ENCE 688P

Java Collections Framework

Contents

Chapter 11

Java Collections Framework

11.1 Pathway from Objects to Groups of Objects

Now that we know how to create objects, the next subject of importance is ...

> **... how to organize collections of objects so that they are easy to store, easy to find, and easy to modify.**

We address this problem in two-step procedure:

1. Choose an appropriate mathematical formalism.
2. Develop software to support each formalism.

As a starting point, Figure 11.1 shows how groups of objects can be organized into sets and maps.

Part I. Sets

A set is nothing more than ...

> **... a group of objects containing no duplicates.**

The left-hand schematic in Figure 11.1 shows three sets X, Y, and Z. Some real-world examples of set include the following:

1. The set of lowercase letters “a” through “z.”
2. The set of digits “0” through “9.”
3. Sets of people – friends, relatives, co-workers.

Sets have the following properties:

1. They can contain only one instance of an item.
2. They may be finite or infinite.
3. They can define abstract concepts (e.g., family relatives).

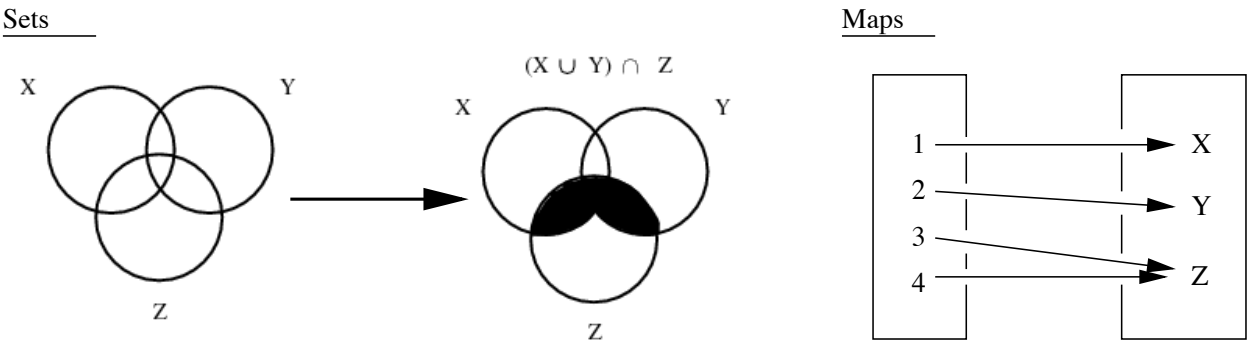

Figure 11.1. Organizing groups of objects into sets and maps.

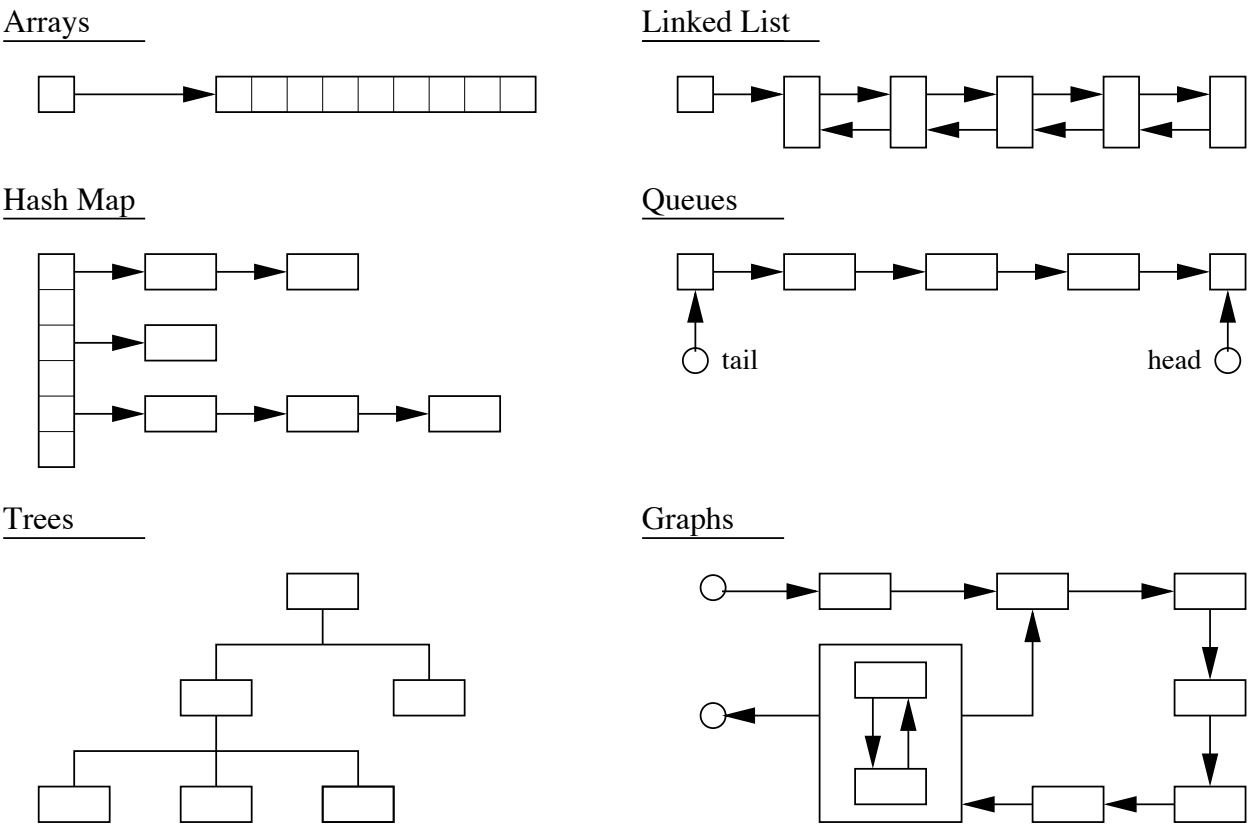

Figure 11.2. Schematics for array, list, queue, hashtable, tree and graph data structures.

Sets are fundamental to logic. As such, things get interesting when we want to evaluate the relationship among sets; for example, compute the intersection, union or difference of sets.

Part II. Maps

A map defines ...

> **... a set of pairs, each pair representing a one-directional mapping from one element to another.**

Examples of maps include:

1. The map of IP addresses to domain names (DNS).
2. A dictionary of words mapped to their meanings.
3. A Celsius to Fahrenheit temperature conversion (this is an infinite map).
4. A telephone directory connecting names to phone numbers.
5. Source and destination maps for edges in a graph.
6. Parent and child relationships connecting dependencies among cells in a spreadsheeet.

In geographic information systems, map relationships connect spatial coordinates to features (i.e., what features are at this location?) and vice versa.

Data Structures and Algorithms

Long before the advent of computers, mathematicians realized the benefits in casting problems as sets and maps. More recently, computer scientists have developed techniques for dealing with these concerns ...

> **... through the study of data structures and algorithms.**

Simply put, a data structure is ...

> **... a particular way of storing and organizing data in a computer so that it can be used efficiently.**

Data structures means to manage large amounts of data efficiently and, as such, they are used in almost every program or software system. The study of data structures and algorithms is interesting because ...

> **... different kinds of data structures are suited to different kinds of applications.**

Usually, efficient data structures are a key to designing efficient algorithms (e.g., for adding/removing elements). Figure 11.2 shows, for example, schematics for array, list, queue, hashtable, tree and graph data structures. A few key points:

1. An array data structure stores a number of elements of the same type. These elements are accessed using an index to indicate which element is required.

2. A linked list data structure consists of a group of nodes which together represent a sequence. In the simplest implementation, each node is composed of a datum and a link to the next node in the sequence. More complex implementations provide additional links to allow for traversal in both directions. Linked lists allow for efficient insertion or removal of elements from any position in the sequence.

3. A queue is a linked list that only allows insertions at the end/tail and removal of items from the front/head of the queue.

4. A hash map provides a key-value pair.

5. A tree data structure orders it data items into a hierarchy. The major advantage of trees over other data structures is that the related sorting and search algorithms, and traversals can be very efficient.

6. A graph data structure consists of a finite set of ordered pairs, called edges or arcs, of certain entities called nodes or vertices. An edge (x,y) is said to point or go from x to y. The nodes may be part of the graph structure, or as illustrated in Figure 11.2 lower-level graphs.

In applications where the data requirements are known at compile time, the data structures can be of a fixed size throughout the program operation. A programmer might choose a data structure that provides very fast access to data. Increasingly, however, software programs are required to operate on streams of data of unknown length. In these cases, a programmer might choose a data structure for its flexibility – that is, ease of adding and removing elements.

11.2 Introduction to Java Collections Framework

A **collection** is an object that groups multiple elements into a unit (in mathematical terms it is equivalent to a set) that can be treated as a single entity. Typically, collections represent items of data that form a natural group:

1. A poker hand (a collection of cards),

2. A mail folder (a collection of letters), or a

3. Telephone directory (a mapping of names to phone numbers).

A collection that requires all of its elements to be of the same type is called homogeneous. Heterogeneous collections allow for elements to be of different types – for example, a collection of fruit might contain apples, oranges and bananas.

The collections framework is a ...

... unified architecture for representing and manipulating collections.

Collections are used to store, retrieve, manipulate, and communicate aggregate data. This includes:

1. **Sets.**

   A set is a collection that cannot contain duplicate elements.

2. **Lists.**

   An ordered collection (sometimes called a sequence). Lists can contain duplicate elements.

3. **Queues.**

   Queues typically order elements in a FIFO (first-in-first-out) manner.

4. **Maps.**

   An object that maps keys to values.

5. **SortedSet.**

   A Set that maintains its elements in ascending order. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

5. **SortedMap.**

   Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

A few points to note:

1. Some collections allow duplicate elements and others do not.

   Some are ordered and others unordered.

2. Lists, queues and maps are examples of linear collections. Elements are arranged in a sequence such that all elements except the first have a unique predecessor, and all except the last have a unique successor.

3. As illustrated in Figure 11.2, trees are hierarchical collections. The elements of the tree are called nodes. A non-trivial tree (i.e., one that is not empty) has a special node called the root. The root node has no predecessors (called parents) and zero or more successors (called children). Tree elements called leaves have one parent and no children.

4. Graph collections are similar to trees (more precisely, trees are a subset of graphs), but unlike trees, graph collections permit cyclic relationships. The elements of a graph are called vertices. Vertices are connected by edges. The graph shown in the bottom right-hand corner of Figure 11.2 is an example of a directed graph – each edge has a clearly defined predecessor (head) and successor (tail).

