The Java™ Generic Library

JGL 1.1 User Guide

ObjectSpace

Version 1.0 last modified 7/19/96
Version 1.1 last modified 9/13/96

Copyright 1996 by ObjectSpace, Inc.

Java is a trademark of Sun MicroSystems Incorporated
Navigator is a trademark of Netscape Communications Corporation.
All other brand or product names are trademarks or registered trademarks of their respective holders.

Table Of Contents

Table Of Contents **3**
Preface **5**
Dedication 5
Acknowledgements 5
Purpose and Audience 5
How To Use This Document 5
Installation 6
Package Layout and Contents 6
Reading the Online Documentation 6
Compiling The Library 7
Running Examples 7
Submitting Bug Reports and New Containers/Algorithms 7
Getting Updates 7
The Java Generic Library (JGL) **9**
Containers 10
Algorithms 12
Function Objects 14
Iterators 15
Containers **17**
Container Interfaces 18
Storing Objects and Primitives 18
Adding, Printing, Clearing, and Sizing 19
Enumerating 20
Copying, Comparing, Assignment, and Cloning 20
Swapping 22
Storing Primitives 23
Storing User-Defined Objects 24
Error Handling 25
Array Adapters 27
Sequences **29**
Pushing, Popping, and Index-based Access 30
Counting, Finding, Removing, and Replacing 31
Inserting 32
Ensuring Capacity 33
Constructing From a Native Java array 35
Splicing 36
Miscellaneous 37
Maps **38**
Putting and Getting 39
Enumerating 41
Ordering 42
Removing and Counting 43
Matching Keys 44
Pairs 45
Sets **46**
Putting and Getting 47

Ordering 50
Removing and Counting 51
Matching Objects 52
Union, Intersection, and other Set Operations 53
Queues and Stacks 54
Stack 55
Queue 57
PriorityQueue 58
Array Adapters 60
Using a Native Java Array as a Container 61
Using a JDK Vector as a Container 62
Algorithms 64
List of Algorithms by Category 64
Applying 67
Copying 68
Counting 69
Finding 70
Filtering 71
Replacing 72
Reversing 73
Sorting 74
Transforming 75
Applying an Algorithm to a Sub-Range of a Container 76
Function Objects 78
Predicates 79
General Functions 83
Binders 85
Composers 86
Creating your own Function Objects 87
Iterators 88
Iterators and Algorithms 92
Iterators and Containers 93
Reverse Iterators 95
Insertion Iterators 96
Iterating over Java Streams 97
Summary of Classes 98
Containers 98
Exceptions 99
Algorithms 99
Function Objects 99
Iterators 101
Utilities 101
Appendix 102
Benchmarks 102
A Comparison of JGL vs. STL 104
    Feature Enhancements 104
    Compatibility Features 104
    Differences 105
    Code Comparisons 105

Preface

Dedication

To my family: Diane, Emily, and Laura.
- Paul Jenkins
To Pooh Bear and Michael Ryan
- David Whitmore
To Lotus Engineering, whose cars are almost as fast as our software
- Graham Glass

Acknowledgements

We proudly acknowledge the following people who contributed to JGL 1.1: Mike Colon, Rob Davis, David, Bill Joy, Carlo Prelz, Rene Schmit, Guy Steele, Lewis Stiller, and Walter Szewelanczyk.

Purpose and Audience

The Java Generic Library (JGL) is a comprehensive set of reusable data structures and algorithms. It embraces and extends the library that comes with every Java Development Kit (JDK). This users guide describes the JGL facilities in detail using a mix of text, diagrams, and sample code. Any programmer who wishes to use JGL will benefit from first reading this guide.

If you wish to see the interface to a particular JGL class, consult the online HTML documentation.

How To Use This Document

First, read the rest of this preface, which contains information about how to install JGL, access the online HTML documentation, run the example code, and get additional information about JGL.

Then, read the first chapter —"The Java Generic Library" — which contains an overview of JGL and its main facilities.

Next, read the "Containers" chapter which describes the common characteristics of all the JGL containers. The 5 chapters that follow it — "Sequences", "Maps", "Sets", "Queues & Stacks", and "Array Adapters" — describe each different kind of container in detail.

The remaining chapters are optional but very useful. For example, to learn about JGL generic algorithms, which allow you to perform sorting, filtering, and other useful operations, read the "Algorithms" and "Function Objects" chapters. For information about advanced iterators, which allow you to perform tasks like enumerating a container in a reverse direction or applying an algorithm to a portion of a container, read the "Iterators" chapter.

For a summary of every JGL class, consult the chapter called "Class Summary".

The Appendix contains some benchmarks and a comparison of JGL against other data structure libraries.

Installation

Once the JGL files have been installed, you must change/set the `CLASSPATH` environment variable to include the JGL home directory. Most users have their `CLASSPATH` set in `autoexec.bat`. For example, if you installed JGL into the directory `C:\jgl_1_1` then you must set your `CLASSPATH` to include `C:\jgl_1_1`.

The rest of this document assumes that you have obtained a copy of the Java Generic Library, performed the installation process, and installed it into the high level directory `\jgl_1_1`.

Package Layout and Contents

Here is a map of the JGL directory structure:

`\jgl_1_1`	
`LICENSE.TXT`	The JGL license agreement.
`README.TXT`	A file that should be read immediately after JGL is installed.
`\benchmarks`	
`*.java`	The Java source code for the JGL benchmarks
`*.txt`	Files that contains some sample benchmarks
`\doc`	
`JGL.html`	The JGL HTML home page.
`\api`	
`*.html`	HTML API docs and examples.
`\images`	.gif files that are referenced by the HTML documentation.
`\user`	
`*.html`	HTML user guide.
`\images`	.gif files that are referenced by the HTML documentation.
`\examples`	
`*.java`	The Java source code for all JGL example code.
`\jgl`	
`*.class`	The JGL object files.
`\src`	
`*.java`	The Java source code for JGL.

Reading the Online Documentation

All of the JGL classes and examples have full online HTML documentation. The entry point into this documentation is `\jgl_1_1\doc\JGL.html`, from which you can jump to the user guide, the API guide, and frequently asked questions (FAQs). To view this from the Internet Explorer, use the "File>Open" menu option. To view this file from the Netscape Navigator, use the "File>Open File in Browser"menu option.

To view a particular class, select the class from the tree structure in the API guide. Most classes have a hot link to some examples from the "See Also" section.

Compiling The Library

The JGL installation process automatically installs the JGL object files, so you do not need to compile the library in order to use it. However, if you ever wish to recompile the library, change to the `\jgl_1_1\src` directory and then enter:

```
javac -O *.java -d \jgl_1_1
```

This command recompiles JGL as optimized code and places the resulting object files into a directory called `jgl` in the `\jgl_1_1` directory. Note that due to circularities in class imports, many Java environments will not allow you to recompile JGL on a file-by-file basis. This is a weakness with the environment, not with the design or implementation of JGL.

Running Examples

To compile the example called *xxx*`.java`, change into the `\jgl_1_1\examples` directory and enter:

```
javac xxx.java
```

Once the example has compiled, you may run it by entering:

```
java xxx
```

All of the examples in this user guide are located in the `\jgl_1_1\examples` directory.

Submitting Bug Reports and New Containers/Algorithms

We welcome bug reports, suggestions for improvement, and new containers/algorithms. All significant contributions to JGL will be acknowledged in future releases. Please send your valuable feedback to `jgl@objectspace.com`.

Getting Updates

If you chose to receive automatic email notification when you downloaded JGL, you will automatically be notified of new JGL releases and other JGL-related news items. If you didn't choose to receive email notification but wish you had, send email to `jgl@objectspace.com` and request future notification.

The Java Generic Library (JGL)

All Java™ programmers need a library of reusable data structures. The Java Developer's Kit (JDK) only contains two data structures - a `Vector` and a `Hashtable` - which is hardly sufficient for serious software development. In addition, the JDK does not include algorithms for sorting, filtering, and other common tasks.

In response to this situation, a collaboration was initiated between the ObjectSpace Components Group and the ObjectSpace Research Group to create the Java™ Generic Library (JGL, pronounced "Juggle"). In order to accelerate the spread of Java as a powerful general purpose programming language, ObjectSpace has decided to make JGL ***absolutely free*** for commercial use! This guarantees that all Java programmers will have instant access to the most sophisticated containers and algorithms on the market. Here is a list of reasons why all Java programmers should use JGL as one of their Java libraries:

Comprehensive Features

- JGL includes 11 highly optimized data structures including sequences, maps, sets, queues, and a stack. Both ordered and hashing versions of set and maps are available.
- JGL contains over 40 reusable algorithms such as sorting, union, intersection, and many more.
- JGL is multi-thread safe. JGL containers protect their methods with the `synchronized` keyword so that they work correctly in the presence of multiple threads.

Easy to Use

- With complete HTML API documentation and a comprehensive HTML user guide packed with over 100 examples, JGL is very easy to learn and use.
- With thousands of users already, ObjectSpace has been able to gather rapid feedback to fine-tune the usability of JGL.
- Although a small and efficient library is critical, ObjectSpace has paid close attention to meeting usability requirements without sacrificing the performance.

Performance

- JGL has been continually optimized for performance. In almost every case it either meets or beats any equivalent functionality available today.
- Don't just take our word for it! ObjectSpace ships a suite of performance benchmarks with JGL. These benchmarks compare JGL performance with the JDK data structures.
- JGL is architected to be lean and mean, making it ideal for Internet distribution.

Compatibility

- JGL embraces and extends the JDK. It is compatible with the `Exception`, `Enumeration`, `Vector`, and `Dictionary` classes.
- All JGL algorithms operate not only on JGL containers, but also on Java arrays of objects, Java arrays of primitives, and any JDK `Vector`.
- As long as you adhere to standard Java language conventions, you don't have to add any code to your classes in order to store them in a JGL container.

This user guide describes the use of JGL using a mix of text and example programs. The rest of this chapter provides an overview of the main features of JGL - containers, algorithms, function objects and iterators. Subsequent chapters describe each JGL topic in detail.

Containers

This section contains some examples of JGL containers. See the “Containers” chapter for more details.

JGL includes 11 highly optimized data structures that satisfy most programmers’ needs. Their performance either meets or beats any other commercially available Java container library, and have been engineered for performance and ease of use. In addition, all JGL containers work correctly in multithreaded situations.

The following example uses a JGL `HashMap` to associate a chemical symbol with its full name. Note that the entire container can be printed easily using the `println()` function.

Example Overview1.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Overview1
  {
  public static void main( String[] args )
    {
    HashMap chemicals = new HashMap();
    chemicals.put( "Ca", "Calcium" );
    chemicals.put( "Au", "Gold" );
    chemicals.put( "He", "Helium" );
    System.out.println( "chemicals = " + chemicals );
    System.out.println( "Au means " + chemicals.get( "Au" ) );
    }
  }
```

Output

```
chemicals = HashMap( Pair( Au, Gold ), Pair( He, Helium ), Pair( Ca, Calcium )
)
Au means Gold
```

The next example illustrates some set operations using the `HashSet` class. Note how simple it is to obtain the union and intersection of two sets.

Example Overview2.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Overview2
  {
  public static void main( String[] args )
    {
    HashSet set1 = new HashSet();
    set1.add( "red" );
    set1.add( "blue" );
    set1.add( "green" );

    HashSet set2 = new HashSet();
    set2.add( "yellow" );
    set2.add( "blue" );

    HashSet set3 = set1.union( set2 );
    HashSet set4 = set1.intersection( set2 );
```

```
    System.out.println( "set1 = " + set1 );
    System.out.println( "set2 = " + set2 );
    System.out.println( "union of set1 and set2 = " + set3 );
    System.out.println( "intersection of set1 and set2 = " + set4 );
    }
  }
```

Output

```
set1 = HashSet( blue, green, red )
set2 = HashSet( blue, yellow )
union of set1 and set2 = HashSet( blue, green, red, yellow )
intersection of set1 and set2 = HashSet( blue )
```

One of JGL's strengths is its seamless compatibility with the existing JDK utility classes. For example, you can use a standard JDK `Enumeration` object to iterate through every element of any JGL container. The following example uses an `Enumeration` to enumerate the contents of an `Array` (growable array) and a `DList` (doubly-linked list).

Example Overview3.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;
import java.util.Enumeration; // JDK enumeration class.

public class Overview3
  {
  public static void main( String[] args )
    {
    Array array = new Array(); // Growable array.
    array.add( "nemesis" );
    array.add( "dig" );
    array.add( "myst" );

    Enumeration iterator = array.elements();
    while( iterator.hasMoreElements() )
      System.out.println( iterator.nextElement() );
    System.out.println();

    DList list = new DList(); // Doubly-linked list.
    list.add( "agatha" );
    list.add( "beauty" );
    list.add( "truth" );

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

Output

```
nemesis
dig
myst

agatha
beauty
truth
```

Algorithms

This section contains some examples of the power of JGL algorithms. See the “Algorithms” chapter for more details.

One of the main strengths of JGL is its complement of 40+ reusable algorithms that can perform tasks ranging from a simple filtering operation to a complex quicksort. Instead of embedding algorithms into a containers, JGL algorithms are encapsulated within their own class and can operate upon a wide variety of data structures, including JGL containers, JDK containers, and native Java arrays. All algorithms that are closely related are encapsulated as class methods of a single class. For example, all of the sorting algorithms are methods in `Sorting`.

To demonstrate the power and flexibility of this approach, the next three sections apply the single JGL reusable sorting algorithm to three different kinds of data structures.

Example Overview4.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Overview4
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( new Integer( 3 ) );
    array.add( new Integer( -1 ) );
    array.add( new Integer( 2 ) );
    System.out.println( "Unsorted Array = " + array );
    Sorting.sort( array ); // Sort the Array.
    System.out.println( "Sorted = " + array );
    }
  }
```

Output

```
Unsorted Array = Array( 3, -1, 2 )
Sorted = Array( -1, 2, 3 )
```

The second example in this series uses the same sorting algorithm to sort a JDK `Vector`. JGL supplies simple “wrapper” classes to make a JDK `Vector` look like a JGL container so that the algorithms can work directly upon them.

Example Overview5.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;
import java.util.Vector;

public class Overview5
  {
  public static void main( String[] args )
    {
    Vector vector = new Vector();
    vector.addElement( new Integer( 3 ) );
    vector.addElement( new Integer( -1 ) );
    vector.addElement( new Integer( 2 ) );
    System.out.println( "Unsorted java.util.Vector = " + vector );
    // A VectorArray makes a JDK vector look like a JGL container.
    VectorArray vectorArray = new VectorArray( vector );
```

```
    Sorting.sort( vectorArray ); // Sort the JDK Vector.
    System.out.println( "Sorted = " + vector );
    }
  }
```

Output

```
Unsorted java.util.Vector = [3, -1, 2]
Sorted = [-1, 2, 3]
```

The third and final example uses the same sorting algorithm to sort a native Java array of `ints`. In a similar manner to the last example, JGL supplies simple "wrapper" classes to make native Java arrays look like a JGL container.

Example Overview6.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Overview6
  {
  public static void main( String[] args )
    {
    int ints[] = { 3, -1, 2 };
    IntArray intArray = new IntArray( ints );
    System.out.println( "Unsorted native int array = " + intArray );
    Sorting.sort( intArray ); // Sort the primitive Java array of ints.
    System.out.print( "Sorted native array = " );
    for( int i = 0; i < ints.length; i++ )
      System.out.print( ints[ i ] + " " );
    System.out.println();
    }
  }
```

Output

```
Unsorted native int array = int[]( 3, -1, 2 )
Sorted native array = -1 2 3
```

JGL includes many different kinds of algorithm. This last example uses the random shuffling algorithm to shuffle the contents of a `Deque` (double-ended queue).

Example Overview7.java

```
import jgl.*;

public class Overview7
  {
  public static void main( String[] args )
    {
    Deque deque = new Deque();
    deque.add( "your" );
    deque.add( "mission" );
    deque.add( "jim" );
    System.out.println( "Original deque = " + deque );
    Shuffling.randomShuffle( deque );
    System.out.println( "Shuffled deque = " + deque );
    }
  }
```

Output

```
Original deque = Deque( your, mission, jim )
Shuffled deque = Deque( jim, your, mission )
```

Function Objects

This section contains a few examples of JGL function objects. See the “Function Objects” chapter for more details.

Many algorithms can accept an optional function object to modify the way that they work. A function object is an object that encapsulates a single function and can execute it on one or more arguments. JGL contains over 30+ reusable function objects.

The default sorting algorithm orders elements in ascending order based on their hash code. The following example uses a `GreaterInteger` function object to make the sorting algorithm order elements in descending order.

Example Overview8.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Overview8
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( new Integer( 3 ) );
    array.add( new Integer( -1 ) );
    array.add( new Integer( 2 ) );
    System.out.println( "Unsorted Array = " + array );
    BinaryPredicate predicate = new GreaterInteger();
    Sorting.sort( array, predicate ); // Sort in descending order.
    System.out.println( "Sorted = " + array );
    }
  }
```

Output

```
Unsorted Array = Array( 3, -1, 2 )
Sorted = Array( 3, 2, -1 )
```

Another couple of algorithms that can accept a function object are `countIf()` and `removeCopyIf()`. `countIf()` counts the elements in a container that satisfy a given condition, and `removeCopyIf()` copies elements that don’t satisfy a condition from one container to another. Both of these algorithms are illustrated by the following example.

