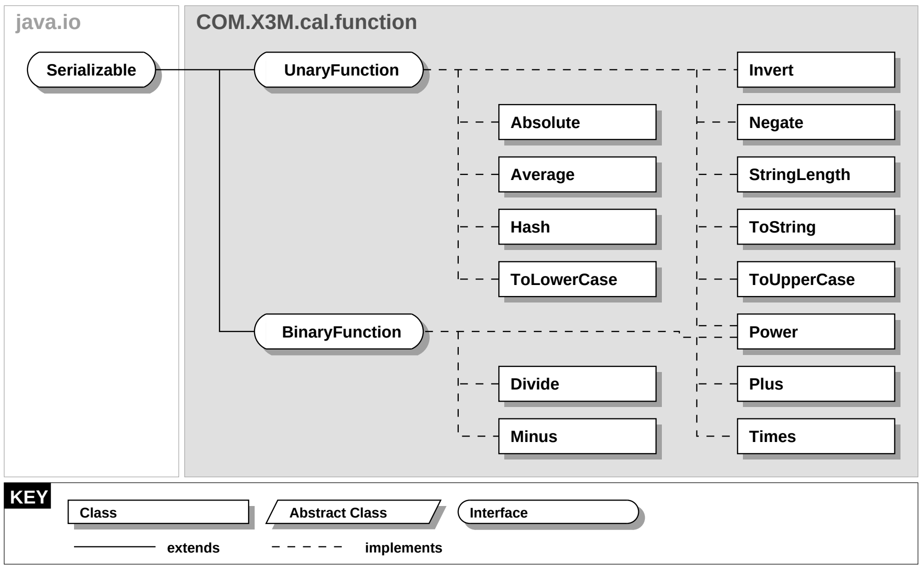

Figure 12.1: Function Object Hierarchy

Table 12.1: Summary of UnaryFunction Objects

Name	*Return value*	*Restriction on argument*
`Average`	`x`	`Number`
`Absolute`	`Math.absolute(x)`	`Number`
`Hash`	`x.hashCode()`	`Object`
`Invert`	`1/x`	`Number, x != 0`
`Negate`	`-x`	`Number`
`Power`	`Math.power(x, x)`	`Number`
`StringLength`	`x.stringLength()`	`String`
`ToString`	`x.toString()`	`String`
`ToLowerCase`	`x.toLowerCase()`	`String`
`ToUpperCase`	`x.toUpperCase()`	`String`

12.1.1 UnaryFunction Objects

`UnaryFunction` objects has a method that takes one argument, an `Object`, and returns an `Object`.

A short summary of the available `UnaryFunction` objects is given in the Table 12.1.

The following example converts all strings to upper case.

Example: `Function1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;

public class Function1
{
    public static void main(String[] args)
    {
        Array array = new Array();

        array.add("Sun");
        array.add("Mercury");
        array.add("Venus");
        array.add("Mars");

        System.out.println("before = " + array.stringList());
        array.transform(new ToUpperCase());
        System.out.println("after  = " + array.stringList());
    }
}
```

Output:

```
before = Sun, Mercury, Venus, Mars
after  = SUN, MERCURY, VENUS, MARS
```

Table 12.2: Summary of BinaryFunction Objects

Name	*Return value*	*Restriction on arguments*
`Divide`	`x/y`	`Number, y != 0`
`Minus`	`x - y`	`Number`
`Plus`	`x + y`	`Number`
`Power`	`x ^ y`	`Number`
`Times`	`x * y`	`Number`

12.1.2 BinaryFunction Objects

`BinaryFunction` objects has a method that takes two arguments, both `Object`, and like `UnaryFunction`, returns an `Object`.

Table 12.2 shows the `BinaryFunction` objects available.

The following example calculates the a special version of the inner product for a sequence. The inner product defined here is:

$$(x_0 + y_0) \cdot (x_1 + y_1) \cdot \ldots$$

Example: `Function2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.adaptor.*;

public class Function2
{
    public static void main(String[] args)
    {
        int a[] = {2, 1, 1};
        int b[] = {3, 4, -2};
        IntArray array1 = new IntArray(a);
        IntArray array2 = new IntArray(b);
        Object result = new Integer(1);

        System.out.println("array1 = " + array1.stringList());
        System.out.println("array2 = " + array2.stringList());

        result = Numeric.innerProduct(array1.begin(), array1.end(),
                                      array2.begin(), result,
                                      new Times(), new Plus());

        System.out.println("result = " + result);
    }
}
```

Output:

```
array1 = 2, 1, 1
array2 = 3, 4, -2
result = -25
```

12.2 Predicate Objects

As for the function objects, there are two versions of the predicate. They are the `UnaryPredicate` and the `BinaryPredicate`. Both are interfaces, and they both define a method `predicate`.

Both define a method predicate, that return a boolean value. UnaryPredicate define this predicate method to take one Object argument, while BinaryPredicate define it to take two Object arguments.

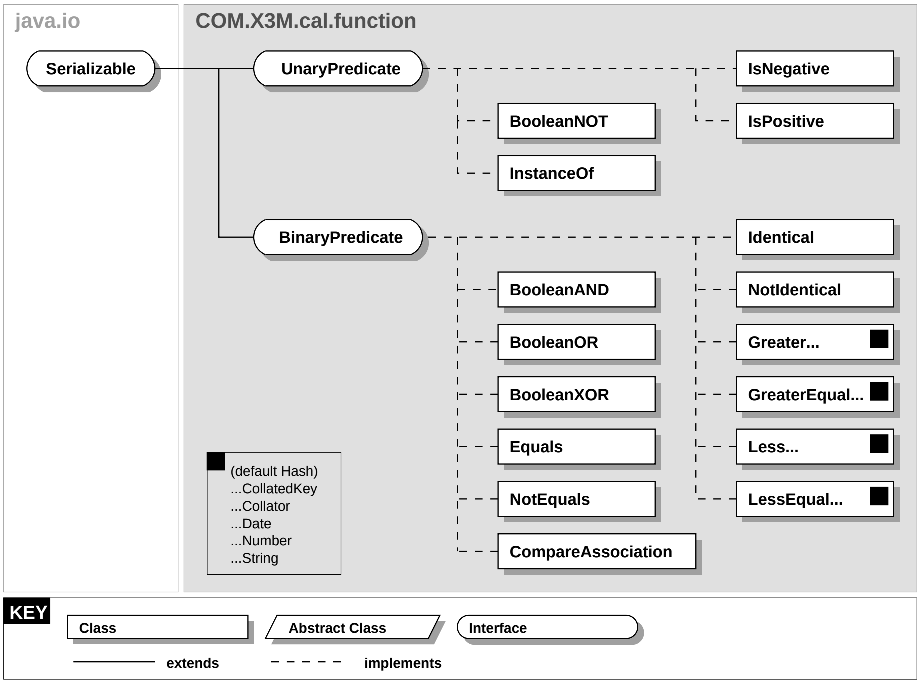

Figure 12.2: Predicate Object Hierarchy

12.2.1 UnaryPredicate Objects

`UnaryPredicate` objects has a method that takes one argument, an `Object`, and returns a `boolean` value.

Table 12.3 shows the `UnaryPredicate` objects available.

The following example removes all elements that are negative.