Benefits. The collections framework offers the following benefits:

1. Reduces programming effort,

2. Increase programming speed and quality,

3. Provide a standard way of accessing collections,

4. Allow for effective reuse of code.

General Purpose Operations

In any implementation of a collection (details specified below), we need a well defined set of operations that will be supported, even across collection types that serve different purposes. The basic operations include support for:

1. Adding an Element

We need to be able to add elements to a collection. The details for how this will happen are collection dependent. For example, collections that are position dependent may allow for insertion at a specific position (e.g., at the front or rear of the collection). Other collections will insert new elements in a position that maintains order in the value of elements. Some collections such as arraylists allow for duplicates; set collections do not allow duplicates.

2. Removing an Element

We also need to be able to remove elements from a collection. Sometimes we will remove an element based on its position – for example, removing an element from the top of a stack. Elements can also be removed from collections based on their target values – for example, remove items from a collection having a specific value.

3. Replacing an Element

Given a position or target element, this operation replaces an element with a different element.

4. Finding and retrieving and Element

Given a position or target element, this operation finds and retrieves a specific element from the collection.

5. Determining if an Element is contained in a Collection

This operation will determine if an eleent is contained in a collection. The operation will return a boolean value of true if it exists. Otherwise, it will return false.

6. Computing the Size of a Collection

The operation computes and returns the number of elements in the collection.

7. Testing to see if a Collection is Empty

This operation will test to see if a collection is empty. The operation will return a boolean value of true if it is empty. Otherwise, it will return false.

8. Traversing a Collection

Traversal is a strategy for systematically visiting the elements of a collection, one element at a time. We will see that iterators provide a uniform framework for collection traversal.

More advanced operations include:

9. Equality

This boolean operation determines if two collections are equal, that is, the collections contain the same number of elements, for each element in the first collections, their is an equal element in the second collection. Collections that are ordered will also require that the positions of the equal elements also match.

10. Cloning

The cloning operation produces a copy of the collection.

A **shallow copy** duplicates the structure of the original collection, but not the elements contained in the collection (i.e., in other words, a shallow copy replicates the structure and referenes to objects). As such, in a shallow copy, both the original collection and its copy will share access to the original elements.

In a **deep copy**, both the elements and the element structure will be replicated.

11. Serialization

Serialization is the ability to write and save the data in a collection to disk.

General Purpose Implementation

From an implementation standpoint, the Java Collections framework is ...

> **... a set of utility classes, interfaces (located in the java.util package), and algorithms for working with collections of objects.**

In a bit more detail:

1. **Interfaces**

   These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation.

2. **Implementations**

   These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

3. **Algorithms**

   These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

Relationship between Interfaces and Implementations

Table 11.1 provides a simplified view of data structure implementations for each of the Set, List and Map interfaces.

Interfaces	Hash Table	Resizable Array	Balanced Tree	Linked List
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Map	HashMap		TreeMap	

Table 11.1. Matrix of collection interfaces mapped to data structure implementations.

And Figure 11.3 shows the organization of interfaces, abstract classes, and concrete classes in the Java Collections Framework.

Key points to note:

1. There are only three container components – Map, List and Set, and as shown above only 2 or 3 implementations of each one.

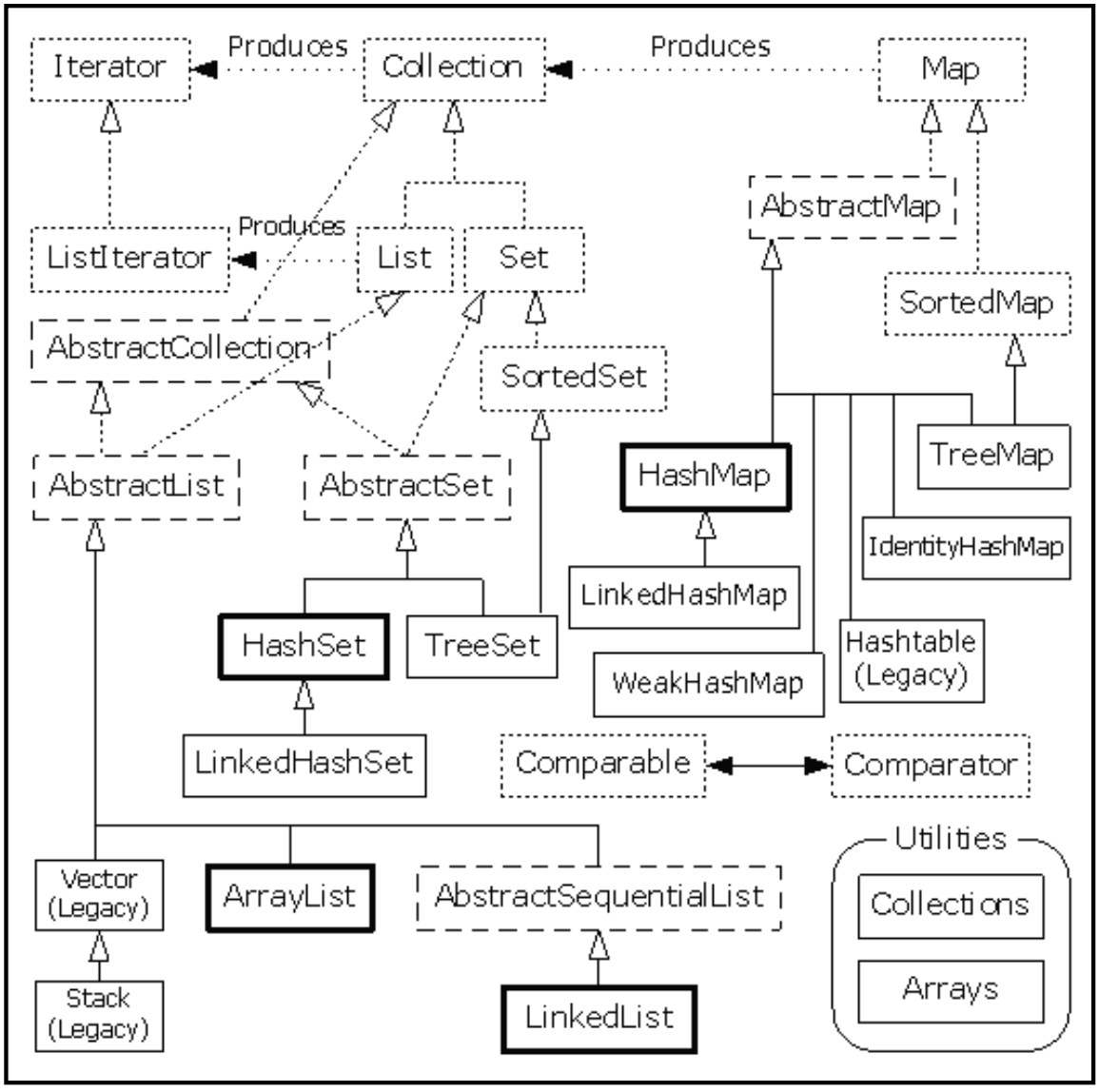

Figure 11.3. Organization of interfaces, abstract classes, and concrete classes in the Java Collections Framework.

2. The dotted boxes represent interfaces, the dashed boxes represent abstract classes, and the solid boxes are regular (concrete) classes.

3. The dotted-line arrows indicate that a particular class is implementing an interface (or in the case of an abstract class, partially implementing that interface).

4. The solid arrows show that a class can produce objects of the class the arrow is pointing to. For example, any Collection can produce an Iterator and a List can produce a ListIterator (as well as an ordinary Iterator, since List is inherited from Collection).

Iterators provide a way for ...

... collections to be traversed in a uniform way.

For a good overview, see Chapter 22 of Liang [2].

11.3 The Core Collection Interfaces

As illustrated in Figure 11.4 below, collections come in four basic flavors, Lists, Sets, Maps and Queues.

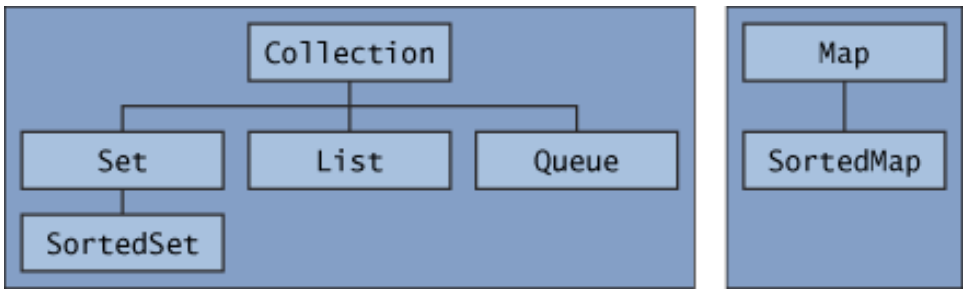

Figure 11.4. Hierarchy of interfaces in the Java Collections Framework.

A Collection holds single elements, and a Map holds associated pairs. Collection and Map are two top level interfaces.

The Collection Interface

The following fragment of code shows the Collection interface.

```
public interface Collection<E> extends Iterable<E> {

   // Basic operations
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(E element);
   boolean remove(Object element);
   Iterator<E> iterator();

   // Bulk operations
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c);
   boolean removeAll(Collection<?> c);
   boolean retainAll(Collection<?> c);
   void clear();

   // Array operations
   Object[] toArray();
   <T> T[] toArray(T[] a);
}
```

This is the root interface in the collections hierarchy. The interface has methods to tell you:

- How many elements are in the collection (size, isEmpty),
- To check whether a given object is in the collection (contains),

- To add and remove an element from the collection (add, remove), and
- To provide an iterator over the collection (iterator).

All of the general-purpose Collection implementation classes typically implement Collection indirectly through one of its subinterfaces. The collections interface specifies that implementations provide two ”standard” constructors:

1. A void (no arguments) constructor, which creates an empty collection, and
2. A constructor with a single argument of type Collection, which creates a new collection with the same elements as its argument.

The latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. There is no way to enforce this convention (as interfaces cannot contain constructors) but all of the general-purpose Collection implementations in the Java platform libraries comply.

The List Interface

The list interface is an extention of the collections interface, i.e.,

```
public interface List<E> extends Collection<E> {

   // Positional access
   E get(int index);
   E set(int index, E element);
   boolean add(E element);
   void add(int index, E element);
   E remove(int index);
   boolean addAll(int index, Collection<? extends E> c);

   // Search
   int indexOf(Object o);
   int lastIndexOf(Object o);

   // Iteration
   ListIterator<E> listIterator();
   ListIterator<E> listIterator(int index);

   // Range-view
   List<E> subList(int from, int to);
}
```

and defines methods for ...