Example Overview9.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Overview9
  {
  public static void main( String[] args )
    {
    Array array1 = new Array();
    array1.add( new Integer( 3 ) );
```

```
    array1.add( new Integer( -1 ) );
    array1.add( new Integer( 2 ) );
    UnaryPredicate predicate = new PositiveInteger();
    int n = Counting.countIf( array1, predicate );
    System.out.println( "# of positive numbers in " + array1 + " = " + n );
    Array array2 = new Array();
    Removing.removeCopyIf( array1, array2, predicate );
    System.out.println( "Array without positive numbers = " + array2 );
    }
  }
```

Output

```
# of positive numbers in Array( 3, -1, 2 ) = 2
Array without positive numbers = Array( -1 )
```

Iterators

This section contains a brief overview of JGL iterators. See the “Iterators” chapter for more details.

In addition to supporting the JDK `Enumeration` type, JGL includes several more powerful classes of iterator. For example, a `ReverseIterator` allows you to iterate backwards over a container. JGL also includes iterators for iterating through an I/O stream and iterating over a virtual container of random values. The following example uses a reverse iterator to enumerate the contents of an `Array` in reverse order.

Example Overview10.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Overview10
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( "jolly" );
    array.add( "good" );
    array.add( "show" );

    // Create a reverse iterator positioned at the end of the array.
    ReverseIterator iterator = new ReverseIterator( array.end() );
    while( iterator.hasMoreElements() )
      System.out.println( iterator.nextElement() );
    }
  }
```

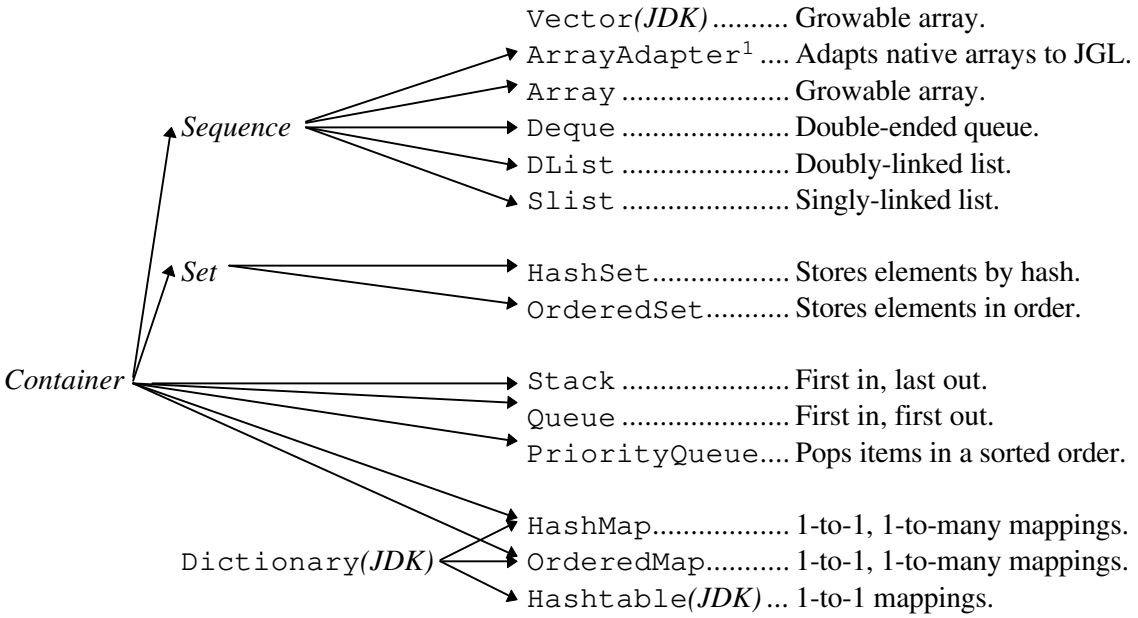
Output

```
show
good
jolly
```

Containers

JGL includes 11 highly optimized data structures for general purpose programming. It also includes 10 array adapters that allow you to apply JGL algorithms to native arrays and JDK Vectors. The following diagram shows all of the container classes in JGL and how they relate to the container classes that are already part of the JDK. Interfaces are indicated using *italics* and classes are indicated using `courier`. Classes that are already part of JDK are indicated by *(JDK).*

Note how tightly JDK integration has been achieved. For example, the JGL `HashMap` and `OrderedMap` extend the existing JDK `Dictionary` class. The only time there is overlap between JGL and JDK is when the JGL container has significantly more functionality or higher performance than its JDK equivalent.



The rest of this chapter contains information pertinent to every JGL container. The “Sequences”, “Maps”, “Sets”, “Stacks and Queues”, and “Array Adapters” chapters describe the methods and interfaces specific to each category of container.

[1] There are 10 subclasses of `ArrayAdapter`, one for each primitive type, one for Object, and one for a JDK Vector.

Container Interfaces

Every JGL container implements the `Container` interface, which defines the methods that are common to all JGL containers. In addition, most JGL containers implement a more specialized interface related to their category. For example, `Array`, `Deque`, `DList`, and `SList` all implement the `Sequence` interface.

The `Container` interface defines the following methods:

- `add` - adds an element to the container
- `clear` - removes every element from the container
- `clone` - returns a shallow copy of the container
- `elements` - returns an `Enumeration` positioned at the start of the container
- `empty` - returns `true` if the container contains no elements
- `equals` - returns `true` if the contents match those of another container of the same type
- `size`/`maxSize` - returns the current/maximum number of elements in the container
- `start`/`finish` - returns an iterator positioned at the start/finish of the container
- `toString` - returns a string that describes the container

The rest of this chapter describes all of these methods except for `start()` and `finish()`, which are described in the "Iterators" chapter.

Storing Objects and Primitives

JGL follows the same philosophy as the JDK when it comes to the way that objects and primitives are stored:

- *A container can store instances of any Java class*. Note that you can store a `String`, a `Date`, and an `Integer` in the same container. When you retrieve an object, you must cast the object back to the required type before applying a type-specific method. Java's dynamic cast-checking mechanism ensures that any attempt to cast an object to the wrong type causes a runtime exception.

- *A container cannot directly store a primitive*. Primitives can only be stored in containers using a special technique described in the "Storing Primitives" section of this chapter.

Note that unlike C++ template containers, you cannot create a JDK or JGL container that can only hold instances of a specific class of object.

Adding, Printing, Clearing, and Sizing

Every JGL container allows you to add an object, print the container, clear the container, and size the container. These behaviors are illustrated by the following example.

Example Container1.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Container1
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( "triangle" );
    array.add( "square" );
    array.add( "pentagon" );
    array.add( "hexagon" );
    System.out.println( "array = " + array );
    System.out.println( "array.size() = " + array.size() );
    System.out.println( "array.empty() = " + array.isEmpty() );
    array.clear();
    System.out.println( "after array is cleared..." );
    System.out.println( "array.size() = " + array.size() );
    System.out.println( "array.empty() = " + array.isEmpty() );
    }
  }
```

Output

```
array = Array( triangle, square, pentagon, hexagon )
array.size() = 4
array.isEmpty() = false
after array is cleared…
array.size() = 0
array.isEmpty() = true
```

Enumerating

Every JDK container defines a method called `elements()` that returns a JDK `Enumeration` object positioned at its first element. Once this object is obtained, you may enumerate every element of the container by using the `hasMoreElements()` and `nextElement()` methods. For ease of use and backwards compatibility, every JGL container also defines an `elements()` method that returns an `Enumeration`. The following example uses an `Enumeration` object to enumerate every object in an `Array`.

Example Container2.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;
import java.util.Enumeration; // JDK enumeration class.

public class Container2
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( "triangle" );
    array.add( "square" );
    array.add( "pentagon" );
    array.add( "hexagon" );

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

Output

```
triangle
square
pentagon
hexagon
```

Copying, Comparing, Assignment, and Cloning

There are two well-known ways to copy a container:

- A *shallow copy* means that the original container and the copy share references to the same objects. This is the fastest kind of copy and is sufficient for most programming purposes.

- A *deep copy* means that the original container and the copy do not share references to the same objects. Instead, every element of the original container is copied as it is added to the new container.

The JGL copy constructors always perform a shallow copy. In addition, you may replace the contents of a container with a shallow copy of another container by using the `copy()` method, or obtain a shallow clone of a container using the standard `clone()` method.

To compare two containers of the same type for equality, use the standard `equals()` method. JGL follows the standard Java convention and extends `Container` from `Cloneable`. The following example illustrates all of these methods.

Example Container3.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Container3
  {
  public static void main( String[] args )
    {
    Array array1 = new Array();
    array1.add( "triangle" );
    array1.add( "square" );
    array1.add( "pentagon" );
    System.out.println( "array1 = " + array1 );

    // Illustrate copy construction.
    Array array2 = new Array( array1 );
    System.out.println( "array2 = " + array2 );
    System.out.println( "array1.equals( array2 ) = "+ array1.equals( array2 )
);

    // Illustrate assignment using copy().
    Array array3 = new Array();
    array3.add( "heptagon" );
    array3.add( "octagon" );
    System.out.println( "before copy, array3 = " + array3 );
    array3.copy( array1 );
    System.out.println( "after copy, array3 = " + array3 );

    // Illustrate cloning.
    Array array4 = (Array) array1.clone();
    System.out.println( "array4 = " + array4 );
    }
  }
```

Output

```
array1 = Array( triangle, square, pentagon )
array2 = Array( triangle, square, pentagon )
array1.equals( array2 ) = true
before copy, array3 = Array( heptagon, octagon )
after copy, array3 = Array( triangle, square, pentagon )
array4 = Array( triangle, square, pentagon )
```

Swapping

Every JGL container allows you to efficiently exchange its contents with those of another container of the same type by using the `swap()` method. This is illustrated by the following example.

Example Container4.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Container4
  {
  public static void main( String[] args )
    {
    Array array1 = new Array();
    array1.add( "ape" );
    array1.add( "bat" );
    array1.add( "cat" );

    Array array2 = new Array();
    array2.add( "red" );
    array2.add( "blue" );

    // Illustrate swapping.
    System.out.println( "array1 = " + array1 + ", array2 = " + array2 );
    array1.swap( array2 );
    System.out.println( "array1 = " + array1 + ", array2 = " + array2 );
    }
  }
```

Output

```
array1 = Array( ape, bat, cat ), array2 = Array( red, blue )
array1 = Array( red, blue ), array2 = Array( ape, bat, cat )
```

Storing Primitives

The JGL containers can only contain objects. To store primitives, "wrap" them in their object equivalents defined in `java.lang`. Here is a table of primitives and their object equivalents:

Primitive	Object Equivalent
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

The following example shows how an `Array` can be used to store a variety of primitives.

ExampleContainer5.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Container5
  {
  public static void main( String[] args )
    {
    Array Array = new Array();
    Array.add( new Integer( 2 ) );
    Array.add( new Boolean( false ) );
    Array.add( new Character( 'x' ) );
    Array.add( new Float( 3.14F ) );
    System.out.println( "Array = " + Array );
    }
  }
```

Output

```
Array = Array( 2, false, x, 3.14 )
```

Note that the JGL algorithms are smart enough to convert these "wrapper" objects to and from their primitive equivalents when necessary. See the "Algorithms" and "Iterators" chapters for examples.

Storing User-Defined Objects

There are three special methods that are invoked by JGL containers on the objects they contain:

equals *boolean equals(Object object)*
Return true if I am equal to the specified object. By default, JGL algorithms and containers use this function when comparing elements.

hashCode *int hashCode()*
Return my hash code. `HashMap` and `HashSet` use this value to store and locate objects. In addition, `OrderedMap` and `OrderedSet` use this value by default to order their elements. Note that if two objects match using `equals()`, their hash codes must be identical.

toString *String toString()*
Return a string that describes me. JGL uses this function when printing containers.

Each of these methods has a default implementation in `java.lang.Object`. If you wish to store instances of your classes into any JGL container, you should override these methods when necessary.

For example, here is a `Company` class that overrides each of these methods appropriately. Note that two companies are considered to match when their names are the same.

Company.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Company
  {
  String myName;

  public Company( String name )
    {
    myName = name;
    }

  public String toString()
    {
    return "Company( " + myName + " )";
    }

  public int hashCode()
    {
    return myName.hashCode();
    }

  public boolean equals( Object object )
    {
    return object instanceof Company &&
           myName.equals(((Company) object).myName);
    }
  }
```

The following example stored instances of `Company` into a `HashMap`. Each instance of `Company` is placed into the `HashMap's` underlying hash structure using its `hashCode()` function.

Example Container6.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;
import Company;

public class Container6
  {
  public static void main( String[] args )
    {
    Company company1 = new Company( "ObjectSpace" );
    Company company2 = new Company( "Sun Microsystems" );

    HashMap headquarters = new HashMap();
    headquarters.put( company1, "Texas" );
    headquarters.put( company2, "California" );

    String location = (String) headquarters.get( company1 );
    System.out.println( "Headquarters of " + company1 + " = " + location );
    }
  }
```

Output

```
The headquarters of Company( ObjectSpace ) = Texas
```

Error Handling

A JGL error causes a Java exception to be thrown. Methods that can cause an exception are commented in the source code using the `@exception` tag and are documented in the online help. Whenever possible, JGL throws the standard JDK exceptions that are defined in `java.lang`. The only exception that is unique to JGL is `InvalidOperationException`.

Here is a hierarchy of the exceptions that JGL can throw, together with a brief description of when the exception is thrown. All of these exception classes are part of the JDK except for the bolded class.

`Throwable` ..... the root of all throwable objects
  `Exception` ..... the root of all exceptions
    `RuntimeException` ..... the root of all runtime exceptions
      `InvalidOperationException` .. invalid operation attempted on a JGL container
      `IllegalArgumentException` ... an illegal argument was supplied to a method
      `IndexOutOfBoundsException` ..... an illegal index was supplied to an operation

Because all JGL exceptions inherit from `RuntimeException`, methods that can generate them do not have to explicitly declare them using `throws`.

The following examples catches an `IndexOutOfBoundsException`.

Example Container7.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Container7
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( "ape" );
    array.add( "cat" );
    try
      {
      Object object = array.at( 5 );
      }
    catch( IndexOutOfBoundsException exception )
      {
      System.out.println( "Caught " + exception );
      }
    }
  }
```

Output

```
Caught java.lang.IndexOutOfBoundsException: Attempt to access index 5 when
valid range is 0..1
```

The next example catches an `InvalidOperationException`.

Example Container8.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Container8
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    try
      {
      Object object = array.front();
      }
    catch( InvalidOperationException exception )
      {
      System.out.println( "Caught " + exception );
      }
    }
  }
```

Output

```
Caught jgl.InvalidOperationException: Array is empty
```

The last example in this series catches an `IllegalArgumentException`.

Example Container9.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Container9
  {
  public static void main( String[] args )
    {
    try
      {
      Array array = new Array( -2 );
      }
    catch( IllegalArgumentException exception )
      {
      System.out.println( "Caught " + exception );
      }
    }
  }
```

Output

```
Caught java.lang.IllegalArgumentException: Attempt to create an Array with a
negative size
```

Array Adapters

Most of the JGL algorithms such as `sort()` and `countIf()` can operate directly on a JGL container. In order for the algorithms to also work with JDK `Vectors` and native Java arrays, JGL includes special *array adapter* classes that wrap them in a `Container`-compliant interface. The following example uses an `IntArray` adapter to allow `sort()` to operate on a native array of `ints`.

Example Container10.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Container10
  {
  public static void main( String[] args )
    {
    int ints[] = { 3, -1, 2, 0, -6 };
    IntArray intArray = new IntArray( ints ); // Construct adapter class.
    System.out.println( "unsorted native int array = " + intArray );
    Sorting.sort( intArray ); // Sort native array.
    System.out.print( "sorted = " );
    for( int i = 0; i < ints.length; i++ )
      System.out.print( ints[ i ] + " " );
    System.out.println();
    }
  }
```

Output

```
unsorted native int array = int[]( 3, -1, 2, 0, -6 )
sorted = -6 -1 0 2 3
```

For more information about array adapters, read the chapter called “Array Adapters”.

Sequences

A sequence is a linear container that allows index-based random access and automatically expands to accommodate new elements. JGL includes four different kinds of sequence:

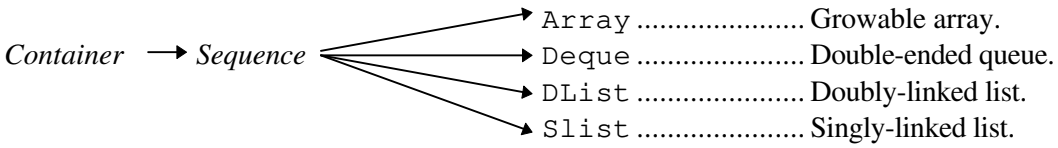
- An `Array` stores its elements internally in a single contiguous structure for very high performance random access. This storage approach allows elements to be added very efficiently to the end of the container, but much slower in other places. An `Array` is similar to a JDK `Vector` except that it is easier to apply JGL algorithms to an `Array` and it is possible to construct an `Array` from an existing native Java array.

- A `Deque` is a double-ended queue that has almost the same interface as an `Array` but stores its elements internally in an array of arrays. This implementation difference results in random access that is slower than that of an `Array` but allows fast insertion at its beginning or end.

- A `DList` stores its elements internally as a doubly linked list of nodes in which every node has a reference to a single element and a reference to its next and previous nodes. This approach yields very fast insertion at any point in the data structure but relatively slow random access as the `DList` must be traversed from its beginning to the target node.