Table 12.3: Summary of UnaryPredicate Objects

Name	*Return value*	*Restriction on argument*
`BooleanNot`	`!x`	`Boolean`
`InstanceOf`	`x instanceof C`	`Object`
`IsNegative`	`x < 0`	`Number`
`IsPositive`	`x >= 0`	`Number`

Example: `Function3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.adaptor.*;

public class Function3
{
    public static void main(String[] args)
    {
        int a[] = {2, 1, 1, -2, -34, 4, -1};
        IntArray array = new IntArray(a);

        System.out.println("before = " + array.stringList());
        array.removeIf(new IsNegative());
        System.out.println("after  = " + array.stringList());
    }
}
```

Output:

```
before = 2, 1, 1, -2, -34, 4, -1
after  = 2, 1, 1, 4
```

12.2.2 BinaryPredicate Objects

`BinaryPredicate` objects has a method that takes two arguments, both `Object`, and like `UnaryPredicate`, returns a `boolean` value.

Table 12.4 shows the `BinaryPredicate` objects available. Note that the `Less`, `LessEqual`, `Greater` and `GreaterEqual` can be used with 6 different types. For example if you want to compare strings, you should use one of the `LessString`, `LessEqualString`, `GreaterString` or `GreaterEqualString` objects. These types are listed in Table 12.5. The default compares the hash values.

The following example sorts a collection of strings in the reverse direction.

Example: `Function4.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;
```

Table 12.4: Summary of BinaryPredicate Objects

Name	*Return value*	*Restriction on argument*
`BooleanAND`	`x && y`	`Boolean`
`BooleanOR`	`x || y`	`Boolean`
`BooleanXOR`	`(x && !y) || (!x && y)`	`Boolean`
`Equals`	`x.equals(y)`	`Object`
`NotEquals`	`!x.equals(y)`	`Object`
`Identical`	`x == y`	`Object`
`NotIdentical`	`x != y`	`Object`
`Less …`	`x < y`	See Table 12.5
`LessEqual …`	`x <= y`	See Table 12.5
`Greater …`	`x > y`	See Table 12.5
`GreaterEqual …`	`x >= y`	See Table 12.5

Table 12.5: Summary of Compare Predicates Types

Type	*Restriction on argument*
…	`java.lang.Object`
… `CollatedKey`	`java.text.CollatedKey`
… `Collator`	`java.text.Collator`
… `Date`	`java.util.Date`
… `Number`	`java.lang.Number`
… `String`	`java.lang.String`

```
import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.algorithm.*;

public class Function4
{
    public static void main(String[] args)
    {
        String str[] = {"Sun", "Venus", "Earth",
                        "Mars", "Uranus", "Pluto"};
        Array array = new Array(str);

        System.out.println("before = " + array.stringList());
        array.sort(new GreaterString());
        System.out.println("after  = " + array.stringList());
    }
}
```

Output:

```
before = Sun, Venus, Earth, Mars, Uranus, Pluto
after  = Venus, Uranus, Sun, Pluto, Mars, Earth
```

12.3 Defining New Function Objects

It is fairly easy to define new function objects that may be used throughout the library. The first thing to decide upon is for what kind of algorithm it should be used. This will affect whether it's going to be a function or a predicate, and whether it will be a unary or binary version. Note that it is possible to have one class be both a function and a predicate, and both unary and binary at the same time. One should be weary about doing this, as it might make code less readable.

Once the type of function object needed is determined, define a new class that implements the corresponding interface. As an example, we define a new `BinaryFunction` that returns the concatenation of two strings.

Example: `Function5.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.algorithm.*;

public class Function5
{
    public static void main(String[] args)
    {
        Array array1 = new Array();

        array1.add("Earth");
```

```
        array1.add("Mars");

        Array array2 = new Array();

        array2.add(" - Moon");
        array2.add(" - Deimos and Phobos");

        System.out.println("array1:\n" + array1.stringList("\n") +
                           "\narray2:\n" +array2.stringList("\n"));
        Mutating.transformCopy(array1.begin(), array1.end(),
                               array2.begin(), array1.begin(),
                               new Concat());

        System.out.println("\narray1:\n" +array1.stringList("\n"));
    }
}

final class Concat implements BinaryFunction
{
    public Object function(Object arg1, Object arg2)
    {
        try
        {
            return (String) arg1 + (String) arg2;
        }
        catch (ClassCastException e)
        {
            throw new IllegalArgumentException("Arguments " +
                                               "must be Strings.");
        }
    }
}
```

Output:

```
array1:
Earth
Mars
array2:
 - Moon
 - Deimos and Phobos

array1:
Earth - Moon
Mars - Deimos and Phobos
```

12.4 Defining New Predicate Objects

This is the same procedure described in the section above. The following example constructs a unary predicate that returns true if the object is of a specific instance. It uses this predicate to remove all elements except instances of `Double`.

Example: `Function6.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
```

```
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.adaptor.*;

public class Function6
{
    public static void main(String[] args)
    {
        Array array = new Array();

        array.add(new Integer(4));
        array.add(new Long(2L));
        array.add(new Float(11.1));
        array.add(new Double(-3.1));
        array.add(new Long(1000L));

        Array array2 = new Array();

        System.out.println("before = " + array.stringList());
        UnaryPredicate pred = new IsInstanceOf("java.lang.Long");
        array.removeIf(new UnaryNot(pred));
        System.out.println("after  = " + array.stringList());
    }
}

final class IsInstanceOf implements UnaryPredicate
{
    String name;

    public IsInstanceOf(String name)
    {
        this.name = name;
    }

    public boolean predicate(Object arg)
    {
        return (arg.getClass().getName().equals(name));
    }
}
```

Output:

```
before = 4, 2, 11.1, -3.1, 1000
after  = 2, 1000
```

CHAPTER 13

Generators

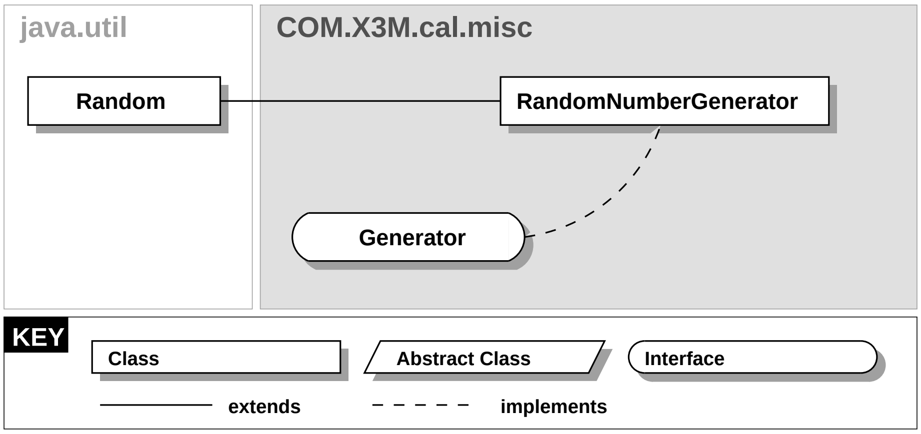

Figure 13.1: Generator hierarchy

A `Generator` is a type of function object that takes no parameters, and returns a value every time it is called. They are fairly simple, but still quite useful in certain algorithms. It only defines one method, the generate method:

- `generate`: generates and returns a new object.

13.1 RandomNumberGenerator

The `RandomNumberGenrerator` generates a random number every time it is called. In addition to the generate method defined by the `Generator` interface, if defines a number of other useful methods shown below.

`RandomNumberGenerator` is derived from the `java.util.Random` class.

- `nextInt`: returns the next random integer.
- `nextLong`: returns the next random long.
- `nextFloat`: returns the next random float.
- `nextDouble`: returns the next random double.

The following example shows how to generate random numbers in the range [0,100 >

Example: `Generator1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.adaptor.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.misc.*;

public class Generator1
{
    public static void main(String[] args)
    {
        Array array = new Array();

        BackInsertIterator i = new BackInsertIterator(array);
        Mutating.generateN(i, 5, new RandomNumberGenerator(100));

        System.out.println("array = " + array.stringList());
    }
}
```

Output:

```
array = 44, 59, 47, 53, 40
```

13.2 Defining New Generators

To define a new generator, all that is needed is to implement a public method named `generate`, taking no parameters and returning an `Object`, and have the class implement the `Generator` interface. As an example, we define a new generator that generates the Fibonacci sequence, which is defined as follows:

$$
\begin{aligned}
F_1 &= 1 \\
F_2 &= 1 \\
F_{n+1} &= F_n + F_{n-1}
\end{aligned}
$$

Beware, this sequence grows fast once it gets started! At 47 elements, the primitive type int will overflow.

Example: `Generator2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.adaptor.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.misc.*;
```

```java
public class Generator2
{
    public static void main(String[] args)
    {
        Array array = new Array();

        BackInsertIterator i = new BackInsertIterator(array);
        Mutating.generateN(i, 10, new FibonacciGenerator());

        System.out.println("array = " + array.stringList());
    }
}

final class FibonacciGenerator implements Generator
{
    private int nextVal = 1;
    private int prevVal = 0;

    public Object generate()
    {
        int tmp = prevVal + nextVal;
        prevVal = nextVal;
        nextVal = tmp;
        return new Integer(prevVal);
    }
}
```