... creating and working with an ordered collection (also known as a sequence) that allows duplicates.

Users of this interface have precise control over where in the list each element is inserted. Users can access elements by their integer index (position in the list), and search for elements in the list. Unlike sets, lists usually allow dupilicate elements.

Implementations of the List Interface

The Java Collections Framework provides three implementations of List interface:

1. **ArrayList**

   An ArrayList is an array which grows dynamically and provides an ordered collection (by index), but not sorted. Array lists give you fast iteration and fast random access.

2. **Vector**

   A Vector (legacy class) is basically the same as an ArrayList, but Vector methods are synchronized for thread safety (i.e., programs that have multiple processes).

3. **LinkedList**

   A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another. This linkage gives you new methods (beyond what you get from the List interface) for adding and removing from the beginning or end, which makes it an easy choice for implementing a stack or queue.

Keep in mind that a LinkedList may iterate more slowly than an ArrayList, but it's a good choice when you need fast insertion and deletion. Moreover, as of Java 5, the LinkedList class has been enhanced to implement the `java.util.Queue` interface. As such, it now supports the common queue methods: peek(), poll(), and offer().

The Set Interface

The set interface builds upon methods declared in the collections interface, i.e.,

```
public interface Set<E> extends Collection<E> {

   // Basic operations
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(E element);
   boolean remove(Object element);
   Iterator<E> iterator();

   // Bulk operations
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c);
   boolean removeAll(Collection<?> c);
   boolean retainAll(Collection<?> c);
```

```
    void clear();

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

and defines methods for ...

... creating and working with a collection that contains no duplicate elements.

Implementations of the Set Interface

The Java Collections Framework provides three implementations of Set interface:

1. HashSet

A HashSet is an unsorted, unordered Set. It uses the hashcode of the object being inserted, so the more efficient your hashCode() implementation the better access performance you'll get.

Use this class when you want a collection with no duplicates and you don't care about order when you iterate through it.

2. LinkedHashSet

A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements.

Use this class instead of HashSet when you care about the iteration order. When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

2. TreeSet

The TreeSet is one of two sorted collections (the other being TreeMap). It uses a Red-Black tree structure to guarantee that the elements will be in ascending order, according to natural order.

Optionally, you can construct a TreeSet with a constructor that lets you give the collection your own rules for what the order should be (rather than relying on the ordering defined by the elements' class) by using a Comparable or Comparator.

As of Java 6, TreeSet implements NavigableSet.

The Map Interface

The Map interface, i.e.,

```
public interface Map<K,V> {

   // Basic operations
   V put(K key, V value);
   V get(Object key);
   V remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();

   // Bulk operations
   void putAll(Map<? extends K, ? extends V> m);
   void clear();

   // Collection Views
   public Set<K> keySet();
   public Collection<V> values();
   public Set<Map.Entry<K,V>> entrySet();

   // Interface for entrySet elements
   public interface Entry {
      K getKey();
      V getValue();
      V setValue(V value);
   }
}
```

is an object that maps keys to values (i.e., key/value or name/value pairs). A map cannot contain duplicate keys; each key can map to at most one value.

The Map interface provides three collection views, which allow a map's contents to be viewed as ...

... a set of keys, collection of values, or set of key-value mappings.

Implementations of the Map Interface

Implementations of the Map interface let you do things like:

- Search for a value based on the key,
- Ask for a collection of just the values, or
- Ask for a collection of just the keys.

The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

The Java Collections Framework provides three implementations of Map interface:

1. HashMap

The HashMap gives you an unsorted, unordered Map.

When you need a Map and you don't care about the order (when you iterate through it), then HashMap is the way to go.

HashMap allows one null key and multiple null values in a collection.

2. Hashtable

Like Vector, Hashtable has existed from prehistoric Java times.

Hashtable is the synchronized counterpart to HashMap.

Note, however, that unlike HashMap, Hashtable doesn't let you have anything that's null.

3. LinkedHashMap

Like its Set counterpart, LinkedHashSet, the LinkedHashMap collection maintains insertion order (or, optionally, access order). Although it will be somewhat slower than HashMap for adding and removing elements, you can expect faster iteration with a LinkedHashMap.

4. TreeMap

You can probably guess by now that a TreeMap is a sorted Map. And you already know that by default, this means "sorted by the natural order of the elements."

Like TreeMap lets you define a custom sort order (via a Comparable or Comparator) when you construct a TreeMap, that specifies how the elements should be compared to one another when they're being ordered.

As of Java 6, TreeMap implements NavigableMap.

Like Sets, Maps rely on the equals() method to determine whether two keys are the same or different.

The Queue Interface

A Queue is a collection for holding elements prior to processing in some way. Queues are typically thought of as ...

... FIFO (first-in, first-out), but other orders are possible.

Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations. The Queue interface is as follows;

```
public interface Queue<E> extends Collection<E> {
   E element();
   boolean offer(E e);
   E peek();
   E poll();
   E remove();
}
```

The collections framework provides a priority queue implementation:

1. PriorityQueue

The PriorityQueue class was introduced with the release of Java 5.

Since the LinkedList class has been enhanced to implement the Queue interface, basic queues can be handled with a LinkedList.

The purpose of a PriorityQueue is to create ...

... a "priority-in, priority out" queue as opposed to a typical FIFO queue.

A PriorityQueue's elements are ordered either by natural ordering (in which case the elements that are sorted first will be accessed first) or according to a Comparator. In either case, the elements' ordering represents their relative priority.

11.4 Working with Array Lists and Linked Lists

An arraylist is an object that provides ...

... a resizable-array implementation of the List interface.

Arraylists have **one major constraint**:

... they can only store references to objects, not primitives.

So, for example, an array list can store a collection of String objects, but cannot store a list of (primitive) floats or ints.

Otherwise, arraylists are straightforward. The fragment of code:

```
ArrayList grocery = new ArrayList();
```

creates an empty arraylist object. To add an object to the arraylist, we call the add() method, passing a reference to the objects we want to store. Let's add a few items to our grocery list,

```
grocery.add ( "Bread" );
grocery.add ( "Milk" );
grocery.add ( "Cheese" );
```

The arraylist gives each element an index number. Like regular arrays, the first element has index number 0, the next is 1, and so forth. We can retrieve and print the individual grocery items with:

```
System.out.println ("Item 1: " + grocery.get(0) );
System.out.println ("Item 2: " + grocery.get(1) );
System.out.println ("Item 3: " + grocery.get(2) );
```

The `size()` method (e.g., grocery.size()) returns the number of elements in the arraylist.

Now let's repeat the grocery list assembly, but define separate String objects first. For example,

```
String item1 = "Bread";
String item2 = "Milk";
String item3 = "Cheese";

ArrayList grocery = new ArrayList();
grocery.add ( item1 );
grocery.add ( item2 );
grocery.add ( item3 );
```

The variables item1, item2, and item3 are references to the respective String objects. When we call the add() method, we pass a reference to the String objects. Arraylist does not make a copy of the object – instead there is only one copy of each String object, and the array list only stores a reference to that object. This results in the layout memory shown in Figure 11.5.

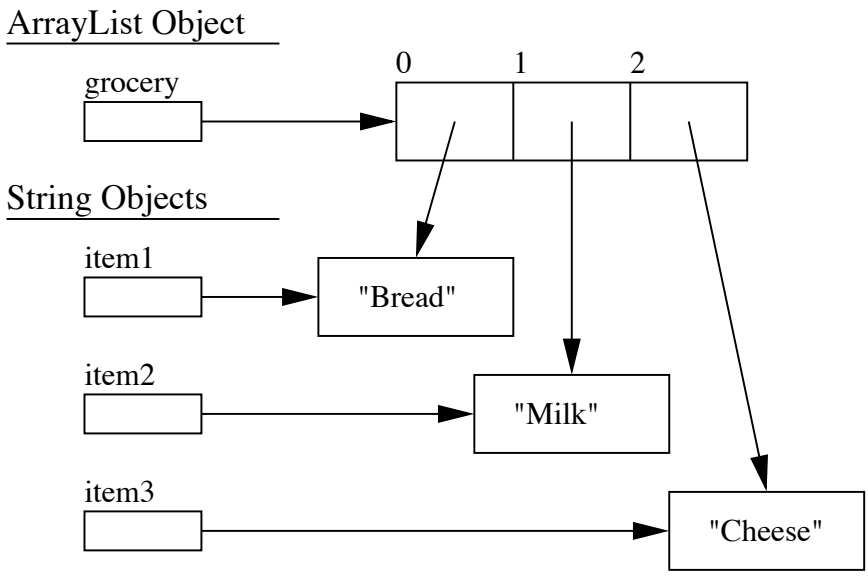

Figure 11.5. Layout of memory for arraylist of grocery items.

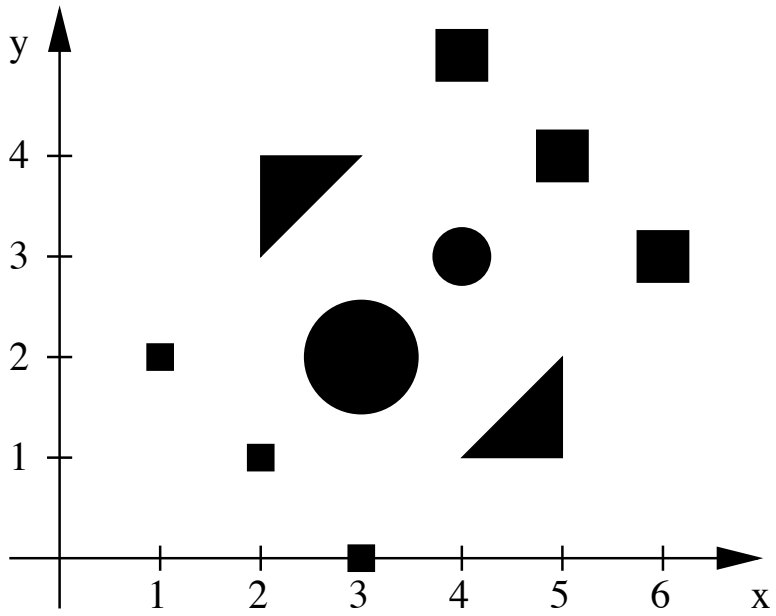

Figure 11.6. Spatial layout of circle, rectangle and triangle shapes.

11.5 Example 1. Arraylist of Shapes

In Chapter **??** we we used the fragment of code,