- An `SList` stores its elements internally as a singly linked list of nodes in which every node has a reference to a single element and a reference to the next node. This approach uses less storage space than a `DList` but only allows high performance insertion at its beginning or its end.

Each of these classes implements the `Sequence` interface, which in turn extends the `Container` interface. Here is a diagram that illustrates these relationships:



All classes that implement the `Sequence` interface support the following methods:

- `pushFront/pushBack` — insert an item at the beginning/end
- `popFront/popBack` — remove the item at the beginning/end
- `front/back` — return the first/last item
- `at/put` — return/replace the item at a specified index
- `remove` — remove a particular value
- `replace` — replaces one value with another
- `count` — count the number of matching elements
- `indexOf` — return the index of a particular element
- `contains` — return `true` if a particular element is present

Array, DList, and SList include additional functionality that is unique to their class. The rest of this chapter describes these classes and methods in detail.

Pushing, Popping, and Index-based Access

All sequences allow you to read/write an element at a particular index using at() and put(), respectively. The first element of a sequence has index 0. If an attempt is made to access an element at an illegal index, an IndexOutOfBoundsException is thrown. add() and pushBack() are synonymous. The following example demonstrates all of the functions for pushing, popping, and index-based access using an Array.

Example Sequences1.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Sequences1
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.pushBack( "bat" ); // Add to end of array.
    array.add( "cat" ); // Synonym for pushBack().
    array.pushFront( "ape" ); // Insert at beginning of array.
    System.out.println( "array = " + array );
    System.out.println();

    System.out.println( "Demonstrate access" );
    System.out.println( "array.at( 0 ) = " + array.at( 0 ) ); // First
element.
    System.out.println( "array.front() = " + array.front() ); // First
element.
    System.out.println( "array.at( 2 ) = " + array.at( 2 ) ); // Last element.
    System.out.println( "array.back() = " + array.back() ); // Last element.

    System.out.println( "array.put( 1, \"fox\" )" );
    array.put( 1, "fox" ); // Replace second element.
    System.out.println( "array = " + array );

    array.popFront(); // Remove first element.
    System.out.println( "After popFront() = " + array );

    array.popBack(); // Remove last element.
    System.out.println( "After popBack() = " + array );
    }
  }
```

Output

```
array = Array( ape, bat, cat )

Demonstrate access
array.at( 0 ) = ape
array.front() = ape
array.at( 2 ) = cat
array.back() = cat
array.put( 1, "fox" )
array = Array( ape, fox, cat )
After popFront() = Array( fox, cat )
After popBack() = Array( fox )
```

Counting, Finding, Removing, and Replacing

All sequences include a wide range of useful methods for counting, finding, removing, and replacing elements. Many of these methods have two variations — one for performing the operation on the entire container, and another for performing the operation on a sub-range. The following example shows some of these variations in action using a `Deque`.

Example Sequences2.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Sequences2
  {
  public static void main( String[] args )
    {
    Deque deque = new Deque();
    deque.add( "ape" );
    deque.add( "bat" );
    deque.add( "cat" );
    deque.add( "bat" );
    deque.add( "bat" );
    deque.add( "cat" );
    System.out.println( "deque = " + deque );

    System.out.println( "deque.count( bat ) = " + deque.count( "bat" ) );
    int index = deque.indexOf( "bat" );
    System.out.println( "deque.indexOf( bat ) = " + index );
    deque.remove( index );
    System.out.println( "After deque.remove( " + index + " ) = " + deque );
    deque.replace( 0, 2, "bat", "BAT" );
    System.out.println( "After deque.replace( 0, 2, bat, BAT ) = " + deque );
    System.out.println( "deque.remove( cat ) = " + deque.remove( "cat" ) );
    System.out.println( "After deque.remove( cat ) = " + deque );
    }
  }
```

Output

```
deque = Deque( ape, bat, cat, bat, bat, cat )
deque.count( bat ) = 3
deque.indexOf( bat ) = 1
After deque.remove( 1 ) = Deque( ape, cat, bat, bat, cat )
After deque.replace( 0, 2, bat, BAT ) = Deque( ape, cat, BAT, bat, cat )
deque.remove( cat ) = 2
After deque.remove( cat ) = Deque( ape, BAT, bat )
```

Inserting

All sequences support a comprehensive range of insert methods that allow you to insert one or more elements at a specific index. The following example illustrates these features using a `DList`.

Example Sequences3.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Sequences3
  {
  public static void main( String[] args )
    {
    DList list = new DList();
    list.add( "bat" );
    list.add( "cat" );
    list.add( "dog" );
    System.out.println( "list = " + list );

    list.insert( 0, "ape" );
    System.out.println( "After insert at begin = " + list );

    list.insert( list.size(), "emu" );
    System.out.println( "After insert at end = " + list );

    list.insert( 3, 2, "fox" );
    System.out.println( "After list.insert( 3, 2, fox ) = " + list );
    }
  }
```

Output

```
list = DList( bat, cat, dog )
After insert at begin = DList( ape, bat, cat, dog )
After insert at end = DList( ape, bat, cat, dog, emu )
After list.insert( 3, 2, fox ) = DList( ape, bat, cat, fox, fox, dog, emu )
```

Ensuring Capacity

An `Array` has a greater control of its underlying storage capacity than any other sequence. An `Array's` internal storage is a native Java array of objects whose default size is 10. When an element is added that causes the internal storage to overflow, an `Array` automatically allocates another native Java array with additional capacity and then copies the contents of the old array into the new array using the fast `System.arraycopy()` method. The old storage is then discarded. This process is repeated when necessary. Note that the current implementation only ever expands the internal storage capacity of an `Array`, and never shrinks it.

Although this process is generally very efficient, you may manually force an `Array` to pre-allocate a specified amount of internal storage by using `ensureCapacity()`. This function is useful if you know in advance the eventual size of a large `Array`. You can find out the current size of an `Array's` internal storage by using the `capacity()` method. To set an `Array's` capacity to the minimum amount of space required to hold its elements, use `trimToSize()`. The next example illustrates the automatic capacity increase and the effect that `ensureCapacity()` and `trimToSize()` have on the amount of internal storage.

Example Sequences4.java

```
import jgl.*;

public class Sequences4
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    System.out.print( "array = " + array + ", size = " + array.size() );
    System.out.println( ", capacity = " + array.capacity() );

    array.insert( 0, 9, "x" );
    System.out.print( "array = " + array + ", size = " + array.size() );
    System.out.println( ", capacity = " + array.capacity() );

    for( int i = 0; i < 2; i++ )
      {
      array.add( "x" ); // Causes reallocation of internal storage.
      System.out.print( "array = " + array + ", size = " + array.size() );
      System.out.println( ", capacity = " + array.capacity() );
      }

    array.ensureCapacity( 1000 ); // Force reallocation of internal storage.
    System.out.print( "array = " + array + ", size = " + array.size() );
    System.out.println( ", capacity = " + array.capacity() );

    array.trimToSize(); // Reduce capacity to the minimum required.
    System.out.print( "array = " + array + ", size = " + array.size() );
    System.out.println( ", capacity = " + array.capacity() );
    }
  }
```

Output

```
array = Array(), size = 0, capacity = 10
array = Array( x, x, x, x, x, x, x, x, x ), size = 9, capacity = 10
array = Array( x, x, x, x, x, x, x, x, x, x ), size = 10, capacity = 10
array = Array( x, x, x, x, x, x, x, x, x, x, x ), size = 11, capacity = 20
array = Array( x, x, x, x, x, x, x, x, x, x, x ), size = 11, capacity = 1000
array = Array( x, x, x, x, x, x, x, x, x, x, x ), size = 11, capacity = 11
```

Constructing From a Native Java array

JGL makes it very easy and efficient to construct an `Array` from a native Java array of objects. If you pass the `Array` constructor an existing array, the `Array` will adopt its storage *without* performing a copy. Operations like `at()`, `put()`, and `replace()` will operate on the original Java array. However, if you add an element using `add()` or `pushBack()`, the original array is copied to an new enlarged storage structure and is then no longer used by the `Array`. You may efficiently copy an `Array's` elements into a native Java array by using `copyTo()`. The following example illustrates these features.

Example Sequences5.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Sequences5
  {
  public static void main( String[] args )
    {
    String strings1[] = { "ape", "bat", "cat" };
    Array array = new Array( strings1 );

    System.out.print( "array = " + array + ", size = " + array.size() );
    System.out.println( ", capacity = " + array.capacity() );

    array.put( 2, "CAT" );
    System.out.println( "array = " + array );
    System.out.print( "Original = " );
    for( int i = 0; i < strings1.length; i++ )
      System.out.print( strings1[ i ] + " " );
    System.out.println();

    array.add( "dog" ); // Forces reallocation of storage.
    System.out.print( "array = " + array + ", size = " + array.size() );
    System.out.println( ", capacity = " + array.capacity() );
    System.out.print( "Original = " );
    for( int i = 0; i < strings1.length; i++ )
      System.out.print( strings1[ i ] + " " );
    System.out.println();

    String string2[] = new String[ array.size() ];
    array.copyTo( string2 ); // Copy contents of array into native array.

    for( int i = 0; i < string2.length; i++ )
      System.out.print( string2[ i ] + " " );
    System.out.println();
    }
  }
```

Output

```
array = Array( ape, bat, cat ), size = 3, capacity = 3
array = Array( ape, bat, CAT )
Original = ape bat CAT
array = Array( ape, bat, CAT, dog ), size = 4, capacity = 6
Original = ape bat CAT
ape bat CAT dog
```

Splicing

The `DList` and `SList` classes support splicing operations that allow you to cut a single node or range of nodes from one list and paste them into another list. The following example uses an `SList` to show splicing in action.

Example Sequences6.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Sequences6
  {
  public static void main( String[] args )
    {
    SList slist1 = new SList();
    slist1.add( "D" );
    slist1.add( "B" );
    SList slist2 = new SList();
    slist2.add( "E" );
    slist2.add( "A" );
    slist2.add( "C" );
    System.out.println( "slist1 = " + slist1 + ", slist2 = " + slist2 );

    // Insert the whole of slist2 in front of the 1st element of slist1.
    slist1.splice( 0, slist2 );
    System.out.println( "slist1 = " + slist1 + ", slist2 = " + slist2 );

    // Move the 1st element to the end of the slist.
    slist1.splice( 5, slist1, 0 );
    System.out.println( "slist1 = " + slist1 + ", slist2 = " + slist2 );

    // Move the 2nd and 3rd elements to be in front of the last element.
    slist1.splice( 4, slist1, 1, 2 );
    System.out.println( "slist1 = " + slist1 + ", slist2 = " + slist2 );
    }
  }
```

Output

```
list1 = SList( D, B ), list2 = SList( E, A, C )
list1 = SList( E, A, C, D, B ), list2 = SList()
list1 = SList( A, C, D, B, E ), list2 = SList()
list1 = SList( A, B, C, D, E ), list2 = SList()
```

Miscellaneous

The `DList` class supports a couple of miscellaneous operations. `unique()` collapses neighboring nodes that contain the same element into a single node, and `reverse()` efficiently reverses the elements of a list. The following example illustrates both of these methods.

Example Sequences7.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Sequences7
  {
  public static void main( String[] args )
    {
    DList list = new DList();
    list.add( "D" );
    list.add( "C" );
    list.add( "C" );
    list.add( "B" );
    list.add( "A" );
    list.add( "A" );
    System.out.println( "list = " + list );

    list.unique();
    System.out.println( "list = " + list );

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

Output

```
list = DList( D, C, C, B, A, A )
list = DList( D, C, B, A )
list = DList( A, B, C, D )
```

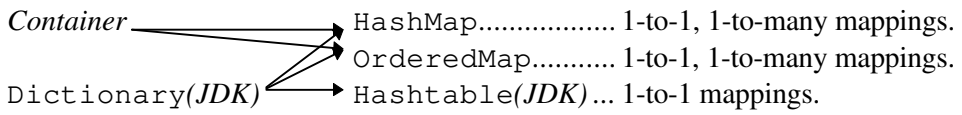
Maps

A map is a data structure that allows you to associate a key with one or more values and then quickly retrieve those values using the key. JGL includes two kinds of map:

- A `HashMap` stores key-value pairs in an internal hash table that does not maintain the pairs in any particular order. However, key-value pairs with matching keys are guaranteed to be adjacent. `HashMap` obtains the hash value of a key by using the standard `hashCode()` method whose default implementation in `Object` returns a number that is unique for each object in the system.

- An `OrderedMap` stores key-value pairs in an internal tree structure. By default, the key-value pairs are ordered based on the hash code of the key, although this criterion can easily be changed. Key-value pairs with matching keys are guaranteed to be adjacent.

Each of these classes implements the `Container` interface and extends the abstract JDK `Dictionary` class. Here is a diagram that illustrates these relationships:



Under normal circumstances, hashing maps are faster than ordered maps. However, there are some occasions when it is preferred that the contents of a map are kept sorted. All classes that extend the `Dictionary` class support the following methods:

- `put` — replace the first value associated with a particular key or add it if none exists
- `get` — return the first value associated with a particular key
- `elements/keys` — return a JDK `Enumeration` over the container's values/keys
- `remove` — remove all the key-value pairs with a particular key

By default, JGL dictionaries do not allow duplicate keys and use the standard `equals()` method for comparing keys and values. Both of these behaviors may be overridden using techniques that are described later in this chapter.

Please note that the most common JGL user error is forgetting to properly define the `hashCode()` function in a user-defined class. See the section "Storing User-Defined Objects" in the "Containers" chapter for more information.

The rest of this chapter describes maps in detail.

Putting and Getting

By default, maps do not allow duplicate keys. If you wish a map to allow duplicates, use a constructor that allows you to specify `true` for the flag `allowDuplicates`.

There are two ways to associate a value with a particular key:

- `add()` — if the key doesn't exist or duplicates are allowed, associate the value with the key and return `null`, otherwise don't modify the map and return the current value associated with the key.

- `put()` — if the key doesn't exist, associate the value with the key and return `null`, otherwise replace the first value associated with the key and return the old value.

To retrieve the first value associated with a key, use `get()`. If no value is associated with the key, `get()` returns `null`.

The first example illustrates these behaviors for a `HashMap` that does not allow duplicates.

Example Maps1.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Maps1
  {
  public static void main( String[] args )
    {
    HashMap map = new HashMap(); // Don't allow duplicates.
    Object value;

    value = map.add( "Dog", "Barky" );
    System.out.println( "value from add = " + value );

    value = map.add( "Cat", "Beauty" );
    System.out.println( "value from add = " + value );
    System.out.println( "map = " + map );

    value = map.add( "Cat", "Agatha" );
    System.out.println( "value from add = " + value );
    System.out.println( "map = " + map );

    value = map.get( "Cat" );
    System.out.println( "Cat name is " + value );

    value = map.get( "Ape" );
    System.out.println( "Ape name is " + value );

    value = map.put( "Cat", "Agatha" );
    System.out.println( "value from put = " + value );
    System.out.println( "map = " + map );

    String name3 = (String) map.get( "Cat" );
    System.out.println( "Cat name is " + name3 );
    }
  }
```

Output

```
value from add = null
value from add = null
map = HashMap( Pair( Cat, Beauty ), Pair( Dog, Barky ) )
value from add = Beauty
map = HashMap( Pair( Cat, Beauty ), Pair( Dog, Barky ) )
Cat name is Beauty
Ape name is null
value from put = Beauty
map = HashMap( Pair( Cat, Agatha ), Pair( Dog, Barky ) )
Cat name is Agatha
```

The next example is the same as **Maps1.java** except that duplicates are allowed.

Example Maps2.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Maps2
  {
  public static void main( String[] args )
    {
    HashMap map = new HashMap( true ); // Allow duplicates.
    Object value;

    value = map.add( "Dog", "Barky" );
    System.out.println( "value from add = " + value );

    value = map.add( "Cat", "Beauty" );
    System.out.println( "value from add = " + value );
    System.out.println( "map = " + map );

    value = map.add( "Cat", "Agatha" );
    System.out.println( "value from add = " + value );
    System.out.println( "map = " + map );

    value = map.get( "Cat" );
    System.out.println( "Cat name is " + value );

    value = map.get( "Ape" );
    System.out.println( "Ape name is " + value );

    value = map.put( "Cat", "Agatha" );
    System.out.println( "value from put = " + value );
    System.out.println( "map = " + map );

    String name3 = (String) map.get( "Cat" );
    System.out.println( "Cat name is " + name3 );
    }
  }
```

Output

```
value from add = null
value from add = null
map = HashMap( Pair( Cat, Beauty ), Pair( Dog, Barky ) )
value from add = null
map = HashMap( Pair( Cat, Beauty ), Pair( Cat, Agatha ), Pair( Dog, Barky ) )
Cat name is Beauty
Ape name is null
value from put = Beauty
map = HashMap( Pair( Cat, Agatha ), Pair( Cat, Agatha ), Pair( Dog, Barky ) )
Cat name is Agatha
```

Enumerating

For compatibility with the JDK, `elements()` returns a `Enumeration` over every value in a map and `keys()` returns a `Enumeration` over every key in a map. There are variations of these functions that enumerate over all keys with a specified value and all values with a particular key. The following example illustrates `elements()` and `keys()`.

Example Maps3.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;
import java.util.Enumeration;

public class Maps3
  {
  public static void main( String[] args )
    {
    HashMap map = new HashMap( true ); // Allow duplicates.
    map.add( new Pair( "Dog", "Barky" ) );
    map.add( new Pair( "Cat", "Beauty" ) );
    map.add( new Pair( "Cat", "Agatha" ) );
    System.out.println( "map = " + map );

    System.out.println( "Enumerator through values..." );
    Enumeration values = map.elements();
    while( values.hasMoreElements() )
      System.out.println( "  " + values.nextElement() );

    System.out.println( "Enumerate through keys..." );
    Enumeration keys = map.keys();
    while( keys.hasMoreElements() )
      System.out.println( "  " + keys.nextElement() );
    }
  }
```

Output

```
map = HashMap( Pair( Cat, Beauty ), Pair( Cat, Agatha ), Pair( Dog, Barky ) )
Enumerator through values...
  Beauty
  Agatha
  Barky
Enumerate through keys...
  Cat
  Cat
  Dog
```

Ordering

By default, `OrderedMap` orders key-value pairs in ascending order based on the hash codes of their keys. For example, when `Integer` objects are stored into an ordered map, they appear in sorted order because the hash code of an `Integer` is equal to its value. You can change the ordering criterion by supplying a function object with the required sorting criterion. A key-value pair *A* is placed to the left of the key-value pair *B* if the function object returns `true` when passed *A*'s key as its first operand and *B*'s key as its second operand. For more information about function objects, consult the "Function Objects" chapter. The following example uses a `GreaterInteger` object to order elements in descending order instead of ascending order.