Output:

```
array = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

CHAPTER 14

Pattern Matchers

In many cases, when searching through data, you don't know exactly what you are looking for. Maybe you know the structure of the input, and want to extract parts of it, or you want to verify that the structure of some text follows some criteria. In these cases, where structure is more important than contents, pattern matchers are ideal.

CAL comes with two different pattern matchers, the simple *wildcard* matcher and the more advanced *regular expression* matcher. To integrate these macthers in the rest of the library, we have included a set of predicates and functions to use with generic algorithms.

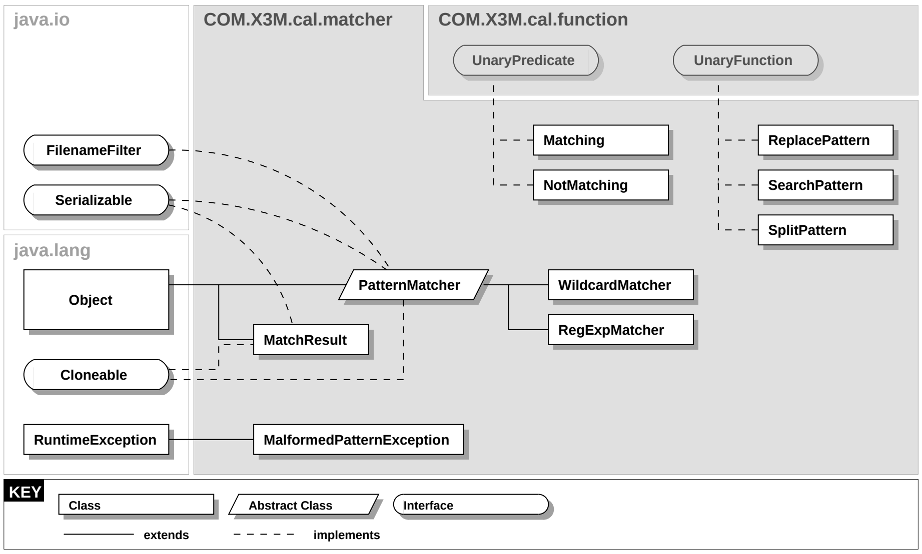

Figure 14.1: Pattern Matcher Hierarchy

Table 14.1: Summary of Wildcards

Wildcard	*Matches*
`?`	any character
`*`	any number of any characters
`[a-b]`	any character between, and including, `a` and `b`
`[^a-b]`	any character not in the range `a` to `b`
other characters	the character

14.1 Wildcard Matcher

The wildcard matcher is a very simple pattern matcher. Most have used text based user interfaces like DOS or UNIX and written things like

```
%> ls *.txt
```

in a UNIX environment or

```
C:\> dir *.txt
```

in DOS. Both these command lines will list all files in the current directory that ends in "`.txt`". The "`*`" will match anything that happens to be in front. A wildcard pattern must match the whole string to be a true match.

The type of wildcard matching that is implemented in CAL is the UNIX type of wildcards, which is more powerful that the one found in DOS.

The syntax of wildcard matching is summarized in table 14.1.

To summarize the wildcard matching section, table 14.2 shows some examples on patterns, test strings, and whether they match or not.

The following example shows how to build a simple directory lister (`ls`/`dir`). It takes two parameters, the first is the directory path, and the second is the pattern.

Example: `PatternMatcher1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import java.io.File;
import COM.X3M.cal.matcher.*;

public class PatternMatcher1
{
    public static void main(String[] args)
    {
        if (args.length > 1)
        {
            File file = new File(args[0]);
```

Table 14.2: Wildcard Examples

Pattern	*Test String*	*Match?*	*Explanation*
`*`	`anyfile.anyextension`	true	`*` matches any character, and any number of them
`???file.*`	`anyfile.anyextension`	true	`???` matches any substring that is three characters
`[a-d]*`	`astring`	true	the test string starts with a character in the range `a` to `d`
`[^a-d]*`	`astring`	false	the test string starts with an illegal character
`*stuff*`	`some_stuff_here`	true	there is "`stuff`" somewhere in the test string
`name`	`name`	true	the test string is exactly "`name`"

```
            PatternMatcher matcher = new WildcardMatcher(args[1]);

            String[] files = file.list(matcher);

            System.out.println("Directory '" + args[0] +
                               "' pattern '" + args[1] + "'");
            for (int i = 0; i < files.length; i++)
                System.out.println("  " + files[i]);
        }
        else
            System.out.println("Must have 2 arguments.");
    }
}
```

Output:

```
Directory 'examples' pattern 'A*[M-N]*.java'
  AlgorithmsNumeric1.java
  AlgorithmsNonmutating1.java
  AlgorithmsMutating1.java
```

14.2 Regular Expression Matcher

The regular exception matcher allows much more complex patterns, including back references, controlled quantifications of elements, and other features. In opposition to the wildcard matcher, regular expressions are not required to match the whole string unless this is explicitly stated.

The regular expression flavor used in CAL is very similar to the one used in the perl scripting language [WCS96].

As regular expressions allows a more complex pattern syntax, there is some

Table 14.3: List of Zero Width Atoms

Atom	*Description*
`^`	Matches only at the beginning of the string.
`$`	Matches only at the end of the string.
`\b`	Matches at word boundary, i.e. at the start of, or at the end of a word.
`\B`	Matches anywhere but at word boundaries.
`\A`	Matches at the beginning of a line, i.e. at the start of the string, or after newline.
`\Z`	Matches at the end of a line, i.e. at the end of the string, or before newline.
`(?= … )`	Matches if … matches, but does not have a width.
`(?! … )`	Matches if … does not match, but does not have a width.

characters that have special meanings to the pattern compiler, these are: “\”, “|”, “(”, “)”, “[”, “{”, “^”, “$”, “*”, “+”, “?”, “.”. These characters are called meta-characters. If you want to match any of these characters (and not their special meaning), simply put a backslash character in front of it.The backslash itself is also a meta-character, but a backslash followed by another backslash is the character.

In Java strings, a double backslash is required for every backslash. This is because backslash functions as a escape character.

A regular expression is built of atoms. There are several types of atoms. Some that matches zero width, others that includes sub atoms in their match, and atoms that matches single characters and character sets.

Zero width matching atoms are used to verify certain properties about the position in the test string. Table 14.3 shows all zero width atoms.

Character matching atoms include those who matches single characters and those who matches ranges of characters. Table 14.4 shows the character matching atoms.

Parts of the expression may be enclosed in parentheses, which allows the text it matched to be restored later through back referencing. An example of this is the pattern “`(.)\1`” which matches two equal characters next to each other. The `\1` references back to the first set of parentheses, and the dot within the parentheses matches any character.

One of the real powers of regular expressions are quantifications of atoms. You can specify in the pattern how many times each atoms should match, with both a minimum and a maximum number, and whether the match should be minimal or maximal. Quantifiers are applicable to any atoms except those listed in table 14.3. Table 14.5 shows the possible quantifiers.

The following example demonstrates a general use of the regular expression pattern matcher. It takes two parameters, the first is the targe string, and the second is the regular expression pattern. It outputs all the successful matches.

Table 14.4: List of Character Atoms

Atom	*Description*
`\a`	Matches the ASCII alarm character.
`\n`	Matches newline.
`\r`	Matches carriage return.
`\t`	Matches tab.
`\f`	Matches form feed.
`char`	Matches the character.
`\d`	Matches any digit.
`\D`	Matches anything but digits.
`\w`	Matches any word character.
`\W`	Matches anything but word characters.
`\s`	Matches whitespace.
`\S`	Matches anything but whitespace.
`.`	Matches any character but newline.
`[a-b]`	Matches the range from `a` to `b`.
`[^a-b]`	Matches but the range from `a` to `b`.

Table 14.5: List of Quantifiers

Maximal Match	*Minimal Match*	*Description*
{m,n}	{m,n}?	Matches minimum m and maximum n times.
{n,}	{n,}?	Matches minimum n times.
{n}	{n}	Matches exactly n times.
`*`	`*?`	Matches 0 or more times. Same as `{0,}`.
`+`	`+?`	Matches 1 or more times. Same as `{1,}`.
`?`	`??`	Matches 0 or 1 times. Same as `{0,1}`.