```
Shape s[] = new Shape [3];

s[0] = new Rectangle( 3.0, 3.0, 2.0, 2.0 );
s[1] = new Circle( 1.0, 2.0, 2.0 );
s[2] = new Rectangle( 2.5, 2.5, 2.0, 2.0 );
```

to create an array of abstract shapes implemented as combinations of circles and rectangles.

The following program builds upon this idea, and uses an arraylist to store a grid of rectangle, circle and triangle shapes as shown in Figure 11.6. The small and large rectangles have sidelength 0.25 and 0.5 respectively. The small and large circles have radius 0.5 and 1.0 respectively. Circle and rectangle shapes are positioned at their center points. Triangles are defined by the position of the three nodal/corner points.

Here is the source code:

```
/*
 *  ================================================================
 *  ShapeGrid.java: Use an array list to store items in a grid of
 *                  spatially arranged shapes.
 *
 *  Written By: Mark Austin                                December 2009
 *  ================================================================
 */

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

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

      // Create and initialize a grid of ten shapes

      List shapes = new ArrayList();

      Shape s0 = new Rectangle( 0.25, 0.25, 1.0, 2.0 );
      Shape s1 = new Rectangle( 0.25, 0.25, 2.0, 1.0 );
      Shape s2 = new Rectangle( 0.25, 0.25, 3.0, 0.0 );
      Shape s3 = new Circle   ( 0.5, 3.0, 2.0 );
      Shape s4 = new Circle   ( 1.0, 4.0, 3.0 );
      Shape s5 = new Triangle ( "t1", 2.0, 3.0, 3.0, 4.0, 2.0, 4.0 );
      Shape s6 = new Triangle ( "t2", 4.0, 1.0, 5.0, 1.0, 5.0, 2.0 );
      Shape s7 = new Rectangle( 0.5, 0.5, 4.0, 5.0 );
      Shape s8 = new Rectangle( 0.5, 0.5, 5.0, 4.0 );
      Shape s9 = new Rectangle( 0.5, 0.5, 6.0, 3.0 );

      // Add shapes to the array list ....
```

```
      shapes.add( s0 ); shapes.add( s1 );
      shapes.add( s2 ); shapes.add( s3 );
      shapes.add( s4 ); shapes.add( s5 );
      shapes.add( s6 ); shapes.add( s7 );
      shapes.add( s8 ); shapes.add( s9 );

      // Print details of individual shapes in the grid ....

      System.out.println("Grid of spatially arranged shapes");
      System.out.println("---------------------------------");

      for (int ii = 1; ii <= shapes.size(); ii = ii + 1)
         System.out.println ( shapes.get(ii-1).toString() );

      System.out.println("---------------------------------");

      // Compute and print total shape area .....

      double dArea = 0.0;
      for (int ii = 1; ii <= shapes.size(); ii = ii + 1) {
           Shape s = (Shape) shapes.get(ii-1);
           dArea = dArea + s.area();
      }

      System.out.println("");
      System.out.printf("Total Area   = %10.2f\n", dArea );
      System.out.println("---------------------------------");
   }
}
```

The abbreviated output is as follows:

```
prompt >>
prompt >> java ShapeGrid
Grid of spatially arranged shapes
---------------------------------
Rectangle : Side1 = 0.25 Side2 = 0.25
Rectangle : Side1 = 0.25 Side2 = 0.25
Rectangle : Side1 = 0.25 Side2 = 0.25
Circle : Radius = 0.5 [x,y] = [3.0,2.0]
Circle : Radius = 1.0 [x,y] = [4.0,3.0]

... details of triangle and rectangle output removed .....

Total Area   =       5.86
---------------------------------
```

Key points to note are as follows:

1. We can combine the tasks of shape creation and addition to the arraylist into a single statement. For example,

```
shapes.add( new Rectangle( 0.25, 0.25, 1.0, 2.0 ) );
```

2. The looping construct:

```
for (int ii = 1; ii <= shapes.size(); ii = ii + 1) {
    Shape s = (Shape) shapes.get(ii-1);
    dArea = dArea + s.area();
}
```

walks along the array list, retrieves the (ii-1)th item, then computes the required operation. While this approach is very straightforward, it is slow primarily because the operation

```
shapes.get(ii-1)
```

has to start at the list and walk to the (ii-1) item.

A faster approach is to use Iterators. The corresponding implementation looks like:

```
Iterator iterator1 = shapes.iterator();
while ( iterator1.hasNext() != false ) {
      Shape s = (Shape) iterator1.next();
      dArea = dArea + s.area();
}
```

Now the iterator simply walks along the list once and computes the required operation.

11.6 Example 2. Create and Sort an Arraylist Folded Boxes

In the following script of code we create an arraylist of folded box objects (see the classes and objects chapter for details),

```
-------------------------------------------------------------
Folded Box:     Name   Length (in)  Width (in)  Height (in)
-------------------------------------------------------------
               Match           2.0         1.0          0.5
                Shoe          12.0         8.0          7.0
             Packing          36.0        36.0         36.0
                Gift          12.0        12.0          5.0
            Dumpster          72.0        48.0         48.0
-------------------------------------------------------------
```

and then use inner classes implemented with a **comparator interface** to sort and print the collection ranked according to surface area and volume.

The source code is as follows:

```
/*
 *  ========================================================================
 *  FoldedBoxArray.java: Create and sort an arraylist of folded box objects.
 *
 *  Written By: Mark Austin                                          May 2007
 *  ========================================================================
 */

import java.util.*;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class FoldedBoxArray {
   ArrayList boxes = new ArrayList();

   // Sort boxes by surface area ....

   public void sortBySurfaceArea() {
      Collections.sort( boxes, new areaCompare() );
   }

   class areaCompare implements Comparator {
      public int compare( Object o1, Object o2) {
          FoldedBox b1 = (FoldedBox) o1;
          FoldedBox b2 = (FoldedBox) o2;
          if ( b1.surfaceArea() == b2.surfaceArea() )
               return  0;
          else if ( b1.surfaceArea() > b2.surfaceArea() )
               return  1;
          else
               return -1;
      }
   }

   // Sort boxes by volume ....

   public void sortByVolume() {
      Collections.sort( boxes, new volumeCompare() );
   }

   class volumeCompare implements Comparator {
      public int compare( Object o1, Object o2) {
          FoldedBox b1 = (FoldedBox) o1;
          FoldedBox b2 = (FoldedBox) o2;
          if ( b1.volume() == b2.volume() )
               return  0;
          else if ( b1.volume() > b2.volume() )
               return  1;
          else
               return -1;
      }
   }

   // ========================================================
```

```
// Create an array list of folded box objects ....
// ======================================================

public static void main ( String args [] ) {

   // Create an object of type folded box array ....

   FoldedBoxArray fba = new FoldedBoxArray();

   // Create and initialize folded box objects

   FoldedBox fbMatch = new FoldedBox ("Match",  2.0, 1.0, 0.5 );
   FoldedBox fbShoe  = new FoldedBox ( "Shoe", 12.0, 8.0, 7.0 );
   FoldedBox fbPacking  = new FoldedBox(  "Packing", 36.0, 36.0, 36.0 );
   FoldedBox fbGift     = new FoldedBox(     "Gift", 12.0, 12.0,  5.0 );
   FoldedBox fbDumpster = new FoldedBox( "Dumpster", 72.0, 48.0, 48.0 );

   // Add boxes to the array list .....

   fba.boxes.add( fbMatch );
   fba.boxes.add( fbShoe );
   fba.boxes.add( fbPacking );
   fba.boxes.add( fbGift );
   fba.boxes.add( fbDumpster );

   // Walk along unsorted list and print details ....

   System.out.println("Open folded boxes            ");
   System.out.println("=============================");

   for ( int i = 0; i < fba.boxes.size(); i = i + 1 ) {
       FoldedBox fp = (FoldedBox) fba.boxes.get(i);
       System.out.println( fp.toString() );
   }

   // Sort and print list by volume...

   fba.sortByVolume();

   System.out.println("Folded boxes sorted by volume ");
   System.out.println("============================== ");

   for ( int i = 0; i < fba.boxes.size(); i = i + 1 ) {
       FoldedBox fp = (FoldedBox) fba.boxes.get(i);
       System.out.println( fp.toString() );
   }

   // Sort and print list by surface area...

   fba.sortBySurfaceArea();

   System.out.println("Folded boxes sorted by surface area ");
   System.out.println("=================================== ");

   for ( int i = 0; i < fba.boxes.size(); i = i + 1 ) {
```

```
            FoldedBox fp = (FoldedBox) fba.boxes.get(i);
            System.out.println( fp.toString() );
        }
    }
}
```

The abbreviated input and output is as follows:

```
prompt >> java FoldedBoxArray
Open folded boxes
============================
FoldedBox: Match
Volume       = 1.0
Surface Area = 5.0

.... output removed ...

FoldedBox: Dumpster
Volume       = 165888.0
Surface Area = 14976.0

Folded boxes sorted by volume
============================
FoldedBox: Match
Volume       = 1.0
Surface Area = 5.0

FoldedBox: Shoe
Volume       = 672.0
Surface Area = 376.0

.... output removed ...