Example Maps4.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Maps4
  {
  public static void main( String[] args )
    {
    // Order key-value pairs based on the hash value of the key.
    OrderedMap map1 = new OrderedMap();
    map1.put( new Integer( 1 ), "one" );
    map1.put( new Integer( 3 ), "three" );
    map1.put( new Integer( 2 ), "two" );
    System.out.println( "map1 = " + map1 );

    // Order key-value pairs in descending numeric order of their key.
    OrderedMap map2 = new OrderedMap( new GreaterInteger() );
    map2.put( new Integer( 1 ), "one" );
    map2.put( new Integer( 3 ), "three" );
    map2.put( new Integer( 2 ), "two" );
    System.out.println( "map2 = " + map2 );
    }
  }
```

Output

```
map1 = OrderedMap( Pair( 1, one ), Pair( 2, two ), Pair( 3, three ) )
map2 = OrderedMap( Pair( 3, three ), Pair( 2, two ), Pair( 1, one ) )
```

Removing and Counting

All maps include operations for counting and removing all the key-value pairs that have a certain key. The following example illustrates these methods.

Example Maps5.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Maps5
  {
  public static void main( String[] args )
    {
    // Order key-value pairs in descending lexicographical order of keys.
    OrderedMap map = new OrderedMap( new GreaterString(), true ); // Allow
dups.
    map.add( "10", "X" );
    map.add( "10", "ten" );
    map.add( "5", "V" );
    map.add( "5", "five" );

    System.out.println( "map = " + map );
    int n = map.count( "10" );
    System.out.println( "There are " + n + " key-value pairs with key 10" );

    System.out.println( "Removing all occurrences of 10..." );
    map.remove( "10" );

    n = map.count( "10" );
    System.out.println( "There are now " + n + " key-value pairs with key 10"
);
    System.out.println( "map = " + map );
    }
  }
```

Output

```
map = OrderedMap( Pair( 5, V ), Pair( 5, five ), Pair( 10, X ), Pair( 10, ten
))
There are 2 key-value pairs with key 10
Removing all occurrences of 10...
There are now 0 key-value pairs with key 10
map = OrderedMap( Pair( 5, V ), Pair( 5, five ) )
```

Matching Keys

By default, `HashMap` use the standard `equals()` method for matching keys. To override the way that keys are compared, use a constructor that allows you to specify a binary predicate for matching keys. JGL includes several pre-defined binary predicates. For example, `IdenticalTo` uses == to compare objects and therefore only considers two keys to match if they are the *same* object. Consult the "Function Objects" chapter for more information about binary predicates.

The following example illustrates the difference between using `equals()` and == when matching keys.

Example Maps6.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Maps6
  {
  public static void main( String[] args )
    {
    Integer i1 = new Integer( 2 );
    Integer i2 = new Integer( 2 );

    HashMap map1 = new HashMap();
    System.out.println( "Using equals() to compare elements..." );
    System.out.println( "map1.add( i1, two ) = " + map1.add( i1, "two" ) );
    System.out.println( "map1.add( i1, two ) = " + map1.add( i1, "two" ) );
    System.out.println( "map1.add( i2, TWO ) = " + map1.add( i2, "TWO" ) );
    System.out.println( "map1.get( i1 ) = " + map1.get( i1 ) );
    System.out.println( "map1.get( i2 ) = " + map1.get( i2 ) );

    HashMap map2 = new HashMap( new IdenticalTo() );
    System.out.println( "Using == to compare elements..." );
    System.out.println( "map2.add( i1, two ) = " + map2.add( i1, "two" ) );
    System.out.println( "map2.add( i1, two ) = " + map2.add( i1, "two" ) );
    System.out.println( "map2.add( i2, TWO ) = " + map2.add( i2, "TWO" ) );
    System.out.println( "map2.get( i1 ) = " + map2.get( i1 ) );
    System.out.println( "map2.get( i2 ) = " + map2.get( i2 ) );
    }
  }
```

Output

```
Using equals() to compare elements...
map1.add( i1, two ) = null // succeed
map1.add( i1, two ) = two // fail
map1.add( i2, TWO ) = two // fail
map1.get( i1 ) = two
map1.get( i2 ) = two
Using == to compare elements...
map2.add( i1, two ) = null // succeed
map2.add( i1, two ) = two // fail
map2.add( i2, TWO ) = null // succeed
map2.get( i1 ) = two
map2.get( i2 ) = TWO
```

Pairs

A `Pair` is a simple object that binds two objects together. After a `Pair` is constructed with two objects, the objects can be accessed using the public variables `first` and `second`.

For compatibility with the `Container` interface, there is a variation of `add()` that operates just like the version of `add()` that takes a key and a value except that it accepts an `Object` that it casts to a `Pair` whose `first` variable is assumed to be a key and whose `second` variable is assumed to be a value. An `InvalidOperationException` is thrown if you attempt to pass in an object that is not a `Pair`.

The following example illustrates how `Pairs` may be added to a map.

Example Maps7.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Maps7
  {
  public static void main( String[] args )
    {
    Pair pair1 = new Pair( "CAT", "Agatha" );
    Pair pair2 = new Pair( "DOG", "Misty" );
    HashMap map = new HashMap();
    map.add( pair1 );
    map.add( pair2 );
    System.out.println( "map = " + map );
    }
  }
```

Output

```
map = HashMap( Pair( CAT, Agatha ), Pair( DOG, Misty ) )
```

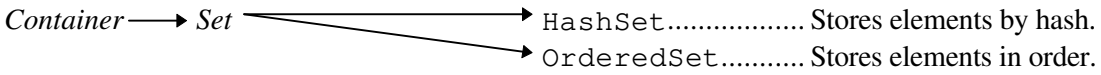
Sets

A set is a data structure that allows you to add objects into it and then quickly test for their presence. JGL includes two kinds of set:

- A `HashSet` stores objects in an internal hash table that does not maintain the objects in any particular order. However, matching objects are guaranteed to be adjacent. `HashSet` obtains the hash value of an object by using the standard `hashCode()` method whose default implementation in `Object` returns a number that is unique for each object in the system.

- An `OrderedSet` stores objects in an internal tree structure. By default, the objects are ordered based on their hash code, although this criterion can easily be changed. Matching objects are guaranteed to be adjacent.

Each of these containers implements the `Set` interface, which in turn extends the `Container` interface. Here is a diagram that illustrates these relationships:



Under normal circumstances, hashing sets are faster than ordered sets. However, there are some occasions when it is preferred that the contents of a set are kept sorted. All classes that extend the `Set` interface support the following methods:

- `put` — replace the first matching object or add it if none exists
- `get` — return the first matching object
- `remove` — remove all matching objects

By default, sets do not allow duplicate objects and use the standard `equals()` method for comparing objects. Both of these behaviors may be overridden using techniques that are described later in this chapter.

Please note that the most common JGL user error is forgetting to properly define the `hashCode()` function in a user-defined class. See the section "Storing User-Defined Objects" in the "Containers" chapter for more information.

The rest of this chapter describes sets in detail.

Putting and Getting

By default, sets do not allow duplicate objects. If you wish a set to allow duplicates, use a constructor that allows you to specify `true` for the flag `allowDuplicates`.

There are two ways to add an object to a set:

- `add()` — if the object doesn't exist or duplicates are allowed, add the object and return `null`, otherwise don't modify the set and return the matching object.

- `put()` — if the object doesn't exist, add the object and return `null`, otherwise replace the first object that matches and return the old object.

To retrieve the first object that matches a specified object, use `get()`. If no value is associated with the object, `get()` returns `null`.

The first example illustrates these behaviors for a `HashSet` that does not allow duplicates. It uses an auxiliary `Widget` class which defines its `equals()` method to return `true` if the names of two widgets are the same.

Widget.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
public class Widget
  {
  public String myName; // public for demo only.
  public int myPrice; // public for demo only.

  public Widget( String name, int price )
    {
    myName = name;
    myPrice = price;
    }

  public int hashCode()
    {
    return myName.hashCode();
    }

  public boolean equals( Object object )
    {
    return object instanceof Widget &&
           myName.equals( ((Widget) object).myName );
    }

  public String toString()
    {
    return "Widget( " + myName + ", " + myPrice + " )";
    }
  }
```

Example Sets1.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;
import Widget;
import java.util.Enumeration;

public class Sets1
  {
  public static void main( String[] args )
    {
    HashSet set = new HashSet(); // Don't allow duplicates.
    Object value;

    value = set.add( new Widget( "button", 100 ) );
    System.out.println( "value from add = " + value );
    value = set.add( new Widget( "menu", 200 ) );
    System.out.println( "value from add = " + value );
    System.out.println( "set = " + set );

    value = set.add( new Widget( "button", 300 ) );
    System.out.println( "value from add = " + value );
    System.out.println( "set = " + set );

    value = set.put( new Widget( "button", 300 ) );
    System.out.println( "value from put = " + value );
    System.out.println( "set = " + set );
    }
  }
```

Output

```
value from add = null
value from add = null
set = HashSet( Widget( button, 100 ), Widget( menu, 200 ) )
value from add = Widget( button, 100 )
set = HashSet( Widget( button, 100 ), Widget( menu, 200 ) )
value from add = Widget( button, 100 )
set = HashSet( Widget( button, 300 ), Widget( menu, 200 ) )
```

The next example is the same as **Sets1.java** except that duplicates are allowed.

Example Sets2.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;
import Widget;
import java.util.Enumeration;

public class Sets2
  {
  public static void main( String[] args )
    {
    HashSet set = new HashSet( true ); // Allow duplicates.
    Object value;

    value = set.add( new Widget( "button", 100 ) );
    System.out.println( "value from add = " + value );
    value = set.add( new Widget( "menu", 200 ) );
    System.out.println( "value from add = " + value );
    System.out.println( "set = " + set );

    value = set.add( new Widget( "button", 300 ) );
    System.out.println( "value from add = " + value );
    System.out.println( "set = " + set );

    value = set.put( new Widget( "button", 300 ) );
    System.out.println( "value from put = " + value );
    System.out.println( "set = " + set );
    }
  }
```

Output

```
value from add = null
value from add = null
set = HashSet( Widget( button, 100 ), Widget( menu, 200 ) )
value from add = null
set = HashSet(Widget( button, 100 ), Widget( button, 300 ), Widget( menu, 200
))
value from put = Widget( button, 100 )
set = HashSet(Widget( button, 300 ), Widget( button, 300 ), Widget( menu, 200
))
```

Ordering

By default, `OrderedSet` orders objects in ascending order based on their hash codes. For example, when `Integer` objects are stored into an ordered set, they appear in sorted order because the hash code of an `Integer` is equal to its value. You can change the ordering criterion by supplying a function object with the required sorting criterion. An object *A* is placed to the left of object *B* if the function object returns `true` when passed *A* as its first operand and *B* as its second operand. For more information about function objects, consult the "Function Objects" chapter. The following example uses a `GreaterInteger` object to order elements in descending order instead of ascending order.

Example Sets3.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Sets3
  {
  public static void main( String[] args )
    {
    // Order elements based on their hash value.
    OrderedSet set1 = new OrderedSet();
    set1.add( new Integer( 1 ) );
    set1.add( new Integer( 3 ) );
    set1.add( new Integer( 2 ) );
    System.out.println( "set1 = " + set1 );

    // Order elements in descending numeric order.
    OrderedSet set2 = new OrderedSet( new GreaterInteger() );
    set2.add( new Integer( 1 ) );
    set2.add( new Integer( 3 ) );
    set2.add( new Integer( 2 ) );
    System.out.println( "set2 = " + set2 );
    }
  }
```

Output

```
set1 = OrderedSet( 1, 2, 3 )
set2 = OrderedSet( 3, 2, 1 )
```

Removing and Counting

All sets include operations for counting and removing all the objects that match a particular object. The following example illustrates these methods.

Example Sets4.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Sets4
  {
  public static void main( String[] args )
    {
    // Order objects in descending lexicographical order
    // Note that "1" comes before "5".
    OrderedSet set = new OrderedSet( new GreaterString(), true ); // Allow
dups.
    set.add( "10" );
    set.add( "10" );
    set.add( "5" );
    set.add( "5" );

    System.out.println( "set = " + set );
    int n = set.count( "10" );
    System.out.println( "There are " + n + " objects that match 10" );

    System.out.println( "Removing all occurrences of 10..." );
    set.remove( "10" );

    n = set.count( "10" );
    System.out.println( "There are now " + n + " objects that match 10" );
    System.out.println( "set = " + set );
    }
  }
```

Output

```
set = OrderedSet( 5, 5, 10, 10 )
There are 2 objects that match 10
Removing all occurrences of 10...
There are now 0 objects that match 10
set = OrderedSet( 5, 5 )
```

Matching Objects

By default, `HashSet` use the standard `equals()` method for matching objects. To override the way that objects are compared, use a constructor that allows you to specify a binary predicate for matching objects. JGL includes several pre-defined binary predicates. For example, `IdenticalTo` uses == to compare objects and therefore only considers two objects to match if they are the *same* object. Consult the "Function Objects" chapter for more information about binary predicates.

The following example illustrates the difference between using `equals()` and == when matching objects.

Example Sets5.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Sets5
  {
  public static void main( String[] args )
    {
    Integer i1 = new Integer( 2 );
    Integer i2 = new Integer( 2 );

    HashSet set1 = new HashSet();
    System.out.println( "Using equals() to compare elements..." );
    System.out.println( "set1.add( i1 ) = " + set1.add( i1 ) );
    System.out.println( "set1.add( i1 ) = " + set1.add( i1 ) );
    System.out.println( "set1.add( i2 ) = " + set1.add( i2 ) );
    System.out.println( "set1.get( i1 ) = " + set1.get( i1 ) );
    System.out.println( "set1.get( i2 ) = " + set1.get( i2 ) );

    HashSet set2 = new HashSet( new IdenticalTo() );
    System.out.println( "Using == to compare elements..." );
    System.out.println( "set2.add( i1 ) = " + set2.add( i1 ) );
    System.out.println( "set2.add( i1 ) = " + set2.add( i1 ) );
    System.out.println( "set2.add( i2 ) = " + set2.add( i2 ) );
    System.out.println( "set2.get( i1 ) = " + set2.get( i1 ) );
    System.out.println( "set2.get( i2 ) = " + set2.get( i2 ) );
    }
  }
```

Output

```
Using equals() to compare elements...
set1.add( i1 ) = null // succeed
set1.add( i1 ) = 2 // fail
set1.add( i2 ) = 2 // fail
set1.get( i1 ) = 2
set1.get( i2 ) = 2
Using == to compare elements...
set2.add( i1 ) = null // succeed
set2.add( i1 ) = 2 // fail
set2.add( i2 ) = null // succeed
set2.get( i1 ) = 2
set2.get( i2 ) = 2
```

Union, Intersection, and other Set Operations

The `HashSet` and `OrderedSet` classes include a useful collection of set operations, including `union()`, `intersection()`, and `difference()`. Note that these methods return their result in a new set and do not affect the contents of the receiver or the argument. All of the set operations are illustrated by the following example.