Example: `PatternMatcher2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.matcher.*;

public class PatternMatcher2
{
    public static void main(String[] args)
    {
        if (args.length > 1)
        {
            System.out.println("Target  = '" + args[0] + "'");
            System.out.println("Pattern = '" + args[1] + "'");

            RegExpMatcher  matcher = new RegExpMatcher(args[1]);
            MatchResult result[] = matcher.globalSearch(args[0]);

            System.out.println("Matched = ");
            if ((result != null) && (result.length > 0))
            {
                for (int i = 0; i < result.length; i++)
                    System.out.println("  " + result[i]);
            }
            else
                System.out.println("  No matches!");
        }
        else
            System.out.println("Must have 2 arguments.");
    }
}
```

Output:

```
Target  = 'Rolling'
Pattern = 'l+'
Matched =
  COM.X3M.cal.matcher.MatchResult[ 2, 4, 'll' ]
```

The `RegExpPattern` class has a few useful utility functions that makes tings a lot easier. These functions are described below.

`search`: Returns the first match in form of a `MatchResult` object.

`globalSearch`: Returns all the matches in form of a `MatchResult` array.

`replace`: Replaces the first occurance of a match with a replacement string, and returns the result.

`globalReplace`: Replaces all occurances of matches with a replacement string, and returns the result.

`split`: Splits the target string at the occurances of matches, and returns an array of the splitted strings.

The following example shows how to rearange words in a string using `replace`. The pattern uses back references in the replacement string.

Example: `PatternMatcher3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.matcher.*;

public class PatternMatcher3
{
    public static void main(String[] args)
    {
        String target = "mad dog bites man";
        String pattern = "(\\w*) (\\w*) (\\w*) (\\w*)";

        RegExpMatcher matcher = new RegExpMatcher(pattern);
        String result = matcher.replace(target, "\\1 \\4 \\3 \\2");

        System.out.println("before replace = '" + target + "'");
        System.out.println("after replace  = '" + result + "'");
    }
}
```

Output:

```
before replace = 'mad dog bites man'
after replace  = 'mad man bites dog'
```

14.3 Predicate Objects

There are two predicate objects that can be used with the `PatternMatcher` classes. These are `Matching` and `NotMatching`, and both are unary predicate objects.

The `Matching` object returns true if the pattern matches the argument, and the `NotMatching` object does the opposite. These predicates can be used with all exsisting generic algorithms in the library that uses unary predicates.

The following example uses the `Matching` predicate to remove all planets that has only one moon.

Example: `PatternMatcher4.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.matcher.*;

public class PatternMatcher4
{
    public static void main(String[] args)
    {
        String [] str = {"Pluto - Charon",
                         "Earth - Moon",
                         "Mars - Phobos and Deimos"};
        Array array = new Array(str);
```

```
        System.out.println("before = " + array.stringList());

        PatternMatcher matcher = new RegExpMatcher("^\\w* " +
                                                   "- \\w*$");
        array.removeIf(new Matching(matcher));

        System.out.println("after  = " + array.stringList());
    }
}
```

Output:

```
before = Pluto - Charon, Earth - Moon, Mars - Phobos and Deimos
after  = Mars - Phobos and Deimos
```

14.4 Function Objects

All function objects described here can only be used with the `RegExpMatcher`. There are 3 function object currently implemented. These are `ReplacePattern`, `SearchPattern` and `SplitPattern`, and is designed to be used with mutating algorithms. All the pattern matcher function object operate on the unary function argument, and this argument must be an instance of `String`.

The following list describes the function objects.

- `ReplacePattern` Returns the result of executing `RegExpPattern.replace` on the function argument.
- `SearchPattern` Returns the result of executing `RegExpPattern.search` on the function argument.
- `SplitPattern` Returns the result of executing `RegExpPattern.split` on the function argument. The result is an array of `String` elements.

The following example demonstrates the use of `SplitPattern`. It finds and lists all the moons seperated by "`, `" and "` and `".

Example: `PatternMatcher5.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.matcher.*;
import COM.X3M.cal.iterator.*;

public class PatternMatcher5
{
    public static void main(String[] args)
    {
        String [] str = {"Charon",
                         "Naiad, Thalassa and Despina",
                         "Moon",
                         "Phobos and Deimos"};
```

```
        Array array = new Array(str);

        System.out.println("before = ");
        for (ForwardIterator it = array.begin(); !it.atEnd(); )
        {
            System.out.println("  " + it.nextElement());
        }

        RegExpMatcher matcher = new RegExpMatcher(", | and ");
        array.transform(new SplitPattern(matcher));

        System.out.println("after  = ");
        for (ForwardIterator it = array.begin(); !it.atEnd(); )
        {
            Object tmp[] = (Object[])it.nextElement();
            for (int i = 0; i < tmp.length; i++)
                System.out.println("  " + tmp[i]);
        }
    }
}
```

Output:

```
before =
  Charon
  Naiad, Thalassa and Despina
  Moon
  Phobos and Deimos
after  =
  Charon
  Naiad
  Thalassa
  Despina
  Moon
  Phobos
  Deimos
```

CHAPTER 15

Container Adaptors

Often one may find it useful to provide restricted access to containers. For this, CAL defines a set of adaptors that wrap around containers, hiding some of the details. These are the `Stack`, `Queue` and `PriorityQueue`, which are all well known abstract data types.

In addition, to ease the integration with native arrays and certain data structures defined in the JDK, CAL provide array adaptors for these.

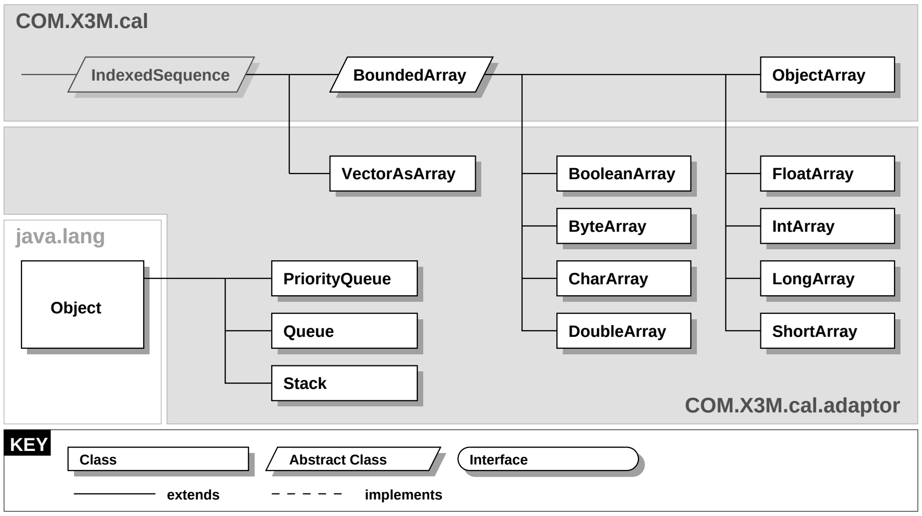

Figure 15.1: Container Adaptor Hierarchy

15.1 Stack

The `Stack` provides restricted access to any class implementing the `BackStackable` or `FrontStackable` interface. A `Stack` is the classic FILO[1] queue, defining the

[1]First In Last Out

following operations:

- `push`: push an element to the top of the Stack.
- `pop`: pop an element from the top of the Stack, and return it to the caller.
- `peek`: return the element at the top of the Stack.
- `size`: return the number of elements in the Stack.
- `isEmpty`: return true if there are no elements in the stack.

Following is an example on how one might use a `Stack`. It is a postfix calculator, with only two operations defined, plus and multiply.