FoldedBox: Dumpster
Volume       = 165888.0
Surface Area = 14976.0

Folded boxes sorted by surface area
==================================
FoldedBox: Match
Volume       = 1.0
Surface Area = 5.0

.... Output is as for sort by surface area ...

prompt >>
prompt >> exit
```

Points to note:

1. This program uses the FoldedBox class provided in Chapter 8. areaCompare and volumeCompare are inner classes. Hence, the files before and after compilation are as follows:

```
-------------------------------------------------------------
Before Compilation          After Compilation
=============================================================
FoldedBox.java              FoldedBox.class
FoldedBoxArray.java         FoldedBox.java
                            FoldedBoxArray$areaCompare.class
                            FoldedBoxArray$volumeCompare.class
                            FoldedBoxArray.class
                            FoldedBoxArray.java
=============================================================
```

2. The first thing that main() does is create an object of type FoldedBoxArray.

3. FoldedBoxArray creates objects of type FoldedBox(), the same code as in Section **??**.

4. The methods sortByVolume() and sortBySurfaceArea() take care of the arraylist sorting. However, since an object of type FoldedBoxArray exits, the method calls need to be:

```
fba.sortByVolume();
fba.sortBySurfaceArea();
```

`areaCompare` and `volumeCompare` are inner classses that implement the `Comparator` interface specification. Each class provides details for a method compare that sysematically compares the surface area and volume properties of two folded box objects.

5. An example of sorting with Collections can be found on the class web page, in Family.java, which creates an arraylist of person objects for the Simpsons.

11.7 Working with Maps

Maps are objects that ...

... map keys onto values.

Keys cannot be duplicated. Maps have three collection views, a set of keys, a collection of values, and a set of key-value mappings. Some maps have a guaranteed order, but not all do.

HashMaps. HashMaps provide ...

... a hashtable-backed implementation of the Map interface.

Under ideal circumstances (no collisions), HashMap offers O(1) performance. Worst case performance (very unlikely) is O(n) – this occurs when all keys map to the same hash code.

HashMap is part of the JDK Collections API. It differs from Hashtable (now deprecated/out-of-date) in that it accepts the null key and null values, and it does not support Enumeration views. Also, it is not synchronized. If you plan to use it in multiple threads, consider using:

```
Map m = Collections.synchronizedMap(new HashMap(...));
```

TreeMaps. The treemap algorithms are adopted from Cormen, Leiserson, and Rivest's *Introduction to Algorithms* text [1] and provide:

... a red-black tree implementation of the SortedMap interface.

Elements in the Map will be sorted by either a user-provided Comparator object, or by the natural ordering of the keys.

A redblack tree is a type of self-balancing (or reasonably balanced) binary search tree, typically to implement associative arrays. As illustrated in Figure 11.7, the following requirements apply to redblack trees:

1. A node is either red or black.

2. The root is black.

3. All leaves are the same color as the root.

4. Both children of every red node are black.

5. Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

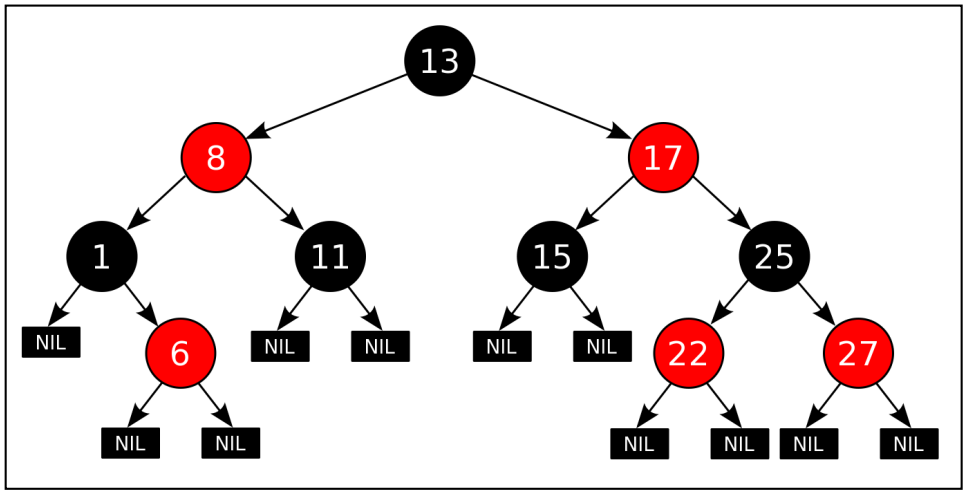

Figure 11.7. Schematic for a small red-black tree.

These constraints work together to ensure that the path from the root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf. The result is that the tree is roughly balanced. Since operations such as inserting, deleting, and finding values require worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows redblack trees to be efficient, even in the worst-case.

Figure 11.8 illustrates the assembly and incremental balancing of a larger red-black tree. The complexity of implementation is justified by good worst-case running time for its operations and overall efficiency in practice. It can search, insert, and delete items in O(log N) time, where N is the total number of elements in the tree. It is important to keep in mind, however, that there is a large constant in front of ”log n” (overhead involved in keeping the tree balanced). As a result, TreeMap may not be the best choice for small collections. If something is already sorted, you may want to just use a LinkedHashMap to maintain the order while providing O(1) access.

TreeMap is a part of the JDK Collections API. Null keys are allowed only if a Comparator is used which can deal with them; natural ordering cannot cope with null. Null values are always allowed. Note that the ordering must be *consistent with equals* to correctly implement the Map interface. If this condition is violated, the map is still well-behaved, but you may have suprising results when comparing it to other maps.

And like HashMap, the implementation of TreeMap is not synchronized. If you need to share this between multiple threads, do something like:

```
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
```

Example 1. Create a Simple HashMap of Strings

In this example we create ...

> **... a hashmap between the social security number for an employee, represented in a string format, and a reference to the employee object.**

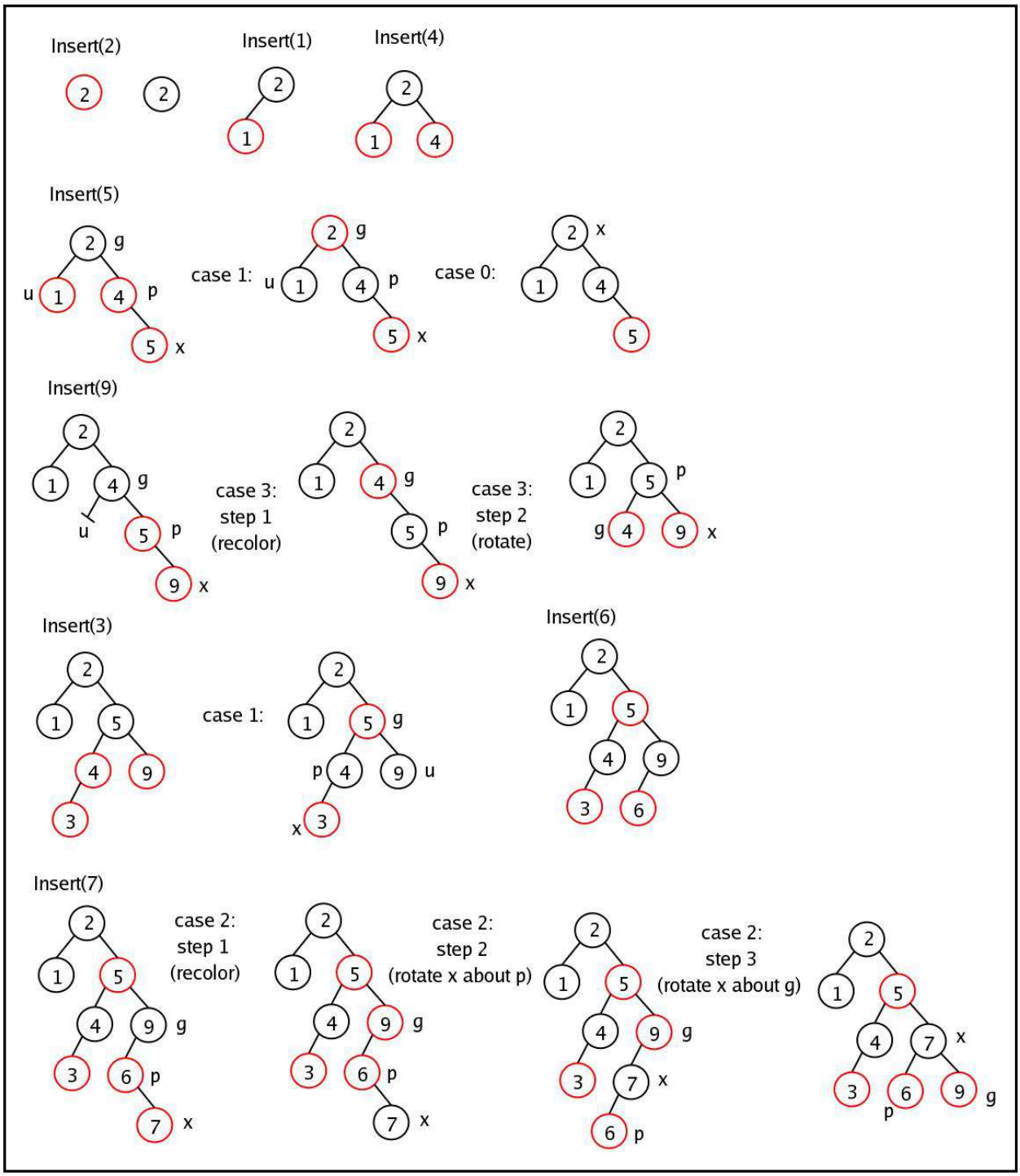

Figure 11.8. Schematic for assembly and incremental balancing of a red-black tree.

This program demonstrates most of the useful map operations: (1) adding new relationships to the map, (2) printing the contents of the map, (3) removing an entry, (4) replacing an entry, (5) finding an entry, and (6) iterating through all of the entries in the map.

A stripped-down version of the Employees class is as follows:

source code

```
/**
 *  ================================================================
 *  Employee.java. A minimalist employee class for testing purposes.
 *  ================================================================
 */

public class Employee {
   private String   name;
   private double salary;

   public Employee(String n) {
      name   = n;
      salary = 0;
   }

   public String toString() {
      return "[name=" + name + ", salary=" + salary + "]";
   }
}
```

And the source code for assembling and testing the hashmap is:

source code

```
/**
 *  ====================================================================
 *  TestHashMap.java. This program demonstrates the use of a map with
 *  key type String and value type Employee.
 *
 *  Author: Cay Horstmann.
 *  Modified by: Mark Austin
 *  ====================================================================
 */

import java.util.*;

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

      // Create and populate the hash map ....

      Map<String, Employee> staff = new HashMap<String, Employee>();

      staff.put("144-25-5464", new Employee("Amy Lee"));
      staff.put("567-24-2546", new Employee("Harry Hacker"));
```

```
      staff.put("157-62-7935", new Employee("Gary Cooper"));
      staff.put("456-62-5527", new Employee("Francesca Cruz"));

      // Print all entries

      System.out.println("Print all entries in the HashMap as a set");
      System.out.println("=======================================");

      System.out.println(staff);

      // Remove an entry

      System.out.println("Remove entry with id = 567-24-2546");
      staff.remove("567-24-2546");

      // Replace an entry

      System.out.println("Replace entry with id = 456-62-5527");
      staff.put("456-62-5527", new Employee("Francesca Miller"));

      // Look up a value

      System.out.println("Find employee with id 157-62-7935");
      System.out.println("==================================");

      System.out.println( staff.get("157-62-7935") );

      // Iterate through all entries

      System.out.println("Iterate over the Hashmap Entries ");
      System.out.println("==================================");

      for (Map.Entry<String, Employee> entry : staff.entrySet()) {
         String key     = entry.getKey();
         Employee value = entry.getValue();
         System.out.println("key = " + key + ", value = " + value);
      }

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

And here is the program output (slightly reformatted):

```
Print all entries in the HashMap as a set
=======================================
{ 157-62-7935=[name=Gary Cooper, salary=0.0],
  567-24-2546=[name=Harry Hacker, salary=0.0],
  144-25-5464=[name=Amy Lee, salary=0.0],
  456-62-5527=[name=Francesca Cruz, salary=0.0] }

Remove entry with id = 567-24-2546
Replace entry with id = 456-62-5527
```

```
Find employee with id 157-62-7935
==================================
[name=Gary Cooper, salary=0.0]

Iterate over the Hashmap Entries
================================
key = 157-62-7935, value = [name=Gary Cooper, salary=0.0]
key = 144-25-5464, value = [name=Amy Lee, salary=0.0]
key = 456-62-5527, value = [name=Francesca Miller, salary=0.0]
================================
```

Points to note are as follows:

1. The statement:

```
Map<String, Employee> staff = new HashMap<String, Employee>();
```

creates an empty hashmap. The use of Java Generics prevents relationships from being added to the map that are not a String, Employee pair.