Example Sets6.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Sets6
  {
  public static void main( String[] args )
    {
    HashSet set1 = new HashSet();
    set1.insert( "ape" );
    set1.insert( "cat" );
    set1.insert( "bat" );

    HashSet set2 = new HashSet();
    set2.insert( "bat" );
    set2.insert( "fox" );
    set2.insert( "ape" );
    System.out.println( "set1 = " + set1 + ", set2 = " + set2 );

    HashSet set3 = set1.union( set2 );
    System.out.println( "set3 = set1.union( set2 ) = " + set3 );

    HashSet set4 = set1.intersection( set2 );
    System.out.println( "set4 = set1.intersection( set2 ) = " + set4 );

    HashSet set5 = set1.difference( set2 );
    System.out.println( "set5 = set1.difference( set2 ) = " + set5 );

    HashSet set6 = set1.symmetricDifference( set2 );
    System.out.println( "set6 = set1.symmetricDifference( set2 ) = " + set6 );

    System.out.println( "set4.subsetOf( set3 ) = " + set4.subsetOf( set3 ) );
    System.out.println( "set3.subsetOf( set4 ) = " + set3.subsetOf( set4 ) );
    }
  }
```

Output

```
set1 = HashSet( ape, bat, cat ), set2 = HashSet( ape, bat, fox )
set3 = set1.union( set2 ) = HashSet( ape, bat, fox, cat )
set4 = set1.intersection( set2 ) = HashSet( ape, bat )
set5 = set1.difference( set2 ) = HashSet( cat )
set6 = set1.symmetricDifference( set2 ) = HashSet( fox, cat )
set4.subsetOf( set3 ) = true
set3.subsetOf( set4 ) = false
```

Queues and Stacks

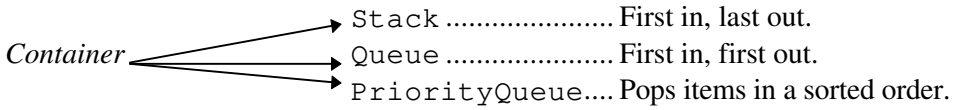
Queues and stacks are containers whose main methods for access are `push()` and `pop()`. JGL includes three containers with this property.

- A `Stack` is a first-in, last-out data structure that pops elements in the opposite order than they were pushed. By default, a `Stack` uses an `Array` for its internal storage, although this can easily be changed.

- A `Queue` is a first-in, first-out data structure that pops elements in the same order than they were pushed. By default, a `Queue` uses an `SList` for its internal storage, although this can easily be changed.

- A `PriorityQueue` is a form of queue that keeps its elements in a quasi-sorted state and pops them based on their sorted order rather than in the order that they were pushed. A `PriorityQueue` uses an `Array` for its underlying storage.

Each of these containers implement the `Container` interface, as shown by the following diagram:



The rest of this chapter describes each of these container types in detail.

Stack

In addition to implementing the methods defined in the `Container` interface, a `Stack` supports the `push()` the `pop()` methods as shown by the following example.

Example Stacks1.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import java.util.Enumeration;
import jgl.*;

public class Stacks1
  {
  public static void main( String[] args )
    {
    Stack stack = new Stack();
    stack.push( "bat" );
    stack.push( "cat" );
    stack.push( "dog" );
    System.out.println( "stack = " + stack );
    System.out.println();

    System.out.println( "Non-destructively enumerate the Stack." );
    Enumeration e = stack.elements();
    while( e.hasMoreElements() )
      System.out.println( e.nextElement() );
    System.out.println();

    System.out.println( "Pop and print each element." );
    while( !stack.isEmpty() )
      System.out.println( stack.pop() );
    }
  }
```

Output

```
stack = Stack( Array( bat, cat, dog ) )

Non-destructively enumerate the Stack.
bat
cat
dog

Pop and print each element.
dog
cat
bat
```

By default, a `Stack` uses an `Array` for its underlying storage. To change this to a different container, supply the container when the `Stack` is constructed.

Example Stacks2.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Stacks2
  {
  public static void main( String[] args )
    {
    // Use a DList as the underlying data structure.
    Stack stack = new Stack( new DList () );
    stack.push( "bat" );
    stack.push( "cat" );
    stack.push( "dog" );

    System.out.println( "Print the Stack." );
    System.out.println( stack );
    }
  }
```

Output

```
Print the Stack.
Stack( DList( bat, cat, dog ) )
```

Queue

In addition to implementing the methods defined in the `Container` interface, a `Queue` supports the `push()` and `pop()` methods as shown by the following example.

Example Stacks3.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import java.util.Enumeration;
import jgl.*;

public class Stacks3
  {
  public static void main( String[] args )
    {
    Queue queue = new Queue();
    queue.push( "bat" );
    queue.push( "cat" );
    queue.push( "dog" );

    System.out.println( "queue = " + queue );
    System.out.println();

    System.out.println( "Non-destructively enumerate the Queue." );
    Enumeration e = queue.elements();
    while( e.hasMoreElements() )
      System.out.println( e.nextElement() );
    System.out.println();

    System.out.println( "Pop and print each element." );
    while( !queue.isEmpty() )
      System.out.println( queue.pop() );
    }
  }
```

Output

```
Print the Queue.
Queue( SList( bat, cat, dog ) )

Non-destructively enumerate the Queue.
bat
cat
dog

Pop and print each element.
bat
cat
dog
```

By default, a `Queue` uses an `SList` for its underlying storage. To change this to a different container, supply the container when the `Queue` is constructed.

PriorityQueue

A `PriorityQueue` uses an internal `Array` to store pushed elements in a heap structure. By default, elements are compared using their hash values, although you can change this behavior by supplying a function object when the queue is constructed. Note that although a `PriorityQueue` always pops elements in the order determined by the comparator, this does not mean that that objects are stored in order within the internal heap structure. The following example uses a `PriorityQueue` to push and pop `Integer` objects in descending order.

Example Stacks4.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import java.util.Enumeration;
import jgl.*;

public class Stacks4
  {
  public static void main( String[] args )
    {
    // Use a HashComparator for comparing elements. Since the hash value of an
    // Integer is its int value, this will order Integers in descending order.
    PriorityQueue queue = new PriorityQueue();
    queue.push( new Integer( 5 ) );
    queue.push( new Integer( -2 ) );
    queue.push( new Integer( 10 ) );
    queue.push( new Integer( 6 ) );
    queue.push( new Integer( 20 ) );
    queue.push( new Integer( -10 ) );

    System.out.println( "queue = " + queue );
    System.out.println();

    System.out.println( "Non-destructively enumerate the PriorityQueue." );
    Enumeration e = queue.elements();
    while( e.hasMoreElements() )
      System.out.print( e.nextElement() + " " );
    System.out.println();

    System.out.println( "Pop and print each element." );
    while( !queue.isEmpty() )
      System.out.print( queue.top() + " " );
    System.out.println();
    }
  }
```

Output

```
queue = PriorityQueue( Array( 20, 10, 5, -2, 6, -10 ) )

Non-destructively enumerate the PriorityQueue.
20 10 5 -2 6 -10
Pop and print each element.
20 10 6 5 -2 -10
```

The following example uses a `GreaterString` function object to pop `String` objects in ascending order.

Example Stacks5.java

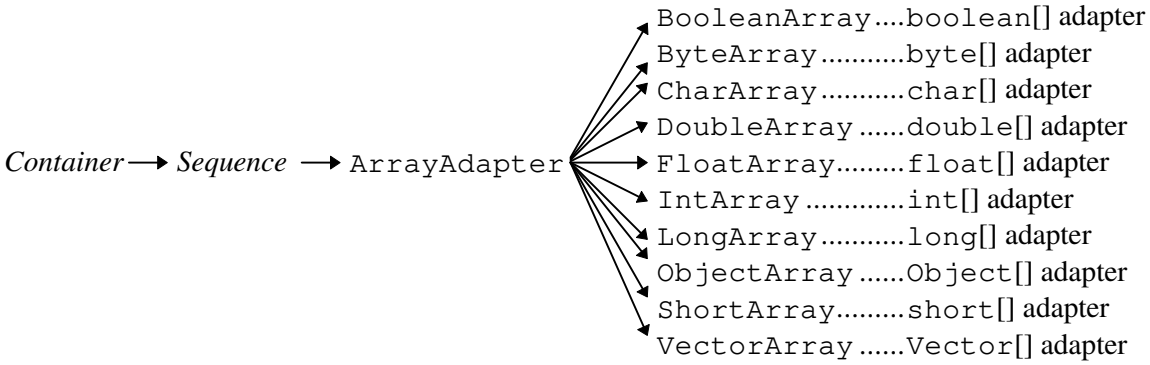
```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Stacks5
  {
  public static void main( String[] args )
    {
    // Use a GreaterString function object for comparing elements. This
    // will order strings in ascending order.
    PriorityQueue queue = new PriorityQueue( new GreaterString() );
    queue.push( "cat" );
    queue.push( "dog" );
    queue.push( "ape" );
    queue.push( "bat" );
    queue.push( "fox" );
    queue.push( "emu" );

    System.out.println( "Pop and print each element." );
    while( !queue.isEmpty() )
      System.out.print( queue.pop() + " ");
    System.out.println();
    }
  }
```

Output

```
Pop and print each element.
ape bat cat dog emu fox
```

Array Adapters

One of the primary strengths of JGL is the ability to apply its generic algorithms to JGL containers, native Java arrays, and JDK data structures. The way that it does this is to define the algorithms to accept data structures that implement the `Container` interface and then to supply special adapter classes that allow native Java arrays and JDK data structures to act as if they were a JGL `Container`. Here is a diagram that shows all of the JGL array adapters:



This chapter describes the adapter classes for native Java arrays and JDK containers.

Using a Native Java Array as a Container

There is a separate adapter class for each kind of native Java array. When you enumerate a native Java array of primitives using the adapter class, each primitive is automatically retrieved as its closest object equivalent as defined in `java.lang`. Here is a table that shows this information for each type:

Native Array Type	Adapter Class	Retrieved As
`boolean[]`	`BooleanArray`	`Boolean`
`byte[]`	`ByteArray`	`Integer`
`char[]`	`CharArray`	`Character`
`short[]`	`ShortArray`	`Integer`
`int[]`	`IntArray`	`Integer`
`long[]`	`LongArray`	`Long`
`float[]`	`FloatArray`	`Float`
`double[]`	`DoubleArray`	`Double`
`Object[]`	`ObjectArray`	`N/A`

The adapter classes implement all of the methods defined by the `Sequence` interface. Because the size of a native Java array is fixed, the `add()`, `clear()`, `pushBack()`, `popBack()`, `pushFront()`, `popFront()`, and `remove()` methods will always throw an `InvalidOperationException`. The following example uses an `IntArray` adapter class to conveniently print, sort, and shuffle a native Java array of `int`s.

Example Adapters1.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Adapters1
  {
  public static void main( String[] args )
    {
    int ints[] = { 3, -1, 2, -3, 4 };
    IntArray intArray = new IntArray( ints );
    System.out.println( "Unsorted native int array = " + intArray );

    Sorting.sort( intArray );
    System.out.println( "Sorted = " + intArray );

    Shuffling.randomShuffle( intArray );
    System.out.println( "Randomized = " + intArray );

    for( int i = 0; i < ints.length; i++ )
      System.out.println( ints[ i ] );
    }
  }
```

Output

```
Unsorted native int array = int[]( 3, -1, 2, -3, 4 )
Sorted = int[]( -3, -1, 2, 3, 4 )
Randomized = int[]( -1, 2, 3, 4, -3 )
-1 2 3 4 -3
```

Using a JDK Vector as a Container

The `VectorArray` adapter class allows a JDK `Vector` to act as if it were a JGL `Container`. The `add()`, `clear()`, `pushFront()`, `pushback()`, `popFront()`, `popBack()`, and `remove()` methods work correctly, and are automatically converted into their JDK equivalents. The following example uses a `VectorArray` to sort and shuffle a JDK Vector.

Example Adapters2.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;
import java.util.Vector;

public class Adapters2
  {
  public static void main( String[] args )
    {
    Vector vector = new Vector();
    vector.addElement( new Integer( 3 ) );
    vector.addElement( new Integer( -1 ) );
    vector.addElement( new Integer( 2 ) );
    vector.addElement( new Integer( -3 ) );
    vector.addElement( new Integer( 4 ) );

    VectorArray vectorArray = new VectorArray( vector );
    System.out.println( "Unsorted JDK Vector = " + vectorArray );

    Sorting.sort( vectorArray );
    System.out.println( "Sorted = " + vectorArray );

    Shuffling.randomShuffle( vectorArray );
    System.out.println( "Randomized = " + vectorArray );

    // Show that the original JDK Vector was altered.
    System.out.println( "JDK vector = " + vector );
    }
  }
```

Output

```
Unsorted JDK Vector = [3, -1, 2, -3, 4]
Sorted = [-3, -1, 2, 3, 4]
Randomized = [4, 3, -3, 2, -1]
JDK vector = [4, 3, -3, 2, -1]
```

Algorithms

One of JGL's primary strengths is its comprehensive set of reusable algorithms that can be applied to a wide variety of data structures, including JGL containers, JDK containers, and native Java arrays. Because JGL algorithms are stand-alone and are not part of any JGL container, you can add new algorithms to JGL without changing any of the existing code. We hope that users of JGL will send new algorithms to ObjectSpace so that they can be incorporated into subsequent releases of JGL. Significant contributions will always be acknowledged in our documentation.

All JGL algorithms are categorized and packaged as public static methods of a class that represents the category. For example, the `remove(), removeIf(), removeCopy(),` and `removeCopyIf()` algorithms are all public static methods of the `Removing` class.

The next section contains a list of all the JGL algorithm classes and methods. Subsequent chapters contain examples of several common algorithms at work.

List of Algorithms by Category

This section lists all of the JGL algorithms by class, together with a brief synopsis. Each algorithm is implemented as a public static method of its associated class - you cannot create *instances* of any algorithm class. For detailed information about a particular algorithm, consult the online class catalog.

Applying
- `forEach` . . . . . apply a function to every item in a range
- `inject` . . . . . iteratively apply a binary function to every item in a range

Comparing
- `equal` . . . . . check that two sequences match
- `lexicographicalCompare` . . lexicographically compare two sequences
- `median` . . . . . return the median of three values
- `mismatch` . . . . . search two sequences for a mismatched item.

Copying
- `copy` . . . . . copy a range of items to another area
- `copyBackward` . . . . . copy a range of items backwards to another area

Counting
- `count` . . . . . count items in a range that match a value
- `countIf` . . . . . count items in a range that satisfy a predicate

Filling
- `fill` . . . . . set every item in a range to a particular value
- `fillN` . . . . . set n items to a particular value

Filtering
- `reject` . . . . . return all values that do not satisfy a predicate
- `select` . . . . . return all values that satisfy a predicate
- `unique` . . . . . collapse all consecutive values in a sequence
- `uniqueCopy` . . . . . copy a sequence, collapsing consecutive values

Finding
- `adjacentFind` . . . . . locate consecutive sequence in a range
- `detect` . . . . . return first item that satisfies a predicate
- `every` . . . . . return true if every item in a range satisfies a predicate
- `find` . . . . . locate an item in a sequence
- `findIf` . . . . . locate an item that satisfies a predicate in a range
- `some` . . . . . return true if at least one item in a range satisfies a predicate

Heap
- `makeHeap` . . . . . make a sequence into a heap
- `popHeap` . . . . . pop the top value from a heap
- `pushHeap` . . . . . place the last element into a heap
- `sortHeap` . . . . . sort a heap

MinMax
- `minElement` . . . . . return the minimum item within a range.
- `maxElement` . . . . . return the maximum item within a range.

Permuting
- `nextPermutation` . . . . . change sequence to next lexicographic permutation
- `prevPermutation` . . . . . change sequence to last lexicographic permutation

Printing
- `print` . . . . . prints a sequence to standard output
- `println` . . . . . prints a sequence to standard output followed by a newline
- `toString` . . . . . returns a description of a sequence

Removing
- `remove` . . . . . remove all matching items from a sequence
- `removeCopy` . . . . . copy a sequence, removing all matching items
- `removeCopyIf` . . . . . copy sequence, removing all that satisfy predicate
- `removeIf` . . . . . remove items that satisfy predicate from sequence

Replacing
- `replace` . . . . . replace specified value in a sequence with another
- `replaceCopy` . . . . . copy sequence, replacing matching values
- `replaceCopyIf` . . . . . copy sequence, replacing values that satisfy predicate
- `replaceIf` . . . . . replace specified values that satisfy a predicate

Reversing
- `reverse` . . . . . reverse the items in a sequence
- `reverseCopy` . . . . . create a reversed copy of a sequence

Rotating
- `rotate` . . . . . rotate a sequence by n positions
- `rotateCopy` . . . . . copy a sequence, rotating it by n positions

SetOperations
- `includes` . . . . . search for one sequence in another sequence
- `setDifference` . . . . . create set of elements in 1st sequence that are not in 2nd
- `setIntersection` . . . . . create set of elements that are in both sequences
- `setSymmetricDifference` . . create set of elements that not in both sequences
- `setUnion` . . . . . create set of elements that are in either sequence

Shuffling
- `randomShuffle` . . . . . randomize sequence using random shuffles

Sorting
- `sort` . . . . . sort a sequence

Swapping
- `iterSwap` . . . . . swap the values indicated by two iterators
- `swapRanges` . . . . . swap two ranges of items

Transforming
- `collect` . . . . . return result of applying a function to every item in a range
- `transform` . . . . . transform one sequence into another

Applying

The `Applying` class has a couple of algorithms for applying a function to every element of a container. `forEach()` applies a unary function to every element of a container. `inject()` calculates an initial result by calling a binary function with the initial value as the first parameter and the first element as the second parameter. It then applies the function to the remaining elements, using the previous result as the first parameter and the next element as the second parameter. The last result is then returned. For more information about unary and binary functions, refer to the "Function Objects" chapter. The following example illustrates both of the applying algorithms.

Example Algorithms1.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Algorithms1
  {
  public static void main( String[] args )
    {
    Array array1 = new Array();
    array1.add( "cat" );
    array1.add( "monkey" );
    array1.add( "goat" );
    Applying.forEach( array1, new PrintFunction() );

    SList list = new SList();
    list.add( new Integer( 3 ) );
    list.add( new Integer( 7 ) );
    list.add( new Integer( 4 ) );
    Integer total = (Integer)
      Applying.inject( list, new Integer( 0 ), new PlusInteger() );
    System.out.println( "list = " + list + ", total = " + total );
    }
  }

class PrintFunction implements UnaryFunction
  {
  public Object execute( Object object )
    {
    System.out.println( "PRINT " + object );
    return null; // Not used.
    }
  }
```

Output

```
PRINT cat
PRINT monkey
PRINT goat
list = SList( 3, 7, 4 ), total = 14
```

Copying

The `Copying` class contains a variety of copying algorithms for adding the contents of one container to another container. Behind the scenes, the algorithms use the destination's `add()` method to perform the copy. The following example uses container adapters to copy the contents of a native Java array of `ints` into a JDK `Vector`.