Example: `ContainerAdaptor1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.adaptor.*;
import java.io.*;

public class ContainerAdaptor1
{
    PrintStream out = System.out;

    /**
     * For JDK1.1, use this instead:
     * BufferedReader in =
     *     new BufferedReader(new InputStreamReader(System.in));
     */
    DataInputStream in = new DataInputStream(System.in);

    Stack stack = new Stack();

    public static void main(String[] args) throws IOException
    {
        new ContainerAdaptor1();
    }

    public ContainerAdaptor1() throws IOException
    {
        String command;
        boolean quit = false;
        while (!quit)
        {
            command = readValue();
            switch (command.charAt(0))
            {
                case '+':
                    add();
                    break;
                case '*':
```

```
                multiply();
                break;
            case 'q':
                quit = true;
                break;
            default:
                pushNumber(command);
        }
        System.out.println("Top of stack: " + stack.peek());
    }
}

private final void add()
{
    stack.push(
        new Double(((Double) stack.pop()).doubleValue() +
                   ((Double) stack.pop()).doubleValue()));
}

private final void multiply()
{
    stack.push(
        new Double(((Double) stack.pop()).doubleValue() *
                   ((Double) stack.pop()).doubleValue()));
}

private final void pushNumber(String number)
{
    stack.push(new Double(number));
}

private final String readValue() throws IOException
{
    out.print("Enter value or operand: ");
    return in.readLine();
}
}
```

Output:

```
Enter value or operand: 4
Top of stack: 4.0
Enter value or operand: 3
Top of stack: 3.0
Enter value or operand: 7
Top of stack: 7.0
Enter value or operand: +
Top of stack: 10.0
Enter value or operand: *
Top of stack: 40.0
Enter value or operand: q
Top of stack: 40.0
```

15.2 Queue and PriorityQueue

The `Queue` and the `PriorityQueue` have the same signature. The `Queue` is a classical FIFO[2] queue, while `PriorityQueue` is, as it's name indicates, a priority queue. The `PriorityQueue` uses a `BinaryPredicate` to put an ordering on the elements. It always dequeues the element with the greatest value. Both `Queue` and `PriorityQueue` defines the following operations:

- `enqueue:` insert an element into the queue.
- `dequeue:` remove the next element from the queue.
- `peek:` return the next element to be removed from the queue.
- `size:` return the number of elements in the queue..
- `isEmpty:` return true if there are no elements in the queue.

The following example shows the difference in output between the `Queue` and `PriorityQueue`. Note that the output from the `PriorityQueue` is correct, even though a `LessString` is passed as a parameter to the constructor. This is because the `PriorityQueue` pushes the largest elements to the front, resulting perhaps in the opposite order of what one might expect.

Example: `ContainerAdaptor2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.adaptor.*;

public class ContainerAdaptor2
{
    public static void main(String[] args)
    {
        Queue queue = new Queue();
        PriorityQueue pqueue = new PriorityQueue(new LessString());
        Array a = new Array();

        a.add("Sun");
        a.add("Mercury");
        a.add("Venus");
        a.add("Earth");
        a.add("Mars");

        for (ForwardIterator i = a.elements();
             i.hasMoreElements(); i.inc())
        {
            queue.enqueue(i.get());
            pqueue.enqueue(i.get());
        }
```

[2] First In First Out

Table 15.1: JDK types and their container representative

JDK type	*Container adaptor*
`boolean`	`BooleanArray`
`byte`	`ByteArray`
`char`	`CharArray`
`double`	`DoubleArray`
`float`	`FloatArray`
`int`	`IntArray`
`long`	`LongArray`
`short`	`ShortArray`
`java.lang.Object`	`ObjectArray`
`java.util.Vector`	`VectorAsArray`

```
        System.out.println("input sequence: " + a.stringList());
        System.out.print("output from the Queue:         ");
        while (!queue.isEmpty())
            System.out.print(queue.dequeue() + ", ");
        System.out.print("\noutput from the PriorityQueue: ");
        while (!pqueue.isEmpty())
            System.out.print(pqueue.dequeue() + ", ");
        System.out.println();
    }
}
```

Output:

```
input sequence: Sun, Mercury, Venus, Earth, Mars
output from the Queue:         Sun, Mercury, Venus, Earth, Mars,
output from the PriorityQueue: Venus, Sun, Mercury, Mars, Earth,
```

15.3 Array Adaptors

Unlike the `Stack` and `Queue`, the array adaptors do not provide a restricted interface to other containers. Instead they wrap around native arrays of primitive data types so that they may be seamlessly integrated into the rest of the library. They are all derived from the `BoundedArray`, and this guarantees that the array used for storage isn't reallocated. That way, one can work with both the native array and with the `<type>Array` without worrying whether the `<type>Array` is using the same native array still.

Table 15.1 shows the relation between the Java™ JDK type and the corresponding array adaptor.

With the exception of `ObjectArray`, all the array adaptors define the following operations:

- `set<type>`: set the element at the specified index. Uses the primitive data type `<type>`.

`get<type>`: get the element at the specified index. Uses the primitive data type `<type>`.

The following example shows how any operation on an IntArray is reflected in the original native array.

Example: `ContainerAdaptor3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.adaptor.*;

public class ContainerAdaptor3
{
    public static void main(String[] args)
    {
        int iarray[] = {1, 4, 3, 2};
        IntArray array = new IntArray(iarray);

        System.out.print("original = ");
        for (int i = 0; i < iarray.length; i++)
            System.out.print(iarray[i] + ", ");
        System.out.println("adaptor = " + array.stringList());

        array.sort();

        System.out.print("original = ");
        for (int i = 0; i < iarray.length; i++)
            System.out.print(iarray[i] + ", ");
        System.out.println("adaptor = " + array.stringList());
    }
}
```

Output:

```
original = 1, 4, 3, 2, adaptor = 1, 4, 3, 2
original = 1, 2, 3, 4, adaptor = 1, 2, 3, 4
```

15.4 VectorAsArray

The `VectorAsArray` allows the `java.util.Vector` to be integrated seamlessly into the library. It is derived from `IndexedSequence`, thus inheriting all the methods and features from that class. Note that insertion or removal of several elements in one go is significantly slower than for other `IndexedSequences`. This is due to the fact that the `Vector` provides no means of inserting more than one element at a time.

This example shows how one may integrate the `java.util.Vector` object with CAL:

Example: `ContainerAdaptor4.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;
```

```
import COM.X3M.cal.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.adaptor.*;
import java.util.*;

public class ContainerAdaptor4
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        v.addElement("Sun");
        v.addElement("Earth");

        VectorAsArray varray = new VectorAsArray(v);
        varray.insert(1, "Mercury");
        varray.insert(2, "Venus");

        v.addElement("Mars");

        System.out.println("original = " + v);
        System.out.println("adaptor  = " + varray.stringList());
        varray.sort(new LessString());
        System.out.println("original = " + v);
        System.out.println("adaptor  = " + varray.stringList());
    }
}
```

Output:

```
original = [Sun, Mercury, Venus, Earth, Mars]
adaptor  = Sun, Mercury, Venus, Earth, Mars
original = [Earth, Mars, Mercury, Sun, Venus]
adaptor  = Earth, Mars, Mercury, Sun, Venus
```

CHAPTER 16

Iterator Adaptors

CAL have a couple of iterator adaptors to further enhance the usability of the library. They are the reverse iterators, stream iterators, insert iterators and iterators to interact directly with native arrays.

16.1 Stream Iterators

To make some of the algorithms work with the streams defined in the Java™ JDK CAL has defined a couple of useful stream iterators.

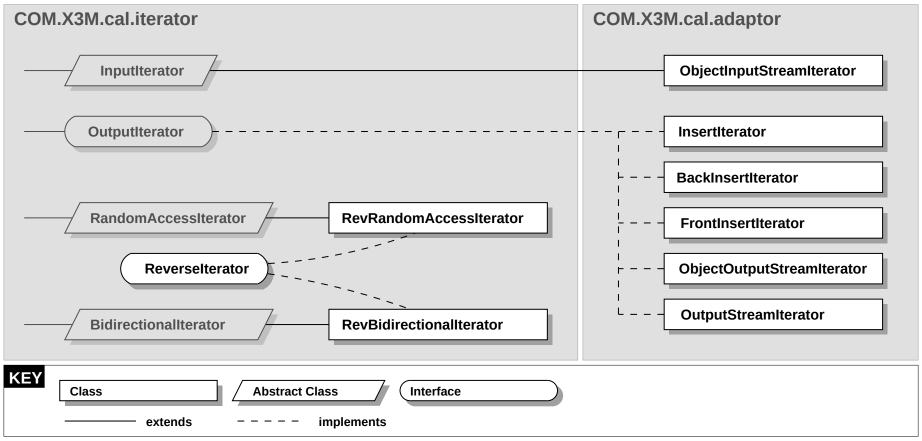

Figure 16.1: Iterator Adaptor Hierarchy

16.1.1 OutputStreamIterator

The `OutputStreamIterator` simply writes the String representation of each Object to an `OuputStream`.

This example shows how the OutputStream may be used to copy the contents of an Array to an `OutputStream`.