2. Hashmaps have their own toString() method. Hence, the statement:

```
System.out.println(staff);
```

creates and prints a string representation of the entries in the hashmap.

Example 2. Use HashMap and TreeMap to Count Frequency of Words in Document

The following program reads a stream of words from the keyboard. We use a hashmap to store words and their frequency of usage. Then, we use a treemap to order the mapping.

The source code is follows:

source code

```
/*
 *  ================================================================================
 *  CountWordFrequency: Count frequency of words in text read from keyboard.
 *  ================================================================================
 */

import java.util.*;

public class CountWordFrequency {
   public static void main(String args[]) {
      Map map = new HashMap();
      Integer ONE = new Integer(1);

      // Read stream of input from keyboard ..
```

```
      for (int i=0, n=args.length; i<n; i++) {
         String key = args[i];
         Integer frequency = (Integer)map.get(key);
         if (frequency == null) {
            frequency = ONE;
         } else {
            int value = frequency.intValue();
            frequency = new Integer(value + 1);
         }
         map.put(key, frequency);
      }

      // Print (unordered) contents of map ..

      System.out.println("HashMap of [ word, frequency ] usage");
      System.out.println("====================================");

      System.out.println(map);

      // Create and print an ordered treemap...

      System.out.println("TreeMap of [ word, frequency ] usage");
      System.out.println("====================================");

      Map sortedMap = new TreeMap(map);
      System.out.println(sortedMap);

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

For the stream of text:

```
This is the test file. Here is a short sentence in the English
language that contains all twenty six letters. The quick brown fox
jumps over the lazy dog.
```

the (edited) program output is as follows:

```
prompt >>
prompt >> java CountWordFrequency This is the test file. Here is the ... etc.

HashMap of [ word, frequency ] usage
====================================
{ short=1, fox=1, test=1, letters.=1, quick=1, sentence=1, contains=1, a=1,
  dog.=1, This=1, six=1, The=1, over=1, Here=1, all=1, file.=1, is=2, jumps=1,
  the=3, in=1, English=1, that=1, twenty=1, brown=1, language=1, lazy=1}

TreeMap of [ word, frequency ] usage
====================================
{ English=1, Here=1, The=1, This=1, a=1, all=1, brown=1, contains=1, dog.=1, file.=1,
```

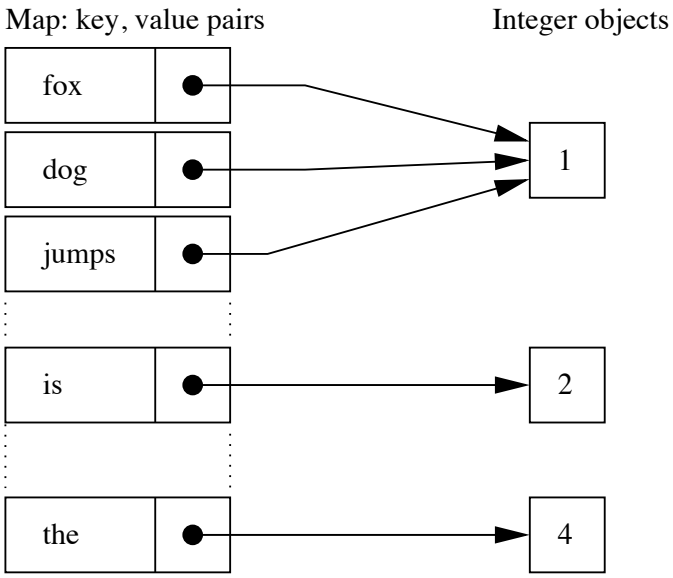

Figure 11.9. Schematic of key,value pairs and connection to word frequency modeled as references to integer objects.

```
  fox=1, in=1, is=2, jumps=1, language=1, lazy=1, letters.=1, over=1,
  quick=1, sentence=1, shortest=1, six=1, test=1, that=1, the=4, twenty=1}
===================================
```

Notice that the hashmap contents are printed according to the [key, value] pairs, where key is the word being stored and value is a reference to an object of type Integer storing the word frequency. Figure 11.9 is a partially complete schematic for the map contents. Generally speaking, a user will not need to know these details.

The treemap constructor takes the hashmap as an argument – this is an example of a treemap being created from another collection – and creates and ordered mapping. A more sophisticated implementation would remove the punctuation symbols (e.g., dog.) from the words.

Example 3. Use a Comparator to Order a TreeMap of Employees

In this example we use customized comparators to affect the ordering of items in a treemap of university employees. The source code is divided into three files:

1. Employees.java – a simple definition for an employee,
2. EmployeeCoparator.java – an implementation of the Comparator interface to order employees, and
3. TestUniversity.java – source code to assemble the employees and systematically build the treeset according to three strategies.

Employee.java. The details of Employee.java are as follows:

source code

```
/*
 *  ================================================================
 *  Employee.java: Create objects for company employees...
 *  ================================================================
 */

import java.util.Comparator;

public class Employee implements Comparable {
  String department, name;

  public Employee(String department, String name) {
     this.department = department;
     this.name = name;
  }

  public String getDepartment() {
     return department;
  }

  public String getName() {
     return name;
  }

  public String toString() {
     return "\n [dept = " + department + ", name = " + name + "]";
  }

  public int compareTo(Object obj) {
     Employee emp = (Employee) obj;
     int deptComp = department.compareTo( emp.getDepartment() );

     return ((deptComp == 0) ? name.compareTo(emp.getName()) : deptComp);
  }

  public boolean equals(Object obj) {
    if (!(obj instanceof Employee)) {
       return false;
    }
    Employee emp = (Employee) obj;
    return department.equals(emp.getDepartment())
        && name.equals(emp.getName());
  }

  public int hashCode() {
    return 31 * department.hashCode() + name.hashCode();
  }
}
```

The statement:

```
int deptComp = department.compareTo( emp.getDepartment() );
```

evaluates to zero when both employees belong to the same department. Then, the syntax:

```
((deptComp == 0) ? name.compareTo(emp.getName()) : deptComp);
```

is equivalent to:

```
if ( deeptComp == 0 )
   return name.compareTo(emp.getName();
else
   return deptComp;
```

A summary of the strategy is as follows: first, the compareTo() method compares employees based on the department to which they belong. For those cases where two employees belong to the same department, employees are ordered alphabetically by name.

EmployeeComparator.java. The following class compares the names of company employees. When two employees have the same name, then they are ranked according to department.

source code

```
/*
 *  ================================================================
 *  EmployeeComparator.java: Class for comparing the names of company
 *  employees. If two employees have the same name, then they are
 *  ranked according to department.
 *  ================================================================
 */

import java.util.Comparator;

public class EmployeeComparator implements Comparator {

   public int compare(Object obj1, Object obj2) {
      Employee emp1 = (Employee) obj1;
      Employee emp2 = (Employee) obj2;

      int nameComp = emp1.getName().compareTo( emp2.getName() );

      return ((nameComp == 0) ? emp1.getDepartment().compareTo(
               emp2.getDepartment()) : nameComp);
   }
}
```

TestUniversity.java. And finally, the details of TestUniversity.java are as follows:

source code

```
/*
 *  ====================================================================
 *  TestUniversity.java: Assemble university employees into a variety of
```

```
 *                       treeset organizations.
 *  ========================================================================
 */
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.Set;
import java.util.TreeSet;

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

      // Define employees at the University of Maryland ...

      Employee e01 = new Employee( "ISR/Finance", "Strahan, Jason");
      Employee e02 = new Employee( "ISR", "Sutton, Steve");
      Employee e03 = new Employee( "ISR", "Coriale, Jeff");
      Employee e04 = new Employee( "ISR", "Austin, Mark");
      Employee e05 = new Employee( "CEE", "Austin, Mark");
      Employee e06 = new Employee( "ISR", "Lovell, David");
      Employee e07 = new Employee( "CEE", "Lovell, David");
      Employee e08 = new Employee( "ISR", "Ghodssi, Reza");
      Employee e09 = new Employee( "ECE", "Ghodssi, Reza");
      Employee e10 = new Employee( "CEE", "Baecher, Greg");
      Employee e11 = new Employee( "CEE", "Haghani, Ali");
      Employee e12 = new Employee( "CEE", "Gabriel, Steve");

      // Define arry of reference to employees ....

      Employee emps[] = { e01, e02, e03, e04, e05, e06,
                          e07, e08, e09, e10, e11, e12 };

      // Part 1. Assemble treeset from array of objects.

      System.out.println("Part 01: Create Treeset based on array of employees");
      System.out.println("================================================");

      Set set01 = new TreeSet( Arrays.asList(emps) );
      System.out.println( set01 );

      // Part 2. Now use Collection.reverseOrder() method to reverse treeset
      //         assembly.

      System.out.println("\nPart 02: Reverse order assembly of items in TreeSet ");
      System.out.println("================================================");

      Set set02 = new TreeSet( Collections.reverseOrder() );
      set02.addAll( Arrays.asList(emps) );
      System.out.println( set02 );

      // Part 3. Use EmployeeComparator to order items in Treeset.