Example Algorithms2.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;
import java.util.Vector;

public class Algorithms2
  {
  public static void main( String[] args )
    {
    int ints[] = { 2, 6, 3, 7 };
    Vector vector = new Vector();
    vector.addElement( new Integer( 1 ) );
    vector.addElement( new Integer( 4 ) );

    // Create container adapters.
    IntArray intArray = new IntArray( ints );
    VectorArray vectorArray = new VectorArray( vector );

    System.out.println( "vector before copying = " + vector );
    Copying.copy( intArray, vectorArray );
    System.out.println( "vector after copying = " + vector );
    }
  }
```

Output

```
vector before copying = [1, 4]
vector after copying = [1, 4, 2, 6, 3, 7]
```

Counting

The `Counting` class has several algorithms for counting all the elements in a container that either match a particular value or satisfy a specified unary predicate. For more information about predicates, refer to the "Function Objects" chapter. The following example illustrates both varieties of the counting algorithm.

Example Algorithms3.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Algorithms3
  {
  public static void main( String[] args )
    {
    SList list = new SList();
    list.add( new Integer( -1 ) );
    list.add( new Integer( 1 ) );
    list.add( new Integer( -2 ) );
    list.add( new Integer( 1 ) );
    list.add( new Integer( -3 ) );
    System.out.println( "list = " + list );

    Object value = new Integer( 1 );
    // Count all objects that are equal to 1.
    int n1 = Counting.count( list, value );
    System.out.println( "Occurences of " + value + " = " + n1 );

    // Count all objects that are negative.
    int n2 = Counting.countIf( list, new NegativeInteger() );
    System.out.println( "Occurences of a negative = " + n2 );
    }
  }
```

Output

```
list = SList( -1, 1, -2, 1, -3 )
Occurences of 1 = 2
Occurences of a negative = 3
```

Finding

The `Finding` class has several algorithms for locating elements in a container that either match a particular value or satisfy a specified unary predicate. For example, `detect()` returns the first object that satisfies a unary predicate, and `some()` returns `true` if at least one object in the container satisfies a unary predicate. For more information about predicates, refer to the “Function Objects” chapter. The following example illustrates two of the finding algorithm.

Example Algorithms4.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Algorithms4
  {
  public static void main( String[] args )
    {
    int ints[] = { 3, 7, 8, 2, -5, 8, 9, -2 };
    IntArray array = new IntArray( ints );
    System.out.println( "array = " + array );

    Integer negative = (Integer) Finding.detect( array, new NegativeInteger()
);
    System.out.println( "first negative = " + negative );

    boolean some = Finding.some( array, new NegativeInteger() );
    System.out.println( "some items are negative = " + some );
    }
  }
```

Output

```
array = int[]( 3, 7, 8, 2, -5, 8, 9, -2 )
first negative = -5
some items are negative = true
```

Filtering

The `Filtering` class has several algorithms for creating a copy of a sequence whilst performing a filtering operation. For example, `select()` and `reject()` return a copy of a sequence that only contains the elements that do/don't satisfy a predicate, respectively. For more information about predicates, refer to the “Function Objects” chapter. The following example illustrates a couple of the filtering algorithms.

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Algorithms5
  {
  public static void main( String[] args )
    {
    Array array1 = new Array();
    array1.add( "cat" );
    array1.add( "monkey" );
    array1.add( "goat" );
    array1.add( "elephant" );
    System.out.println( "array1 = " + array1 );

    // Predicate that returns true if a string is greater
    // than 4 characters long.
    UnaryPredicate predicate =
      new UnaryComposePredicate( new BindSecondPredicate( new
GreaterInteger(),
      new Integer( 4 ) ), new LengthString() );

    Array array2 = (Array) Filtering.select( array1, predicate );
    System.out.println( "strings with length > 4 = " + array2 );

    Array array3 = (Array) Filtering.reject( array1, predicate );
    System.out.println( "strings with length <= 4 = " + array3 );
    }
  }
```

Output

```
array1 = Array( cat, monkey, goat, elephant )
strings with length > 4 = Array( monkey, elephant )
strings with length <= 4 = Array( cat, goat )
```

Replacing

The `Replacing` class contains generic algorithms for replacing objects that have a particular value or that satisfy a particular unary predicate. There are versions of the algorithm that modify the source container and others that copy the result into a separate container. The following example demonstrates two of these variations.

Example Algorithms6.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Algorithms6
  {
  public static void main( String[] args )
    {
    DList list = new DList();
    list.add( new Integer( -1 ) );
    list.add( new Integer( 1 ) );
    list.add( new Integer( -2 ) );
    list.add( new Integer( 1 ) );
    list.add( new Integer( -3 ) );
    System.out.println( "list = " + list );

    Object oldValue = new Integer( 1 );
    Object newValue = new Integer( 4 );
    int n1 = Replacing.replace( list, oldValue, newValue );
    System.out.println( "after 1 -> 4, list = " + list );

    Array array = new Array();
    UnaryPredicate predicate = new NegativeInteger();
    newValue = new Integer( 0 );
    Replacing.replaceCopyIf( list, array, predicate, newValue );
    System.out.println( "list = " + list );
    System.out.println( "array = " + array );
    }
  }
```

Output

```
list = DList( -1, 1, -2, 1, -3 )
after 1 -> 4, list = DList( -1, 4, -2, 4, -3 )
list = DList( -1, 4, -2, 4, -3 )
array = Array( 0, 4, 0, 4, 0 )
```

Reversing

The `Reversing` class contains algorithms for reversing the contents of a container, as demonstrated by the following example.

Example Algorithms7.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Algorithms7
  {
  public static void main( String[] args )
    {
    Deque deque = new Deque();
    deque.add( "Batman" );
    deque.add( "Superman" );
    deque.add( "Phantom" );
    deque.add( "Spiderman" );
    System.out.println( "before reverse = " + deque );
    Reversing.reverse( deque );
    System.out.println( "after reverse = " + deque );
    }
  }
```

Output

```
before reverse = Deque( Batman, Superman, Phantom, Spiderman )
after reverse = Deque( Spiderman, Phantom, Superman, Batman )
```

Sorting

The `Sorting` class contains a highly efficient quicksort-based algorithm for sorting any `Sequence`. By default, elements are sorted in increasing order of hash value. This default behavior works well for sorting `Integer` objects, since their hash value is equal to their actual integer value. To change this default behavior, supply the sort algorithm with a binary predicate object. An object *A* will be placed to the left of an object *B* if the predicate object returns `true` when executed with *A* as the first operand and *B* as the second operand. For more information about predicates, refer to the chapter called "Function Objects". The following example illustrates both uses of the sorting algorithm.

Example Algorithms8.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Algorithms8
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( new Integer( 3 ) );
    array.add( new Integer( -2 ) );
    array.add( new Integer( 4 ) );
    array.add( new Integer( -5 ) );
    System.out.println( "unsorted array = " + array );
    Sorting.sort( array );
    System.out.println( "sorted array = " + array );

    Deque deque = new Deque();
    deque.add( "triangle" );
    deque.add( "square" );
    deque.add( "pentagon" );
    deque.add( "hexagon" );
    System.out.println( "unsorted deque = " + deque );
    Sorting.sort( deque, new LessString() );
    System.out.println( "sorted deque = " + deque );
    }
  }
```

Output

```
unsorted array = Array( 3, -2, 4, -5 )
sorted array = Array( -5, -2, 3, 4 )
unsorted deque = Deque( triangle, square, pentagon, hexagon )
sorted deque = Deque( hexagon, pentagon, square, triangle )
```

Transforming

The `Transforming` class contains a couple of algorithms for replacing every element of a container with a transformed version of itself. The transformation is specified with a unary function object. It also contains an algorithm for applying a binary function in a pair-wise fashion to every element in two containers and storing the results into a third container. For more information about function objects, consult the "Function Objects" chapter. The following example illustrates these algorithms.

Example Algorithms9.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Algorithms9
  {
  public static void main( String[] args )
    {
    int ints1[] = { 1, 3, 5, 2 };
    Array array = new Array();
    IntArray intArray1 = new IntArray( ints1 );
    UnaryFunction function = new NegateInteger();
    Transforming.transform( intArray1, array, function );
    System.out.println( "ints1 = " + intArray1 );
    System.out.println( "array = " + array );
    System.out.println();

    int ints2[] = { 2, 4, 2, 3 };
    int ints3[] = { 3, 6, 2, 1 };
    SList list = new SList();
    IntArray intArray2 = new IntArray( ints2 );
    IntArray intArray3 = new IntArray( ints3 );
    BinaryFunction function2 = new TimesInteger();
    Transforming.transform( intArray2, intArray3, list, function2 );
    System.out.println( "ints2 = " + intArray2 );
    System.out.println( "ints3 = " + intArray3 );
    System.out.println( "list = " + list );

    Array array1 = new Array();
    array1.add( "cat" );
    array1.add( "monkey" );
    array1.add( "goat" );
    System.out.println( "array1 = " + array1 );
    Array array2 = (Array) Transforming.collect( array1, new LengthString() );
    System.out.println( "array2 = " + array2 );
    }
  }
```

Output

```
ints1 = int[]( 1, 3, 5, 2 )
array = Array( -1, -3, -5, -2 )

ints2 = int[]( 2, 4, 2, 3 )
ints3 = int[]( 3, 6, 2, 1 )
list = SList( 6, 24, 4, 3 )

array1 = Array( cat, monkey, goat )
array2 = Array( 3, 6, 4 )
```

Applying an Algorithm to a Sub-Range of a Container

All of the algorithms examples in this section have applied an algorithm to an entire container. If you wish you apply an algorithm to a sub-range of a container, you must take advantage of the JGL iterators. These are described fully in the “Iterators” chapter.

Function Objects

Many of the JGL algorithms and containers require you to specify a *function object* (sometimes known as a *functor*) to perform their operation. A function object can have instance variables and may be created and stored just like any other kind of object. A function object that processes one parameter is called a *unary* function, whereas a function that processes two parameters is called a *binary* function. There are two main types of function object:

- *Predicates* always return a `boolean`, and are used for ordering elements or triggering actions.
- *General functions* perform an arbitrary operation on one or more arguments and return an `Object`.

The rest of this chapter describes predicates and general functions in more detail.

Predicates

Predicates are function objects that are used to trigger actions or order elements, and always return a `boolean`. For example, the `countIf()` algorithm allows you to count all of the values in a sequence that satisfy a user-supplied unary predicate, and the `sort()` algorithm orders the elements of a sequence using a user-supplied binary predicate.

JGL defines interfaces for unary and binary predicates. The `UnaryPredicate` interface defines a single method called `execute()` that takes a single `Object` parameter and returns a `boolean`. The `BinaryPredicate` interface defines a single method also called `execute()` that takes two `Object` parameters and returns a `boolean`. For example, here is the source code for the unary predicate object `PositiveInteger`:

```
package jgl;
import jgl.UnaryPredicate;

/**
 * PositiveInteger is a unary predicate that returns true if
 * its operand is positive.
 */

public final class PositiveInteger implements UnaryPredicate
  {
  /**
   * Return true if the operand is greater than zero.
   * @param object The operand, which must be an Integer.
   * @return object > 0
   */
  public boolean execute( Object object )
    {
    return ((Integer) object).intValue() > 0; // Cast and test.
    }
  }
```

The following example uses `PositiveInteger` to count the number of positive integers in a `Vector`:

Example Functions1.java

```
import jgl.*;

public class Functions1
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( new Integer( 3 ) );
    array.add( new Integer( -2 ) );
    array.add( new Integer( 3 ) );
    array.add( new Integer( -5 ) );
    array.add( new Integer( -4 ) );
    UnaryPredicate predicate = new PositiveInteger();
    int n = Counting.countIf( array, predicate );
    System.out.println( "Number of positive Integers in " + array + " = " + n
);
    }
  }
```

Output

```
Number of positive Integers in Vector( 3, -2, 3, -5, -4 ) = 2
```

Another example is a unary predicate called `LogicalNot` that returns `true` if its single operand is `false`. The following example uses an instance of this class to count all of the `false` elements in a native Java array:

Example Functions2.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Functions2
  {
  public static void main( String[] args )
    {
    boolean array[] = { false, false, true, false, true };
    BooleanArray bools = new BooleanArray( array );
    UnaryPredicate predicate = new LogicalNot();
    int n = Counting.countIf( bools, predicate );
    System.out.println( "Number of false in " + bools + " = " + n );
    }
  }
```

Output

```
Number of false in boolean[]( false, false, true, false, true ) = 3
```

Predicates are often used as a comparator for ordering elements. An object *A* will be placed to the left of an object *B* if the predicate object returns true when executed with *A* as the first operand and *B* as the second operand. **Note that a properly designed comparator should always return `false` when comparing two objects are are equal.** Here is the source code for an example comparator, `GreaterString`.

```
package jgl;

import jgl.BinaryPredicate;

/**
 * GreaterString is a binary predicate that returns true if the first operand
 * as a String is greater than the second operand as a String.
 */

public final class GreaterString implements BinaryPredicate
  {
  /**
   * Return true if the first operand is greater as a String than the
   * second operand as a String.
   * @param first The first operand, which is converted into a String
   * if necessary.
   * @param second The second operand, which is converted into a String
   * if necessary.
   * @return first.toString() > second.toString()
   */
  public boolean execute( Object first, Object second )
    {
    // Convert and test.
    return first.toString().compareTo( second.toString() ) > 0;
    }
  }
```

One variation of `sort()` allows you to specify a function that is used to control the order of sorting. In the following example, `GreaterString` tells `sort()` to place the first operand to the left of the second operand if the first operand is greater than the second operand.

Example Functions3.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Functions3
  {
  public static void main( String[] args )
    {
    Deque deque = new Deque();
    deque.add( "cat" );
    deque.add( "ape" );
    deque.add( "dog" );
    deque.add( "bat" );
    System.out.println( "unsorted = " + deque );
    BinaryPredicate comparator = new GreaterString();
    Sorting.sort( deque, comparator );
    System.out.println( "sorted = " + deque );
    }
  }
```

Output

```
unsorted = Deque( cat, ape, dog, bat )
sorted = Deque( dog, cat, bat, ape )
```

The next example uses the binary predicate `LessInteger` to sort a native array of `int`s.

Example Functions4.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Functions4
  {
  public static void main( String[] args )
    {
    int array[] = { 3, 1, 5, -2, 7, 9 };
    IntArray intArray = new IntArray( array );
    BinaryPredicate comparator = new LessInteger();
    System.out.println( "unsorted = " + intArray );
    Sorting.sort( intArray, comparator );
    System.out.println( "sorted = " + intArray );
    }
  }
```

Output

```
unsorted = int[]( 3, 1, 5, -2, 7, 9 )
sorted = int[]( -2, 1, 3, 5, 7, 9 )
```

For reference, here is a list of all the standard JGL function predicate objects. A later section explains how you can add further variations of your own.

Unary Predicates

UnaryPredicate	Operation
`BindFirstPredicate`	P(V,x)
`BindSecondPredicate`	P(x,V)
`LogicalNot`	!x
`NegativeInteger`	x < 0
`PositiveInteger`	x > 0
`UnaryComposePredicate`	P(Q(x))
`UnaryNot`	!P(x)

Binary Predicates

BinaryPredicate	Operation
`BinaryComposePredicate`	P(Q(x), R(y))
`BinaryNot`	!P(x, y)
`EqualTo`	x.equals(y)
`GreaterInteger`	x > y
`GreaterString`	x.toString().compareTo(y.toString()) > 0
`GreaterEqualInteger`	x >= y
`GreaterEqualString`	x.toString().compareTo(y.toString()) >= 0
`HashComparator`	x.hashCode() < y.hashCode()
`IdenticalTo`	x == y
`LessInteger`	x < y
`LessString`	x.toString().compareTo(y.toString()) < 0
`LessEqualInteger`	x <= y
`LessEqualString`	x.toString().compareTo(y.toString()) <= 0
`LogicalAnd`	x && y
`LogicalOr`	x || y
`NotEqualTo`	!x.equals(y)
`NotIdenticalTo`	x != y

In addition, here is a list of the standard JGL algorithms that accept a predicate:

- `adjacentFind()`
- `countIf()`
- `detect()`
- `every()`
- `findIf()`
- `includes()`
- `makeHeap()`
- `median()`
- `nextPermutation()`
- `popHeap()`
- `prevPermutation()`
- `pushHeap()`
- `reject()`
- `removeCopyIf()`
- `removeIf()`
- `replaceCopyIf()`
- `replaceIf()`
- `select()`
- `setDifference()`
- `setIntersection()`
- `setSymmetricDifference()`
- `setUnion()`
- `some()`
- `sort()`

- `sortHeap()`
- `unique()`
- `uniqueCopy()`

General Functions

General functions are function objects that take objects as parameters and return another object. They are often used to apply a mathematical operation to every element in a collection. JGL defines interfaces for unary and binary general functions. The `UnaryFunction` interface defines a single method called `execute()` that takes a single `Object` parameter and returns an `Object`. The `BinaryFunction` interface defines a single method also called `execute()` that takes two `Object` parameters and returns an `Object`. For example, here's the source code of an example general function, `NegateInteger`.