Example: `IteratorAdaptor1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.adaptor.*;

public class IteratorAdaptor1
{
    public static void main(String[] args)
    {
        Array array = new Array();

        array.add("Sun");
        array.add("Mars");
        array.add("Saturn");
        array.add("Jupiter");

        Mutating.copy(array.begin(), array.end(),
                      new OutputStreamIterator());
    }
}
```

Output:

```
Sun
Mars
Saturn
Jupiter
```

16.1.2 Object Stream Iterators

Requires JDK1.1

These iterators[1] act on object streams, as defined in the Java™ JDK 1.1. This makes it possible to write the contents of one container to an `ObjectOutputStream`, and then read it back in with a different type of container. The other way of doing it, is to write the container directly to the `ObjectOuputStream`, but then one will get exactly the same type of `Container` back. Usually this is exactly what is wanted, but not always.

The following example shows how one might serialize the contents of an Array, and read it back in to a DList.

Example: `IteratorAdaptor2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.adaptor.*;
import java.io.*;
```

[1] `ObjectInputStreamIterator` is derived from an `InputIterator`, while `ObjectOutputStream` is derived from an `OutputIterator`.

```
public class IteratorAdaptor2
{
    public static void main(String[] args) throws Exception
    {
        Array array = new Array();

        array.add("Sun");
        array.add("Mars");
        array.add("Saturn");
        array.add("Jupiter");

        ObjectOutputStreamIterator out = new
            ObjectOutputStreamIterator(
                new FileOutputStream("IteratorAdaptor2.tmp"));

        System.out.println("array = " + array);
        Mutating.copy(array.begin(), array.end(), out);
        array.clear();
        System.out.println("array = " + array);

        DList dlist = new DList();
        System.out.println("dlist = " + dlist);
        ObjectInputStreamIterator in =
            new ObjectInputStreamIterator(
                new FileInputStream("IteratorAdaptor2.tmp"));
        Mutating.copy(in, in.end(), new BackInsertIterator(dlist));
        System.out.println("dlist = " + dlist);
    }
}
```

Output:

```
array = COM.X3M.cal.Array[ Sun, Mars, Saturn, Jupiter ]
array = COM.X3M.cal.Array[  ]
dlist = COM.X3M.cal.DList[  ]
dlist = COM.X3M.cal.DList[ Sun, Mars, Saturn, Jupiter ]
```

16.2 Insert Iterators

To allow insertion in the same way as writing into a container, three types of insert iterators are provided. They are `FrontInsertIterator`, `BackInsertIterator` and `InsertIterator`, and they are all derived from `OutputIterator`.

Note that the `inc` method need not be called for these iterators, as it will have no effect.

16.2.1 FrontInsertIterator

The `FrontInsertIterator` takes any `PushFront` container as an argument to its constructor, and every time the set method is called, it will insert the argument element to the front of the container with the `pushFront` method.

16.2.2 BackInsertIterator

The `BackInsertIterator` takes any `PushBack` container as an argument to its constructor, and every time the `set` method is called, it will insert the argument element to the back of the container with the `pushBack` method.

16.2.3 InsertIterator

The `InsertIterator` allows insertion into any container at any position. The constructor takes two arguments, the container in which the elements should be inserted, and an iterator, positioned at the element before which new elements should be inserted. This iterator uses the `insert(ForwardIterator position, Object value)` defined for all Containers.

The following example shows how one might want to use these iterators.

Example: `IteratorAdaptor3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.adaptor.*;
import java.io.*;

public class IteratorAdaptor3
{
    public static void main(String[] args)
    {
        Array source = new Array();
        Array array = new Array();
        SList slist = new SList();

        source.add("Sun");
        source.add("Mars");
        source.add("Saturn");
        source.add("Jupiter");

        Mutating.copy(source.begin(), source.end(),
                      new BackInsertIterator(array));
        Mutating.copy(source.begin(), source.end(),
                      new FrontInsertIterator(slist));

        System.out.println("array = " + array.stringList());
        System.out.println("slist = " + slist.stringList());

        OutputIterator out =
            new InsertIterator(array, array.find("Mars"));
        out.set("Mercury");
        out.set("Venus");
        out.set("Earth");

        System.out.println("array = " + array.stringList());
    }
}
```

Output:

```
array = Sun, Mars, Saturn, Jupiter
slist = Jupiter, Saturn, Mars, Sun
array = Sun, Mercury, Venus, Earth, Mars, Saturn, Jupiter
```

16.3 Reverse Iterators

The bidirectional and random-access iterators have corresponding reverse iterators that iterate through the datastructure in the opposite direction. Usually the programmer won't even know he is using one, as the containers that support reverse iterators, will define methods **`reverseBegin`** and **`reverseEnd`** that return the corresponding reverse iterator for that datastructure. However, if there is a need for constructing a new reverse iterator, one simply pass the iterator to the constructor of the proper reverse iterator, and voila, there it is.

Here is an example that transforms the elements to upper case, and at the same time reverses the order of the elements when they are copied to a different container..

Example: `IteratorAdaptor4.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.adaptor.*;
import java.util.*;

public class IteratorAdaptor4
{
    public static void main(String[] args)
    {
        Array array1 = new Array();
        Array array2 = new Array();

        array1.add("Sun");
        array1.add("Mars");
        array1.add("Saturn");
        array1.add("Jupiter");

        System.out.println("array1 = " + array1.stringList());
        Mutating.transformCopy(array1.reverseBegin(),
                               array1.reverseEnd(),
                               new BackInsertIterator(array2),
                               new ToUpperCase());
        System.out.println("array2 = " + array2.stringList());
    }
}
```

Output:

```
array1 = Sun, Mars, Saturn, Jupiter
array2 = JUPITER, SATURN, MARS, SUN
```

Table 16.1: JDK types and their iterator representative

JDK type	*Container adaptor*
`boolean`	`BooleanArrayIterator`
`byte`	`ByteArrayIterator`
`char`	`CharArrayIterator`
`double`	`DoubleArrayIterator`
`float`	`FloatArrayIterator`
`int`	`IntArrayIterator`
`long`	`LongArrayIterator`
`short`	`ShortArrayITerator`
`java.lang.Object`	`ObjectArrayIterator`

16.4 Array Iterators

The `<type>ArrayIterator` adaptors, like the `<type>Array` adaptors[2], help integrate native arrays with the rest of the library. They allow the algorithms to work without having to resort to the `<type>Array` adaptors.

Table 16.1 shows the relation between the Java™ JDK type and the corresponding array adaptor.

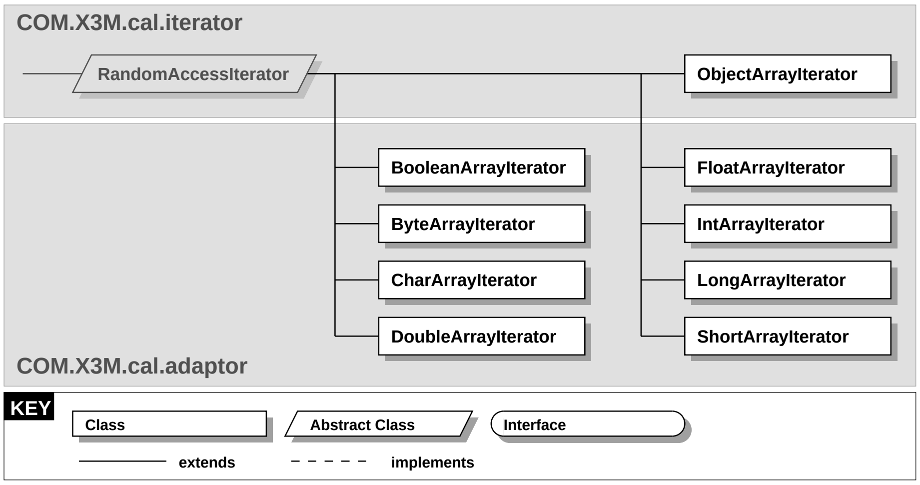

Figure 16.2: Array Iterator Adaptor Hierarchy

This example sorts a native array of `double` using a `DoubleArray`.