      System.out.println("\nPart 03: Use EmployeeComparator to Order TreeSet items");
      System.out.println("================================================");
```

```
      Set set03 = new TreeSet( new EmployeeComparator() );
      for (int i = 0, n = emps.length; i < n; i++) {
         set03.add( emps[i] );
      }
      System.out.println( set03 );

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

The program output is as follows:

```
Part 01: Create Treeset based on array of employees
=====================================================
[
 [dept = CEE, name = Austin, Mark],
 [dept = CEE, name = Baecher, Greg],
 [dept = CEE, name = Gabriel, Steve],
 [dept = CEE, name = Haghani, Ali],
 [dept = CEE, name = Lovell, David],
 [dept = ECE, name = Ghodssi, Reza],
 [dept = ISR, name = Austin, Mark],
 [dept = ISR, name = Coriale, Jeff],
 [dept = ISR, name = Ghodssi, Reza],
 [dept = ISR, name = Lovell, David],
 [dept = ISR, name = Sutton, Steve],
 [dept = ISR/Finance, name = Strahan, Jason]]

Part 02: Reverse order assembly of items in TreeSet
=====================================================
[
 [dept = ISR/Finance, name = Strahan, Jason],
 [dept = ISR, name = Sutton, Steve],
 [dept = ISR, name = Lovell, David],
 [dept = ISR, name = Ghodssi, Reza],
 [dept = ISR, name = Coriale, Jeff],
 [dept = ISR, name = Austin, Mark],
 [dept = ECE, name = Ghodssi, Reza],
 [dept = CEE, name = Lovell, David],
 [dept = CEE, name = Haghani, Ali],
 [dept = CEE, name = Gabriel, Steve],
 [dept = CEE, name = Baecher, Greg],
 [dept = CEE, name = Austin, Mark]]

Part 03: Use EmployeeComparator to Order TreeSet items
=====================================================
[
 [dept = CEE, name = Austin, Mark],
 [dept = ISR, name = Austin, Mark],
 [dept = CEE, name = Baecher, Greg],
 [dept = ISR, name = Coriale, Jeff],
 [dept = CEE, name = Gabriel, Steve],
 [dept = ECE, name = Ghodssi, Reza],
```

```
 [dept = ISR, name = Ghodssi, Reza],
 [dept = CEE, name = Haghani, Ali],
 [dept = CEE, name = Lovell, David],
 [dept = ISR, name = Lovell, David],
 [dept = ISR/Finance, name = Strahan, Jason],
 [dept = ISR, name = Sutton, Steve]]
=====================================================
```

Key points to note:

1. The pair of statements:

```
Set set01 = new TreeSet( Arrays.asList(emps) );
System.out.println( set01 );
```

in Part 1 employ the comparable interface, i.e.,

```
public interface Comparable {
   public int compareTo(Object o)
}
```

and the method `compareTo()` to impose a total ordering of Employee objects. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method. As previously explained, employees are first ordered by name of the department to which they belong (e.g., CEE, ISR, ECE) and then by their name. The statement

```
System.out.println( set01 );
```

is equivalent to:

```
System.out.println( set01.toString() );
```

2. Part 2 uses the Collection.reverseOrder() method to reverse default ordering in the treeset. The TreeSet class has access to Collections methods because it implements the Set interface, which in turn, subclasses the Collections interface.

3. Part 3 uses the custom EmployeeComparator() to order items in treeset. In contrast to Part 1, employees are first ordered by name, then by the departments to which they belong.

Example 4: Create HashMap for Point(x,y)-Value Pairs

This section demonstrates how hashmaps can store mappings from (x,y) coordinates to values. In a sparse matrix or spreadsheet application, the values x and y would correspond to the row and column numbers. In a geography application, (x,y) values might correspond to latitude and longitude and the value might represent a physical quantity associated with that location – elevation, measured rainfall, and so forth.

Figure 11.10 shows the mapping of four coordinate pairs to four values.

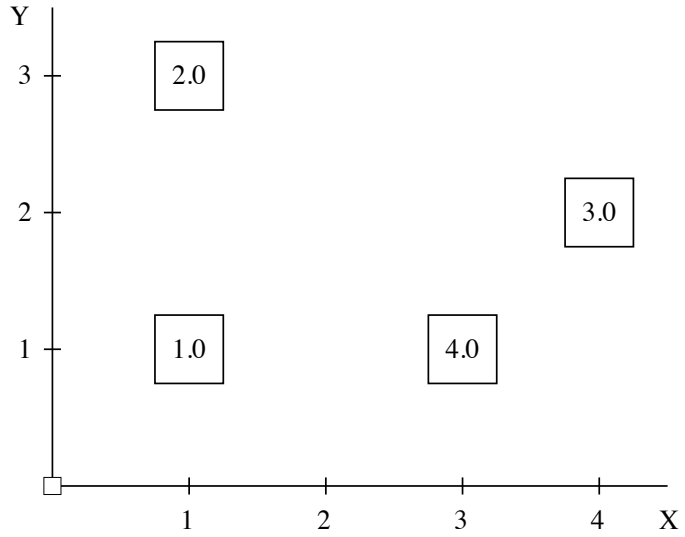

Figure 11.10. Hashmap for storing point(x,y) - double value pairs.

The source code to setup and print the mapping relationship is as follows:

source code

```
/*
 *  ========================================================================
 *  TestHashMapPoint.java: Create and print (x,y) coordinate point to double
 *  value map.
 *
 *  Written By: Mark Austin                                        March, 2013
 *  ========================================================================
 */

import java.awt.Point;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class TestHashMapPoint {

   // Print details of (x,y) coordinate point-to-double mapping ...

   private static void printHashMap( Map<Point, Double> map, final String message ) {
```

```
      System.out.println( "Begin: " + message );
      System.out.println( "----------------------------------------------" );

      final Iterator iterator = map.keySet().iterator();
      while ( iterator.hasNext() ) {
         final Point    key = (Point) iterator.next();
         final Double value = map.get( key );

         System.out.println( key + " " + value.toString() );
      }

      System.out.println( "----------------------------------------------" );
      System.out.println( "End: " + message );
   }

   // Exercise methods in (x,y) coordinate point-to-double mapping ...

   public static void main(String[] args) {

      // Part 01: Create point-to-double hashmap ...

      System.out.println("Part 01: Build simple hashmap<Point,Double>");
      Map<Point, Double> map = new HashMap<Point, Double>();

      // Add data to the hashmap ....

      Point  key01 = new Point(1, 1);
      Double val01 = new Double( 1.0 );
      Point  key02 = new Point(1, 3);
      Double val02 = new Double( 2.0 );
      Point  key03 = new Point(4, 2);
      Double val03 = new Double( 3.0 );
      Point  key04 = new Point(3, 1);
      Double val04 = new Double( 4.0 );

      map.put(key01, val01);
      map.put(key02, val02);
      map.put(key03, val03);
      map.put(key04, val04);

      // Print details of hashmap ...

      printHashMap( map, "Map (x,y) coordinate --> double value" );

      // Part 02: Create point-to-double hashmap ...

      System.out.println("");
      System.out.println("Part 02: Demonstrate that points must be unique .. ");

      // Modify mappings for (1,1) and (1,3) ....

      map.put( new Point(1, 1), new Double( 5.0 ) );
      map.put( new Point(1, 3), new Double( 6.0 ) );

      // Print details of hashmap ...
```

```
      printHashMap( map, "Map (x,y) coordinate --> double value" );

   }
}
```

The program output is as follows:

```
prompt >>
Part 01: Build simple hashmap<Point,Double>
Begin: Map (x,y) coordinate --> double value
-------------------------------------------
java.awt.Point[x=4,y=2] 3.0
java.awt.Point[x=1,y=1] 1.0
java.awt.Point[x=1,y=3] 2.0
java.awt.Point[x=3,y=1] 4.0
-------------------------------------------
End: Map (x,y) coordinate --> double value

Part 02: Demonstrate that points must be unique ..
Begin: Map (x,y) coordinate --> double value
-------------------------------------------
java.awt.Point[x=4,y=2] 3.0
java.awt.Point[x=1,y=1] 5.0
java.awt.Point[x=1,y=3] 6.0
java.awt.Point[x=3,y=1] 4.0
-------------------------------------------
End: Map (x,y) coordinate --> double value
prompt >>
```

Key points to note:

1. The statement:

```
Map<Point, Double> map = new HashMap<Point, Double>();
```

creates a hashmap that will map keys of type `java.awt.Point` to values of type `Double`. The use of `java.awt.Point` is simply a convenience – you could just as well define a class Coordinate of the type:

```
public class Coordinate {
   double dX, dY;

   .... etc ....
}
```

and use

```
Map<Coordinate, Double> map = new HashMap<Coordinate, Double>();
```

instead.

2. Within the method `printHashMap(...)` the line:

```
System.out.println( key + " " + value.toString() );
```

systematically assembles a character string to be assembled. When the method argument is traversed from left to right, the argument key (of type java.awt.Point) is automatically converted to an entity of type String (i.e., key.toString()) and then concatenated to the second argument. The element `value.toString()` explicitly defines that a string will be generated for the value.

3. HashMap key-value relationships need to be unique. The principal purpose of Part 2 is to test that this requirement is maintained. We systematically attempt to give the key `Point(1,1)` the value 5.0 while maintaining the original value, 1.0. This fails. A similar story applies to the statement:

```
map.put( new Point(1, 3), new Double( 6.0 ) );
```

Example 5: Demonstrate DeepCopy for a HashMap

This example demonstrates the difference between shallow- and deep- copies of a java collection. As previously noted:

1. A **shallow copy** duplicates the structure of the original collection, but not the elements contained in the collection. In other words, a shallow copy replicates the structure and referenes to objects.

As such, in a shallow copy, both the original collection and its copy will share access to the original elements.

2. In a **deep copy**, both the elements and the element structure will be replicated.

Creating a deep copy of a java collection is often a lot easier said than done, since this requires a copy of the basic organizing mechanism (e.g., a map structure or an arraylist structure) plus copies of all of the items that are referenced from the collection.