```
package jgl;

import jgl.UnaryFunction;

/**
 * NegateInteger is a unary function object that assumes that its operand is
 * an instance of Integer and returns its negation.
 */

public final class NegateInteger implements UnaryFunction
  {
  /**
   * Return the negation of my operand.
   * @param object The operand, which must be an instance of Integer.
   * @return -object
   */
  public Object execute( Object object )
    {
    return new Integer( -((Integer) object).intValue() );
    }
  }
```

This function object can be used by `transform()` to negate every object in a sequence and store the result into another sequence. The follow example uses `NegateInteger` to negate every element in a `Deque`:

Example Functions5.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Functions5
  {
  public static void main( String[] args )
    {
    Deque deque = new Deque();
    deque.add( new Integer( 4 ) );
    deque.add( new Integer( -2 ) );
    deque.add( new Integer( 3 ) );
    UnaryFunction function = new NegateInteger();
    System.out.println( "before = " + deque );
    Transforming.transform( deque, deque.begin(), function );
    System.out.println( "after = " + deque );
    }
  }
```

Output

```
before = Deque( 4, -2, 3 )
```

```
after = Deque( -4, 2, -3 )
```

For reference, here's a list of all the standard JGL general functions. A later section explains how you can add further variations of your own.

Unary Functions

Unary Function	Operation
`BindFirst`	P(V,x)
`BindSecond`	P(x,V)
`Hash`	x.hashCode()
`LengthString`	x.toString().length()
`NegateInteger`	-x
`SelectFirst`	x.first
`SelectSecond`	x.second
`ToString`	x.toString()
`UnaryCompose`	P(Q(x))
`UnaryPredicateFunction`	P(x)

Binary Functions

Binary Function	Operation
`BinaryCompose`	P(Q(x), R(y))
`BinaryPredicateFunction`	P(x,y)
`DividesInteger`	x / y
`MinusInteger`	x - y
`ModulusInteger`	x % y
`PlusInteger`	x + y
`PlusString`	x.toString() + y.toString()
`TimesInteger`	x * x

In addition, here's a list of all the standard JGL algorithms that accept general functions:

- `adjacentDifference()`
- `collect()`
- `inject()`
- `transform()`

Binders

The predicate objects `BindFirstPredicate` and `BindSecondPredicate` allow you to give a fixed value to either the 1st or the 2nd argument of a binary predicate, respectively. For example, here's how you can use `BindSecondPredicate` to count all of the strings in a `DList` that are greater than "cat".

Example Functions6.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Functions6
  {
  public static void main( String[] args )
    {
    DList list = new DList();
    list.add( "dog" );
    list.add( "ape" );
    list.add( "emu" );
    UnaryPredicate pred = new BindSecondPredicate( new GreaterString(), "bat"
);
    int n = Counting.countIf( list, pred );
    System.out.println( "The number of strings in " + list + " > bat = " + n
);
    }
  }
```

Output

```
The number of strings in DList( dog, ape, emu ) > bat = 2
```

The function objects `BindFirst` and `BindSecond` allow you to bind operands to general function objects.

Composers

The `UnaryComposePredicate` and `BinaryComposePredicate` function objects allow you to apply a secondary function to each operand before applying the main predicate. For example, to sort strings based on their length, use `BinaryComposePredicate` as follows.

Example Functions7.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Functions7
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( "ape" );
    array.add( "giraffe" );
    array.add( "lizard" );
    BinaryPredicate comparator = new BinaryComposePredicate(
      new GreaterInteger(), new LengthString(), new LengthString() );
    System.out.println( "before = " + array );
    Sorting.sort( array, comparator );
    System.out.println( "after = " + array );
    }
  }
```

Output

```
before = Array( ape, giraffe, lizard )
after = Array( giraffe, lizard, ape )
```

The function objects `UnaryCompose` and `BinaryCompose` allow you to apply a secondary function to each operand before applying the main general function.

Creating your own Function Objects

To create your own function object, use the following rules:

- If the function object you want to create is a predicate that must process one argument, implement the `UnaryPredicate` interface and define an `execute()` method that takes one `Object` argument and returns a `boolean`.

- If the function object you want to create is a predicate that must process two arguments, implement the `BinaryPredicate` interface and define an `execute()` method that takes two `Object` arguments and returns a `boolean`.

- If the function object you want to create is a general function that must process one argument, implement the `UnaryFunction` interface and define an `execute()` method that takes one `Object` argument and returns an `Object`.

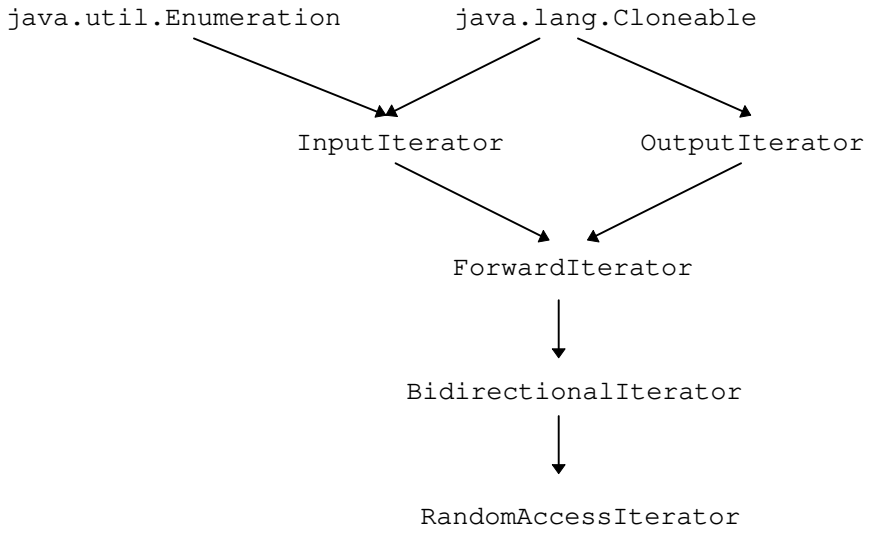
- If the function object you want to create is a general function that must process two arguments, implement the `BinaryFunction` interface and define an `execute()` method that takes two `Object` arguments and returns an `Object`.

Cast the input objects as necessary. If the method must return a number or character, make sure a primitive value is converted into its object equivalent using the primitive wrapper classes in `java.lang`.

Iterators

Most Java programmers are familiar with the concept of an *iterator*, which is an object that may be used to traverse a sequence of elements, whether they are in a JGL container, a JDK container, a native Java array, a Java stream, or another kind of sequence. The JDK `Enumeration` is a kind of iterator that is created by invoking `elements()` on a JDK container. When an `Enumeration` is first created, it is positioned at the first element of the container. As long as `hasMoreElements()` returns true, it is safe to use `nextElement()` to get the element that the Enumeration is currently positioned at and then advance the `Enumeration` to the next element.

An `Enumeration` is a limited form of iterator because it is read-only and can only be advanced by one position at a time. JGL includes a more powerful hierarchy of interfaces that embrace and extend `Enumeration`, as shown by the following diagram.



Here is a brief overview of each iterator interface.

Iterator Interface	Description
`InputIterator`	can read one item at a time, in a forward direction only
`OutputIterator`	can write one item at a time, in a forward direction only
`ForwardIterator`	combines characteristics of input and output iterators
`BidirectionalIterator`	like forward, plus the ability to move backwards
`RandomAccessIterator`	like bidirectional, plus the ability to jump by an arbitrary distance

Each JGL container has an associated class of iterator. Here's a table that lists every JGL container class in alphabetical order together with its associated iterator and the interface that its iterator implements.

Container	Associated Iterator	Iterator Interface
Array	ArrayIterator	RandomAccessIterator
BooleanArray	BooleanIterator	RandomAccessIterator
ByteArray	ByteIterator	RandomAccessIterator
CharArray	CharIterator	RandomAccessIterator
Deque	DequeIterator	RandomAccessIterator
DoubleArray	DoubleIterator	RandomAccessIterator
FloatArray	FloatIterator	RandomAccessIterator
IntArray	IntIterator	RandomAccessIterator
DList	DListIterator	BidirectionalIterator
HashMap	HashMapIterator	ForwardIterator
ObjectArray	ObjectIterator	RandomAccessIterator
OrderedMap	OrderedMapIterator	BidirectionalIterator
OrderedSet	OrderedSetIterator	BidirectionalIterator
PriorityQueue	ArrayIterator	RandomAccessIterator
Queue	Same as underlying sequence's	Same as underlying sequence's
HashSet	HashSetIterator	ForwardIterator
ShortArray	ShortIterator	RandomAccessIterator
SList	SListIterator	ForwardIterator
Stack	Same as underlying sequence's	Same as underlying sequence's
VectorArray	VectorIterator	RandomAccessIterator

For compatibility with JDK, all JGL containers respond to `elements()` by returning an instance of their associated iterator class cast to an `Enumeration` and positioned at their first element. The following example uses an `Enumeration` to iterate through every element of an `Array`.

Example Iterators1.java

```
import jgl.*;
import java.util.Enumeration;

public class Iterators1
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( "magical" );
    array.add( "mystery" );
    array.add( "tour" );
    Enumeration iterator = array.elements();
    while( iterator.hasMoreElements() )
      System.out.println( iterator.nextElement() );
    }
  }
```

Output

```
magical
mystery
tour
```

All JGL containers respond to `start()` by returning an instance of their associated iterator class cast to a `ForwardIterator` and positioned at the first element. `finish()` works in a similar fashion except that the iterator is positioned immediately after the last element.

All JGL iterators may be compared using `equals()`, cloned using the standard `clone()` message, and advanced by one or more positions using `advance()`. You can read the object that an iterator is currently positioned at by using `get()`, and replace the object that an iterator is currently positioned at by using `put()`. The following example uses these operations to convert every `String` in an `Array` and a `DList` to uppercase.

Example Iterators2.java

```
import jgl.*;

public class Iterators2
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( "magical" );
    array.add( "mystery" );
    array.add( "tour" );
    System.out.println( "before array = " + array );
    toUppercase( array );
    System.out.println( "after array = " + array );

    DList list = new DList();
    list.add( "magical" );
    list.add( "mystery" );
    list.add( "tour" );
    System.out.println( "before list = " + list );
    toUppercase( list );
    System.out.println( "after list = " + list );
    }

  static void toUppercase( Container container )
    {
    // Obtain iterator positioned at first element.
    ForwardIterator iterator = container.start();

    // Obtain iterator positioned immediately after last element.
    ForwardIterator end = container.finish();

    // Loop through every element.
    while( !iterator.equals( end ) )
      {
      String current = (String) iterator.get();
      // Replace current element with uppercase equivalent.
      iterator.put( current.toUpperCase() );
      iterator.advance(); // Move forward by one element.
      }
    }
  }
```

Output

```
before array = Array( magical, mystery, tour )
after array = Array( MAGICAL, MYSTERY, TOUR )
before list = DList( magical, mystery, tour )
after list = DList( MAGICAL, MYSTERY, TOUR )
```

In most cases, the functionality of an `Enumeration` or a `ForwardIterator` will suffice. However, there are occasions that require you to use the full power of the container's native iterator type. For example, a `BidirectionalIterator` can also move backwards by one or more positions using `retreat()`, and can get and put elements that are a specified distance from their current position using a more powerful version of `get()` and `put()`. All JGL containers respond to `begin()` by returning an uncast instance of their associated iterator class positioned at their first element. Similarly, `end()` returns an uncast instance of a container's associated iterator class that is positioned immediately after its last element.

The following example uses some of these more advanced features to iterator backwards through an `Array`. Note that the same thing can be accomplished more easily by using a reverse iterator, described later in this chapter.

Example Iterators3.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Iterators3
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( "ape" );
    array.add( "giraffe" );
    array.add( "lizard" );
    System.out.println( "array = " + array );

    // Obtain iterator positioned "just-past-the-end".
    ArrayIterator iterator = array.end();

    // Obtain iterator positioned at first element.
    ArrayIterator begin = array.begin();

    while( !iterator.equals( begin ) )
      {
      iterator.retreat();
      System.out.println( iterator.get() );
      }
    }
  }
```

Output

```
array = Array( ape, giraffe, lizard )
lizard
giraffe
ape
```

Iterators and Algorithms

All JGL algorithms can be easily applied to an entire container. However, there are occasions where you might want to apply an algorithm to a sub-range of a container. Most algorithms therefore have a variation that allows you to specify a sub-range by supplying an iterator positioned at the first element of the range and another iterator positioned *immediately after* the last element of the range. The following two lines of code are therefore equivalent:

```
Sorting.sort( array );
Sorting.sort( array.begin(), array.end() );
```

The following example uses iterator-based versions of the sort and replace algorithms to sort the first half of an `Array` and to replace every 7 in the last three elements with a 0.

Example Iterators4.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Iterators4
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( new Integer( 4 ) );
    array.add( new Integer( 7 ) );
    array.add( new Integer( 2 ) );
    array.add( new Integer( 7 ) );
    array.add( new Integer( 1 ) );
    array.add( new Integer( 7 ) );
    System.out.println( "array = " + array );

    // Obtain iterator positioned at first element.
    ArrayIterator first = array.begin();

    // Obtain iterator positioned immediately after third element.
    ArrayIterator last = array.begin();
    last.advance( 3 );

    // Sort first three elements.
    Sorting.sort( first, last );
    System.out.println( "array = " + array );

    // Replace 7 with 0 in last three elements.
    Replacing.replace( last, array.end(), new Integer( 7 ), new Integer( 0 )
);
    System.out.println( "array = " + array );
    }
  }
```

Output

```
array = Array( 4, 7, 2, 7, 1, 7 )
array = Array( 2, 4, 7, 7, 1, 7 )
array = Array( 2, 4, 7, 0, 1, 0 )
```

Iterators and Containers

All operations can be performed on JGL containers without any knowledge of iterators. However, there are a few methods that can operate more efficiently with iterators. For example, let's say you wish to remove the first occurrence of a particular object from a `DList`. One way to do this is to use `indexOf()` to find the index of the first occurrence and then use index-based version of `remove()` to delete the node with that index. However, since a `DList` is not random access, the index-based versions of the methods can only reach a node with a particular index by starting at the head node and moving link by link until the specified node is reached. For long `DLists`, this would be very slow. A more efficient way to achieve the same result is to use `find()` to obtain an iterator positioned at the first occurrence and then use the iterator-based version of `remove()` to delete the node that the iterator is positioned at. Because the iterator refers directly to the node, it is very fast to unlink the node and no list traversal is necessary. The following example illustrates this technique.

Example Iterators5.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Iterators5
  {
  public static void main( String[] args )
    {
    DList list = new DList();
    list.add( "ape" );
    list.add( "bat" );
    list.add( "cat" );
    list.add( "dog" );
    System.out.println( "list = " + list );
    DListIterator iterator = list.find( "cat" );
    System.out.println( "iterator positioned @ " + iterator.get() );
    list.remove( iterator );
    System.out.println( "list = " + list );
    }
  }
```

Output

```
list = DList( ape, bat, cat, dog )
iterator positioned @ cat
list = DList( ape, bat, dog )
```

Another example of an iterator-oriented method is `equalRange()` in `HashMap`. This method returns a pair of iterators, with the first iterator positioned at the first occurrence of a particular key and the second iterator positioned immediately after the last occurrence of the key.

Note that `HashMapIterator` and `OrderedMapIterator` both contain the methods `key()` and `value()` which allow you to read the key and value associated with the iterators current position, respectively. They also both contain a version of `value()` that allows you to modify the value that the iterator is currently positioned at.

The following example illustrates most of these features.

Example Iterator6.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Iterators6
  {
  public static void main( String[] args )
    {
    HashMap map = new HashMap( true ); // Allow dupicates.
    map.add( "cat", "beauty" );
    map.add( "dog", "barky" );
    map.add( "cat", "agatha" );
    map.add( "fox", "paula" );
    System.out.println( "map = " + map );

    Pair pair = map.equalRange( "cat" );
    HashMapIterator iterator = (HashMapIterator) pair.first;
    HashMapIterator last = (HashMapIterator) pair.second;
    while( !iterator.equals( last ) )
      {
      System.out.print( "pair = " + iterator.get() );
      System.out.print( ", key = " + iterator.key() );
      System.out.println( ", value = " + iterator.value() );
      iterator.advance();
      }
    }
  }
```

Output

```
map = HashMap( Pair( dog, barky ), Pair( fox, paula ), Pair( cat, beauty ),
Pair( cat, agatha ) )
pair = Pair( cat, beauty ), key = cat, value = beauty
pair = Pair( cat, agatha ), key = cat, value = agatha
```

Reverse Iterators

Reverse iterators are just like regular iterators except that they traverse a sequence backwards instead of forwards. To iterator backwards through a container, construct a `ReverseIterator` onto a `BidirectionalIterator` that is positioned immediately after the end of the container. `nextElement()` moves a reverse iterator's underlying iterator one place backwards and then returns its current item. `hasMoreElements()` returns `true` until the reverse iterator has read the first element of the container.