Example: `IteratorAdaptor5.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.iterator.*;
```

[2] See Chapter 15

```
import COM.X3M.cal.algorithm.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.adaptor.*;
import java.io.*;

public class IteratorAdaptor5
{
    public static void main(String[] args)
    {
        double array[] = { 42.2, 3.14, 1.8, 24.0, 49.01};

        System.out.print("array = ");
        RandomAccessIterator i = DoubleArrayIterator.begin(array);
        RandomAccessIterator j = DoubleArrayIterator.end(array);
        Mutating.copy(i, j, new OutputStreamIterator(", "));
        Sorting.sort(i, j, new LessNumber());
        System.out.print("\narray = ");
        Mutating.copy(i, j, new OutputStreamIterator(", "));
        System.out.println();
    }
}
```

Output:

```
array = 42.2, 3.14, 1.8, 24.0, 49.01,
array = 1.8, 3.14, 24.0, 42.2, 49.01,
```

CHAPTER 17

Function and Predicate Adaptors

CAL comes with quite a few function and predicate objects. When these do not meet the needs of the programmer, one will have to come up with new ones. However, it isn't always necessary to write entirely new ones. Often all that is needed is to combine the existing functions and predicates in new ways. For that purpose, function and predicate adaptors have been added to the library.

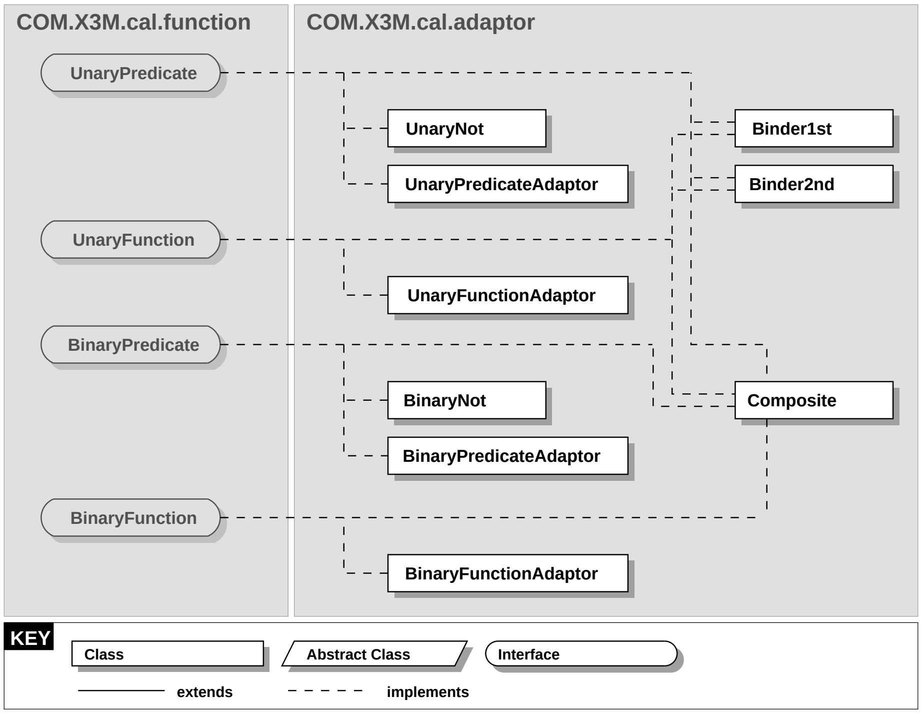

Figure 17.1: Function and Predicate Adaptor Hierarchy

Another kind of adaptor for use in algorithms is function and predicate objects that take a `Method`, as defined in the reflection model in Java™ JDK 1.1, as parameter. They provide the means to use any existing method that have the correct set of arguments and return type.

17.1 Negator

There are two negators available, `UnaryNot` and `BinaryNot`. They provide the means to negate any existing predicate objects. Many of the predicates have already a negator that does not use the negator adaptors, but in special cases, where you have one type of predicate and want the opposite predicate, the negator adaptor is quite useful.

The following examples keeps all numbers that is between 20 and 40.

Example: `FunctionAdaptor1.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.adaptor.*;

public class FunctionAdaptor1
{
    public static void main(String[] args)
    {
        int iarray[] = { 2, 40, 21, 32, 64, 13, 37 };
        IntArray array = new IntArray(iarray);

        System.out.println("before = " + array.stringList());
        array.removeIf(new UnaryNot(new Between(20, 40)));
        System.out.println("after  = " + array.stringList());
    }
}

// Returns true if it is between the specified range.
class Between implements UnaryPredicate
{
    int from, to;

    public Between(int from, int to)
    {
        this.from = from;
        this.to   = to;
    }

    public boolean predicate(Object o)
    {
        return ((((Number)o).intValue() > from) &&
                (((Number)o).intValue() < to));
    }
}
```

Output:

```
before = 2, 40, 21, 32, 64, 13, 37
after  = 21, 32, 37
```

17.2 Binder

There are two binders defined, `Binder1st` and `Binder2nd`. They may be used as either a `UnaryFunction` or a `UnaryPredicate`. If it's to be used as a `UnaryFunction`, the constructor taking a `BinaryFunction` and an `Object` must be called, while the constructor taking a `BinaryPredicate` and an `Object` should be used if using it as a `UnaryFunction`.

The `Binder1st` works by binding the first argument of the binary function or binary predicate to a fixed value. `Binder2nd` works similarly, except it is the second argument that is bound to the fixed value.

The following example removes all numbers less than 34.

Example: `FunctionAdaptor2.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.adaptor.*;

public class FunctionAdaptor2
{
    public static void main(String[] args)
    {
        int iarray[] = { -1, 3, 89, 34, 80 };
        IntArray array = new IntArray(iarray);

        System.out.println("before = " + array.stringList());
        array.removeIf(new Binder2nd(new LessNumber(),
                                     new Integer(34)));
        System.out.println("after  = " + array.stringList());
    }
}
```

Output:

```
before = -1, 3, 89, 34, 80
after  = 89, 34, 80
```

17.3 Composite

The `Composite` may be used in place of any function and predicate object, whether they be unary or binary. The idea is to compose new behaviors out of the existing ones quickly. Since both the `UnaryFunction` and `BinaryFunction` both return an `Object`, their return values may be used as input values for either a function or predicate object, both unary and binary.

The following example removes all strings that is longer than 5 characters.

Example: `FunctionAdaptor3.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;
```

```
import COM.X3M.cal.*;
import COM.X3M.cal.adaptor.*;
import COM.X3M.cal.function.*;

public class FunctionAdaptor3
{
    public static void main(String[] args)
    {
        String str[] = {"Sun", "Mercury", "Venus",
                        "Earth", "Mars", "Saturn", "Jupiter"};
        Array array = new Array(str);

        UnaryPredicate greater5 =
            new Binder2nd(new GreaterNumber(), new Integer(5));
        UnaryPredicate longer5 =
            new Composite(greater5, new StringLength());

        System.out.println("before = " + array.stringList());
        array.removeIf(longer5);
        System.out.println("after  = " + array.stringList());
    }
}
```

Output:

```
before = Sun, Mercury, Venus, Earth, Mars, Saturn, Jupiter
after  = Sun, Venus, Earth, Mars
```

17.4 Method Adaptors

Requires Java™ JDK 1.1 reflection api.

This kind of adaptor makes use of the reflection api introduced in Java™ JDK 1.1. The advantage of this, is that already defined methods may easily be used in the algorithms, as long as they have the right set of arguments and return type. They all provide two constructors. The first argument is the `Object` in which the actual method is defined. The second argument is either `Method` itself, or a `String` with the name of the method.

The following example does the same thing as the composite example, but uses the method adaptor instead.

Example: `FunctionAdaptor4.java`

```
// Copyright (c) 1996, 1997 X3M Solutions
package COM.X3M.cal.examples;

import COM.X3M.cal.*;
import COM.X3M.cal.function.*;
import COM.X3M.cal.adaptor.*;

public class FunctionAdaptor4
{
    public FunctionAdaptor4() throws Exception
    {
        String str[] = {"Sun", "Mercury", "Venus",
                        "Earth", "Mars", "Saturn", "Jupiter"};
        Array array = new Array(str);
```

```
        System.out.println("before = " + array.stringList());
        array.removeIf(new UnaryPredicateAdaptor(this, "longer5"));
        System.out.println("after  = " + array.stringList());
    }

    public boolean longer5(Object string)
    {
        return ((String) string).length() > 5;
    }

    public static void main(String[] args) throws Exception
    {
        new FunctionAdaptor4();
    }
}
```

Output:

```
before = Sun, Mercury, Venus, Earth, Mars, Saturn, Jupiter
after  = Sun, Venus, Earth, Mars
```

Appendix A

Benchmarking

CAL comes with a benchmarking suite that benchmarks various functions in `Array`, `Deque`, `DList`, and `SList`. It has also the ability to execute the same tests for ObjectSpace's JGL™ version 2.0.2 [Obj], and some `java.util.Vector` test in the Java™ JDKIn the last section of this chapter, we are looking at performance differences between CAL JGL and JDK.

These tests can be executed by you, since we have included all binaries and source for the benchmark utility.

To run the benchmark utility you have to specify a number of parameters. These parameters can be found by running the following command:

```
java COM.X3M.cal.benchmark.Main -h
```

This requires that the `benchmark.zip` is in your `CLASSPATH`.

Results are not included in this release of the documentation, because of lack of time.

Appendix B

An Introduction to the O-notation

Througout the reference documentation the word *time-complexity* is frequently used. This is a form of measuring the performance of a task. The performance mesurement helps to make wise choices among components, such as choosing whether to use a list, array, or deque to represent a sequence in a perticular application or applet.

B.1 Definitions

In most cases, talking about the computing time of an algorithm is simplified by classifying all inputs to the algorithm into subsets characterized by some somple parameter(s). For algorithms that work on containers, usually the size n of the container is a convenient choice of parameter. For inputs of size n, we then consider the *maximum time*[1], (n), the algorithm takes.

To simplify matters further, we focus on how (n) behaves for large n, and instead of trying to write a precise formula for (n), we just look for a simple function that provides an *upper bound* on the function. For example, we might have $(n) \leq cn$ for some constant c and all sufficiently large n. We say in this case that "the time grows worst linearly with the size of input". The constant c might have to be chosen differently when the algorithm is run on different machines, so one last simplification that is frequently made is to express such bounds in a way that hides the constant. This is done by using the -notation. For example, the above relation is written $(n) = (n)$. In general, if there is some function f such that $(n) \leq cf(n)$ for some constant c and all sufficiently large n, we write $(n) = (f(n))$.

In the CAL documentation, five cases of the -notation are used:

1. $(n) = (n)$
   This is the case mentioned in the previous paragraph. The algorithm is said to have a *linear time bound*, or to be *linear*.
2. $(n) = (n^2)$
   The algorithm is said to hava a *quadratic time bound*, or to be *quadratic*.

[1] This is also called *worst-case* time.

3. $(n) = \quad (\log n)$
   The algorithm is said to have a *logarithmic time bound*, or to be *logarithmic*.
4. $(n) = \quad (n \log n)$
   The algorithm is said to have a $n \log n$ *time bound*.
5. $(n) = \quad (1)$
   The algorithm is said to have a $\quad (1)$ *time bound* or to be *constant time*. In this case the -notation is shorthand for $\quad (n) \leq c$.

In each of this cases, it is importaint to keep in mind that we are characterizing computing times with upper bounds on *worst-case computing times*. This can be misleading if the worst-case occurs extremely rarely, as in the CAL `sort` algorithm. The worst-case time for this algorithm is quadratic, which is much slower than two other CAL sorting algorithms, `partialSort` and `stableSort`. For most situations the `sort` algorithm is the best choice, since it's faster *on the avareage* than the other algorithms. The inputs that cause worst-case for `sort` are in fact so rare that in most situations they can be ignored.

B.2 Amortized Time

In several cases, the most useful characterization of an algorithm's computing time is neither worst-case time nor average time, but *amortized time*. Amortized time can be a useful way to describe the time an operation on some container takes in cases *where the time can vary widely* as a sequence of the operations is done, but the total time for a sequence of n operations has a *better bound* than just n times the worst-case time. The amortized time for an operation is the total time for a sequence of N operations divided by n.

B.3 Limitations

The -notation itself has well-known limitations. The time for an $\quad (n)$ algorithm might grow at a rate slower than linear growth since it is only an upper bound, or it might grow faster than a linear rate for small n, since the bound only holds for large n. Two $\quad (n)$ algorithms can differ dramatically in their computing times. One could be uniformly 2, or 10, or 100 times faster than the other, but the -notation supresses all such differences. As one moves from one compiler or hardware architecture to another, the hidden constants can change, perhaps enough to alter the case for choosing one algorithm over another. For these and many other resons, it is advisable to do empirical preformance testing of programs under situations that approximate, to the extent possible, the environment and data that are likely to be encountered in production runs.

Appendix C

The Container and Algorithm Library License Agreement and Warranty

This license is an Agreement between you ("Licensee") and X^3M Solutions.

1. **License Grant.**
   X^3M Solutions grants to you ("Licensee") a non-exclusive, non-transferable, worldwide, royal-free license to use The Container and Algorithm Library Version 1.0b2 (the "Software") and related materials without charge provided the Licensee agrees to all the terms and conditions of this Agreement.

   By downloading, using, or copying the Software or any portion thereof, Licensee agrees to abide by the intellectual property laws, and all other applicable laws of the United States of America and to all of the terms and conditions of this Agreement, and agrees to take all necessary steps to ensure that the terms and conditions of this Agreement are not violated by any person or entity under the Licensee's control or in the Licensee's service.

   Licensee shall maintain the copyright and trademarks notices on the materials within or otherwise related to the Software, and not alter, erase, deface or overprint any such notice.
2. **Restrictions.**
   Except as specifically provided in this Agreement, Licensee is expressly prohibited from copying, modifying, merging, selling, leasing, assigning, or transferring in any matter, the Software or any portion thereof.

   Licensee may copy materials within or otherwise related to the Software that bear the X^3M Solutions copyright only as required for backup purposes or for use solely by the Licensee.

   Licensee may not distribute in any form of electronic or printed communication the materials within or otherwise related to the Software that bear the X^3M Solutions copyright, including but not limited to the source code, documentation, examples, and benchmarks, without prior written consent from X^3M Solutions. Send any requests for limited distribution rights to `info@x3m.com`.

   The Licensee may distribute binaries (.class files) derived from or contained within the Software ("Binaries") provided that:

(a) The Binaries are not integrated, bundled, combined, or otherwise associated with a Java development environment ("JDE") or a Java development tool; and

(b) The Binaries are not a documented part of any distribution material.

3. **Disclaimer of Warranty.**
SOFTWARE IS PROVIDED "AS IS", WITHOUT ANY WARRANTY OF ANY KIND. ALL EXPRESS OR IMPLIED REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HERBY EXCLUDED.

4. **Limitation of Liability.**
X3M SOLUTIONS DA SHALL NOT BE LIABLE FOR ANY DIRECT OR CONSEQUENTIAL DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING, OR DISTRIBUTING SOFTWARE.

5. **Termination.**
Licensee may terminate this License at any time by destroying all copies of the Software. This License will terminate immediately without notice from X^3M Solutions if Licensee fails to comply with any provision of this License. Upon such termination, Licensee must destroy all copies of the Software.

6. **Governing Law.**
Any action related to this License will be governed by California law and controlling U.S. federal law. No choice of law rules of any jurisdiction will apply.

Bibliography

[BM95] Javier Barreiro and David R. Musser. An stl hash table implementation with gradual resizing. Technical report, Computer Science Department, Rensseleaer Polytechnic Institute, February 1995. Available by anonymous ftp from `ftp://ftp.cs.rpi.edu/pub/stl/hashimp2.tar.gz`.

[Fla96] David Flanagan. *Java in a Nutshell.* O'Reilly & Associates, Inc., 1996. This handbook is an indispensable quick reference designed to lie flat and wait faithfully by the side of every Java programmer's keyboard.

[Obj] ObjectSpace, Inc. *JGL Version 2.0.2 User Guide.* JGL manual and library can be found at ObjectSpace's web site, `http://www.objectspace.com`.

[Sit96] Edward Sitarski. Hats: Hashed array trees. *Dr. Dobb's Journal*, 21(9):107–110, September 1996. C/C++ code for this article can be found at Dr. Dobb's Journal web site, `http://www.ddj.com`.

[SL95] Alexander A. Stepanov and Meng Lee. The standard template library. Technical report, Hewlett-Packard Laboratorium, September 1995. available by anonymous ftp from `ftp://ftp.cs.rpi.edu/pub/stl/doc.ps.gz`.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[THCR94] Charles E. Leiserson Thomas H. Cormen and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachussets, London, England, 1994. Introduction to Algorithms is a rigorous and comprehensive guide to algorithms and datastructures. It is useful both as an introductory book and as a reference.

[WCS96] Larry Wall, Tom Christiansen, and Randy L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., 1996. If you buy only one book about Perl, no one in the Perl community will question your judgement in buying this one.

Index