The following example uses a reverse iterator for iterating backwards through an `Array` of `Strings`.

Example Iterators7.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Iterators7
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( "ape" );
    array.add( "bat" );
    array.add( "cat" );
    array.add( "dog" );
    System.out.println( "array = " + array );

    ReverseIterator iterator = new ReverseIterator( array.end() );
    while( iterator.hasMoreElements() )
      System.out.println( iterator.nextElement() );
    }
  }
```

Output

```
array = Array( ape, bat, cat, dog )
dog
cat
bat
ape
```

Insertion Iterators

JGL includes an output iterator called an `InsertIterator` that adds an object to a specified container using `add()` whenever the iterator is written to. The following example uses an `InsertIterator` to append the results of a `transform()` algorithm to a `Deque`.

Example Iterators8.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;

public class Iterators8
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( "ape" );
    array.add( "giraffe" );
    array.add( "elephant" );
    System.out.println( "array = " + array );

    Deque deque = new Deque();
    InsertIterator iterator = new InsertIterator( deque );
    UnaryFunction function = new LengthString();
    Transforming.transform( array, iterator, function );
    System.out.println( "deque = " + deque );
    }
  }
```

Output

```
array = Array( ape, giraffe, elephant )
deque = Deque( 3, 7, 8 )
```

Iterating over Java Streams

JGL allows you to construct an iterator onto a `java.io.OutputStream` by using the `OutputStreamIterator` class. When an object is written using `put()`, it is converted into a string using `toString()` and then written to the output stream. The default constructor connects the iterator to `System.out`. The following example uses an `OutputStreamIterator` to send the contents of a `Array` to the standard output stream.

Example Iterators9.java

```
// Copyright(c) 1996 ObjectSpace, Inc.
import jgl.*;
import java.util.Enumeration;

public class Iterators9
  {
  public static void main( String[] args )
    {
    Array array = new Array();
    array.add( "ape" );
    array.add( "bat" );
    array.add( "cat" );

    OutputStreamIterator output = new OutputStreamIterator( System.out );

    Enumeration input = array.elements();

    while( input.hasMoreElements() )
      output.put( input.nextElement() );
    System.out.println(); // Flush output.

    // You can use an algorithm to do the same thing.
    Copying.copy( array, output );
    System.out.println(); // Flush output.
    }
  }
```

Output

```
ape bat cat
ape bat cat
```

Summary of Classes

This section contains a summary of the classes in the Java Generic Library, organized by category.

Containers

Related Interfaces

`ArrayAdapter` ..... The interface implemented by all JGL array adapters.
`Container` ..... The interface implemented by all JGL containers.
`Sequence` ..... The interface implemented by all sequential JGL containers.
`Set` ..... The interface implemented by all JGL sets.

Sequences

`Array` ..... Linear, contiguous storage, fast inserts at the end only.
`Deque` ..... Linear, non-contiguous storage, fast inserts at extremities.
`DList` ..... Doubly linked list, fast inserts anywhere.
`SList` ..... Singly linked list, fast inserts at either extremity.

Maps

`HashMap` ..... 1-1 and 1-many mappings, stored using hash.
`OrderedMap` ..... 1-1 and 1-many mappings, stored in order.

Sets

`HashSet` ..... Fast object lookip, stores objects in order.
`OrderedSet` ..... Fast object lookup, stores object using hash.

Queues and Stacks

`PriorityQueue` ..... Pops items in a sorted order.
`Queue` ..... Strict first-in, first-out data structure.
`Stack` ..... Strict first-in, last-out data structure.

Array Adapters

`BooleanArray` ..... Allows a native array of booleans to act like a Container.
`ByteArray` ..... Allows a native array of bytes to act like a Container
`CharArray` ..... Allows a native array of chars to act like a Container.
`DoubleArray` ..... Allows a native array of doubles to act like a Container.
`FloatArray` ..... Allows a native array of floats to act like a Container.
`IntArray` ..... Allows a native array of ints to act like a Container.
`LongArray` ..... Allows a native array of longs to act like a Container.
`ObjectArray` ..... Allows a native array of Objects to act like a Container.
`ShortArray` ..... Allows a native array of shorts to act like a Container.
`VectorArray` ..... Allows a JDK Vector to act like a Container.

Exceptions

`InvalidOperationException` .... Inappropriate operation on a JGL container.

Algorithms

`Applying` ..... Applies a function to every element of a sequence.
`Comparing` ..... Mismatches, equality tests, lexicographical comparison.
`Copying` ..... Copies a sequence.
`Counting` ..... Counts unconditionally and conditionally.
`Filling` ..... Fills a sequence with a single element.
`Filtering` ..... Filters a sequence.
`Finding` ..... Finds an object or an element that satisfies a predicate.
`Heap` ..... Makes, pushes, pops, and sorts a heap.
`MinMax` ..... Finds the min and max of a sequence.
`Permuting` ..... Cycles through permutations of a sequence.
`Printing` ..... Prints sequences and containers.
`Removing` ..... Removes an object or an element that satisfies a predicate.
`Replacing` ..... Replaces an object or an element that satisfies a predicate.
`Reversing` ..... Reverses a sequence.
`Rotating` ..... Rotates a sequence.
`SetOperations` ..... Union, intersection, difference, and inclusion.
`Shuffling` ..... Shuffles a sequence.
`Sorting` ..... Sorts a sequence.
`Swapping` ..... Swaps elements or sequences.
`Transforming` ..... Maps one sequence to another.

Function Objects

Related Interfaces

`BinaryFunction` ..... The interface of all binary function objects.
`BinaryPredicate` ..... The interface of all binary predicate objects.
`UnaryFunction` ..... The interface of all unary function objects.
`UnaryPredicate` ..... The interface of all unary predicate objects.

Classes that implement the `BinaryFunction` interface

`BinaryCompose` ..... P(Q(x), R(x))
`BinaryPredicateFunction` ..... P(x)
`DividesInteger` ..... x / y
`MinusInteger` ..... x - y
`ModulusInteger` ..... x % y
`PlusInteger` ..... x + y
`PlusString` ..... x.toString() + y.toString()
`TimesInteger` ..... x * y

Classes that implement the `BinaryPredicate` interface

`BinaryComposePredicate` ..... P(Q(x), R(x))
`BinaryNot` ..... !P(x, y)
`EqualTo` ..... x.equals(y)
`GreaterEqualInteger` ..... x >= y
`GreaterEqualString` ..... x.toString().compareTo(y.toString()) >= 0
`GreaterInteger` ..... x > y
`GreaterString` ..... x.toString().compareTo(y.toString()) > 0
`HashComparator` ..... x.hashCode() < y.hashCode()
`IdenticalTo` ..... x == y
`LessEqualInteger` ..... x <= y
`LessEqualString` ..... x.toString().compareTo(y.toString()) <= 0
`LessInteger` ..... x < y
`LessString` ..... x.toString().compareTo(y.toString()) < 0
`LogicalAnd` ..... x && y
`LogicalOr` ..... x || y
`NotEqualTo` ..... !x.equals(y)
`NotIdenticalTo` ..... x != y

Classes that implement the `UnaryFunction` interface

`BindFirst` ..... P(V, x)
`BindSecond` ..... P(x, V)
`Hash` ..... x.hashCode()
`LengthString` ..... x.toString().length()
`NegateInteger` ..... -x
`Print` ..... stream.println(x)
`SelectFirst` ..... x.first
`SelectSecond` ..... x.second
`ToString` ..... x.toString()
`UnaryCompose` ..... P(Q(x))
`UnaryPredicateFunction` ..... P(x)

Classes that implement the `UnaryPredicate` interface

`BindFirstPredicate` ..... P(V, x)
`BindSecondPredicate` ..... P(x, V)
`LogicalNot` ..... !x
`NegativeInteger` ..... x < 0
`PositiveInteger` ..... x > 0
`UnaryComposePredicate` ..... P(Q(x))
`UnaryNot` ..... !P(x)

Iterators

Related Interfaces

`InputIterator` ..... Can read one item at a time in a forward direction
`OutputIterator` ..... Can write one item at a time in a forward direction
`ForwardIterator` ..... Combines characteristics of input and output iterators
`BidirectionalIterator` ..... Like forward, plus the ability to move backwards
`RandomAccessIterator` ..... Like bidirectional, plus the ability to jump

Classes that implement the `OutputIterator` interface

`InsertIterator` ..... Writes items into a container.
`OutputStreamIterator` ..... Writes items to an output stream.

Classes that implement the `ForwardIterator` interface

`SListIterator` ..... Iterates over an SList.

Classes that implement the `BidirectionalIterator` interface

`DListIterator` ..... Iterates over a DList.
`HashMapIterator` ..... Iterates over a HashMap.
`HashSetIterator` ..... Iterates over a HashSet.
`OrderedMapIterator` ..... Iterates over an OrderedMap.
`OrderedSetIterator` ..... Iterates over an OrderedSet.
`ReverseIterator` ..... Iterates backwards.

Classes that implement the `RandomAccessIterator` interface

`ArrayIterator` ..... Iterates over a Vector.
`BooleanIterator` ..... Iterates over a native array of booleans.
`ByteIterator` ..... Iterates over a native array of bytes.
`CharIterator` ..... Iterates over a native array of chars.
`DequeIterator` ..... Iterates over a Deque.
`DoubleIterator` ..... Iterates over a native array of doubles.
`FloatIterator` ..... Iterates over a native array of floats.
`IntIterator` ..... Iterates over a native array of ints.
`LongIterator` ..... Iterates over a native array of longs.
`ReverseIterator` ..... Iterates backwards.
`ShortIterator` ..... Iterates over a native array of shorts.
`VectorIterator` ..... Iterates over a java.util.Vector.

Utilities

`Benchmark` ..... Allows simple collection of benchmark information.

Appendix

Benchmarks

JGL is a high performance library whose containers and algorithms meet or beat the performance of existing libraries. In addition, the JGL algorithms are generally at least as fast as their hand-coded equivalents. The JGL installation includes the following set of benchmarks:

```
\jgl_1_1
  \benchmarks
    ArrayBenchmarks.java      Compares JGL Array against JDK Vector
    MapBenchmarks.java        Compares JGL Map against JDK Hashtable
    SortingBenchmarks.java    Compares JGL sort() against hand-coded quicksort
    *.txt                     Sample benchmarks
```

To run these benchmarks, compile them in the benchmarks directory by typing:

```
javac *.java
```

and then run them by typing

```
java ArrayBenchmarks
java MapBenchmarks
java SortingBenchmarks
```

Each of these programs uses the Benchmark class to perform timing comparisons, and display intermediate benchmarks as they are gathered. The final performance ratio is displayed at the end of each benchmark. A ratio of less than 1.0 means that JGL ran faster. A sample run of these benchmarks is included in the files `*.txt`. A summary of these files is included on the next page.

Please note that random variations in the garbage collection system can cause significant variance in an individual performance statistic. The performance claims that we make for JGL are based on an average of large numbers of measurements.

SAMPLE BENCHMARKS

These benchmarks were run on a Toshiba Tecra computer with
32Mb RAM running Windows 95 and JDK 1.0.2.

A ratio of less than 1.0 means that JGL ran faster.

ArrayBenchmarks.txt

```
ratio of jglArrayGetting to jdkVectorGetting is 0.850775
ratio of jglArrayPutting to jdkVectorPutting is 1.21103
ratio of jglArrayIterating to jdkVectorIterating is 1.12203
ratio of jglArrayAdding to jdkVectorAdding is 0.715431
ratio of jglArrayClearing to jdkVectorClearing is 0.204819
ratio of jglArrayInserting to jdkVectorInserting is 1.02515
ratio of jglArrayRemoving to jdkVectorRemoving is 0.980042
```

MapBenchmarks.txt

```
ratio of jglMapAdding to jdkHashtableAdding is 0.983074
ratio of jglMapFinding to jdkHashtableFinding is 1.00821
ratio of jglMapRemoving to jdkHashtableRemoving is 1.02933
ratio of jglMapClearing to jdkHashtableClearing is 0.44186
```

SortingBenchmarks.txt

```
ratio of jgl sort algorithm to handcoded sorting is 1.04997
```

A Comparison of JGL vs. STL

JGL takes advantage of the STL philosophy of separating containers from algorithms. It also embeds some of the most common algorithms directly into the JGL containers and supplies container-oriented versions of all algorithms. This approach allows JGL to be easy to use without losing the benefit of reusable algorithms.

This section lists the differences between JGL and STL as well as a side-by-side comparison of JGL and STL code segments.

Feature Enhancements

Here is a list of the feature that are in JGL but not in STL:

- JGL containers can be accessed simultaneously by multiple threads without a problem.
- JGL includes hashing maps and sets as well as ordered maps and sets.
- JGL includes a singly linked list.
- JGL `Maps` include an `add()` method that takes a key and a value.
- JGL `Maps` include a `put()` method for easily replacing the value associated with a key.
- JGL containers include additional methods for algorithms such as `replace()` and `remove()`.
- JGL sets directly support set operations such as `union()`.
- JGL lists support indexed access.
- JGL includes additional function objects such as `NegativeInteger`.
- JGL arrays support `pushFront()` and `popFront()`.
- JGL pop operations return the Object that is begin popped.
- JGL sequential containers provide `add()` as a synonym for `pushBack()`.
- JGL supplies additional algorithms such as those in `Printing`.
- JGL map iterators provide easy access to the key and value of their associated key/value pair.
- JGL random access iterators provide the `index()` method for obtaining their index.
- JGL random access containers provide indexed access to methods such as `remove()`, `replace()`.
- JGL supports heap functionality for bidirectional iterators as well as random access iterators.
- JGL algorithms can operate directly on containers as well as iterator pairs.
- JGL container iterators include the `atBegin()` and `atEnd()` testing methods.
- JGL `randomShuffle()` works with bidirectional iterators.
- JGL `sort()` can sort any kind of sequence.
- JGL stacks and queues work with any sequence, whereas STL required random access sequences.

Compatibility Features

Here is a list of the features that JGL has for compatibility with existing Java code:

- JGL algorithms can work on native arrays of objects and primitives as well as JGL containers.
- JGL iterators are backwards compatible with the `java.util.Enumeration` interface.
- JGL `Arrays` may be constructed to take ownership of a native array.
- JGL uses standard Java exceptions whenever possible.
- JGL maps extend the `java.util.Dictionary` abstract base class.
- JGL calls the STL container vector `Array` to avoid a name clash with `java.util.Vector`.
- JGL calls the hashing associative containers `HashMap` and `HashSet`.
- JGL uses `remove()` instead of `erase()`.
- JGL calls the ordered associative containers `OrderedMap` and `OrderedSet`.

Differences

Here is a list of the differences between JGL and STL that will probably always remain:

- JGL `Sets` and `Maps` return an `Object` from a failed `insert()` operation.
- JGL maps and sets provide `getComparator()` instead of `key_comp()`, `value_comp()`.
- JGL packages algorithms as `public static` methods of algorithm classes.
- JGL does not include `pointer_to_function` function objects.
- JGL does not need function object adapter functions.
- Every JGL container implements `add()`.

Code Comparisons

This section contains some code samples that illustrate some of the similarities and differences between JGL and STL.

Creating a Vector and performing some common functions.

STL/C++

```
vector<int> v;
v.push_back( 1 );
v.push_back( 3 );
v.push_back( 2 );
cout << v.front() << ", " << v.back() << endl;
replace( v.begin(), v.end(), 1, 3 );
```

JGL/Java

```
Array v = new Array();
v.pushBack( 1 ); // Could use v.add( 1 ) instead.
v.pushBack( 3 );
v.pushBack( 2 );
System.out.println( v.front() + ", " + v.back() );
v.replace( 1, 3 ); // Note that common functions are part of container interface.
```

Iterating through a container.
STL/C++
```
vector<int> v;
v.push_back( 1 );
v.push_back( 3 );
v.push_back( 2 );
for( vector<int>::iterator i = v.begin(); i != v.end(); i++ )
  cout << *I << endl;
```

JGL/Java
```
Array v = new Array ();
v.pushBack( 1 );
v.pushBack( 3 );
v.pushBack( 2 );
for( ArrayIterator i = v.begin(); !i.equals( v.end() ); i.advance() )
  System.out.println( i.get() );
```

or (clearer and more efficient)

```
Array v = new Array ();
v.pushBack( 1 );
v.pushBack( 3 );
v.pushBack( 2 );
Enumeration e = v.begin();
while( e.hasMoreElements() )
  System.out.println( e.nextElement() );
```

Iterating through a native array.
STL/C++
```
int array[] = { 1, 3, 2 };
for( int* p = array; p != array + 3; p++ )
  cout << *p << endl;
```

JGL/Java
```
int array[] = { 1, 3, 2 };
for( IntIterator p = IntIterator.begin( array ); p.hasMoreElements(); p.advance() )
  System.out.println( p.get() );
```

Sorting a container.
STL/C++
```
vector<int> v;
v.push_back( 1 );
v.push_back( 3 );
v.push_back( 2 );
sort( v.begin(), v.end() );
```

JGL/Java
```
Array v = new Array();
v.pushBack( new Integer( 1 ) );
v.pushBack( new Integer( 3 ) );
v.pushBack( new Integer( 2 ) );
Sorting.sort( v );
```

Sorting a native array.
STL/C++
```
int array[] = { 1, 3, 2 };
sort( array, array + 3 );
```

JGL/Java
```
int ints [] = { 1, 3, 2 };
IntArray array = new IntArray( ints );
Sorting.sort( array );
```