The following program illustrates the basic difference between a shallow copy and a deep copy, and the consequences of these mechanims when items are added/removed from a collection.

source code

```
/*
 *  ====================================================================
 *  DeepCopyTest.java: Illustrate the difference between shallow copy
 *  and a deep copy of a HashMap.
 *
```

```
 *  Written by: Mark Austin                                    March, 2013
 *  ================================================================
 */

import java.util.HashMap;
import java.util.Iterator;

public class DeepCopyTest {
   public static void main( final String[] args ) {
      testReference();
      testDeepCopy();
   }

   // Deep Copy Test ....

   private static void testDeepCopy() {

      System.out.println( "" );
      System.out.println( "Deep Copy Test" );
      System.out.println( "========================" );

      HashMap<Integer, String> data1 = new HashMap<Integer, String>();
      data1.put( new Integer( "1" ), "thing one" );
      data1.put( new Integer( "2" ), "thing two" );
      data1.put( new Integer( "3" ), "thing three" );

      HashMap<Integer, String> data2 = new HashMap<Integer, String>( data1 );

      printHashMap( data1, "data1" );
      printHashMap( data2, "data2" );

      System.out.println( "Remove data in data2" );

      data1.remove( new Integer( 2 ) );
      data1.put( new Integer( "2" ), "thing two (version2)" );

      printHashMap( data1, "data1" );
      printHashMap( data2, "data2" );

      System.out.println( "========================" );
   }

   // A HashMap is referenced by two variables, data1 and data2....

   private static void testReference() {

      System.out.println( "" );
      System.out.println( "Simple Copy Test" );
      System.out.println( "========================" );

      HashMap<Integer, String> data1 = new HashMap<Integer, String>();
      data1.put( new Integer( "1" ), "one" );
      data1.put( new Integer( "2" ), "two" );

      HashMap<Integer, String> data2 = data1;
```

```
      printHashMap( data1, "data1" );
      printHashMap( data2, "data2" );

      System.out.println( "Remove data in data2" );

      data1.remove( new Integer( 2 ) );
      printHashMap( data1, "data1" );
      printHashMap( data2, "data2" );

      System.out.println( "=======================" );
   }

   private static void printHashMap( HashMap<Integer, String> map, String message ) {
      System.out.println( "---- " + message + " Begin ----" );

      final Iterator iterator = map.keySet().iterator();
      while ( iterator.hasNext() ) {
         final Integer key = (Integer)iterator.next();
         final String value = map.get( key );

         System.out.println( key + " " + value );
      }

      System.out.println( "---- " + message + " End ----" );
   }
}
```

The program output is as follows:

```
Simple Copy Test
=======================
---- data1 Begin ----
1 one
2 two
---- data1 End ----
---- data2 Begin ----
1 one
2 two
---- data2 End ----
Remove data in data2
---- data1 Begin ----
1 one
---- data1 End ----
---- data2 Begin ----
1 one
---- data2 End ----
=======================

Deep Copy Test
=======================
---- data1 Begin ----
1 thing one
2 thing two
```

```
3 thing three
---- data1 End ----
---- data2 Begin ----
1 thing one
2 thing two
3 thing three
---- data2 End ----
Remove data in data2
---- data1 Begin ----
1 thing one
2 thing two (version2)
3 thing three
---- data1 End ----
---- data2 Begin ----
1 thing one
2 thing two
3 thing three
---- data2 End ----
========================
```

Key points to note:

1. Figure 11.11 shows the layout of memory for the shallow copy of the hashmap referenced by `data1`.

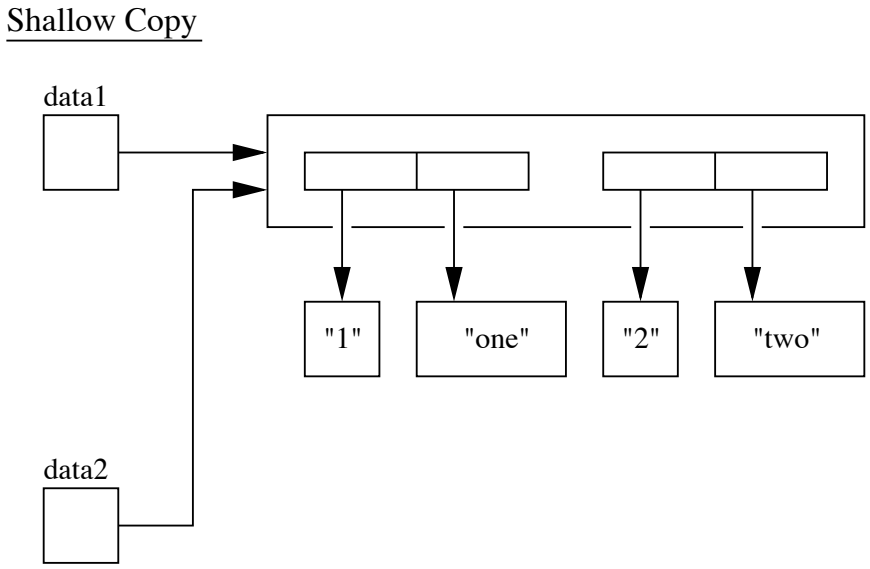

Figure 11.11. Shallow copy of a hashmap and its contents.

In a shallow copy, a copy of the reference structure is made, but not the `(key,value)` items themselves, i.e.,

```
HashMap<Integer, String> data2 = data1;
```

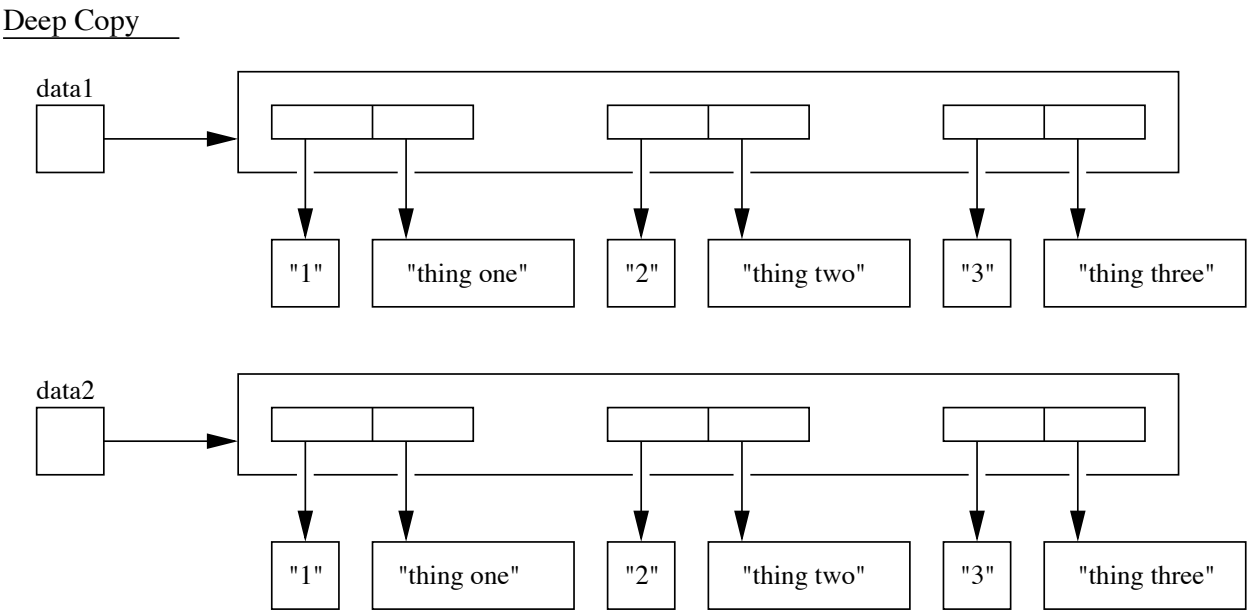

Figure 11.12. Deep copy of a hashmap and its contents.

Hence, in the method `testReference()`, when the reference to `"2"` and `"two"` is removed from one structure, the items are also removed from the second.

2. Figure 11.12 shows the layout of memory for the deep copy of the hashmap referenced by `data1`. The statement:

```
HashMap<Integer, String> data2 = new HashMap<Integer, String>( data1 );
```

will make a deep copy of `HashMap data1` if and only if all of the elements in `data1` have clone methods. Objects of type Integer and String both come with clone methods. This tiny observation is the key to cloning more complicated data structures such as composite hierarchy design patterns.

11.8 Working with Sets

Sets provide ...

... a collection of things containing no duplicates.

The Set interface provides methods for adding individual items to a set (i.e., with the method add (Object o)), a complete collection to a set (i.e., with the method addAll(Collection c)), removing an object from the set (i.e., with remove (Object o)), and tests to see if a set contains a specific object (i.e., with contains (Object o)). The method toArray() converts the contents of a set into an array format.

HashSets. HashSets provide ...

...a hashmap-backed implementation of the Set interface.

Most operations are O(1), assuming no hash collisions. In the worst case (where all hashes collide), operations are O(n). Setting the initial capacity too low will force many resizing operations, but setting the initial capacity too high (or loadfactor too low) leads to wasted memory and slower iteration.

HashSet accepts the null key and null values. It is not synchronized, so if you need multi-threaded access, consider using:

```
Set s = Collections.synchronizedSet( new HashSet(...) );
```

TreeSets. TreeSets provide ...

... a TreeMap-backed implementation of the SortedSet interface.

The elements will be sorted according to their *natural order*, or according to the provided Comparator.

Most operations are O(log n), but there is so much overhead that this makes small sets expensive. Note that the ordering must be *consistent with equals* to correctly implement the Set interface. If this condition is violated, the set is still well-behaved, but you may have suprising results when comparing it to other sets.

TreeMap implementations are not synchronized. If you need to share this between multiple threads, do something like:

```
SortedSet s = Collections.synchronizedSortedSet( new TreeSet(...) );
```

Example 1. Create a HashSet of Strings

In this example we use a hashset to store unique words in a stream of text read from the keyboard (or standard input). The program then prints up to first 20 distinct words.

The source code is as follows:

source code

```
/**
 * ================================================================================
 * TestHashSet.javat: Use a HashSet to print all unique words in System.in.
 *
 * Author: Cay Horstmann
 * ================================================================================
 */

import java.util.*;

public class TestHashSet {
   public static void main(String[] args) {
      Set<String> words = new HashSet<String>(); // HashSet implements Set
      long totalTime = 0;

      // Use a Scanner to read words from standard input.
      // Keep track of the time, measured in milliseconds.

      Scanner in = new Scanner(System.in);
      while (in.hasNext()) {
         String word   = in.next();
         long callTime = System.currentTimeMillis();
         words.add(word);
         callTime = System.currentTimeMillis() - callTime;
         totalTime += callTime;
      }

      // Print up to the first 20 words ....

      System.out.println("Print up to the first 20 distinct words ");
      System.out.println("======================================= ");

      Iterator<String> iter = words.iterator();
      for (int i = 1; i <= 20 && iter.hasNext(); i++)
         System.out.println(iter.next());

      // Print number of distinct words ...

      System.out.println("======================================= ");
      System.out.println(words.size() + " distinct words. " + totalTime + " milliseconds.");
      System.out.println("======================================= ");
   }
}
```

For the input stream (for convenience we put the text in a file called input),

```
Here is the shortest sentence in the english language
that contains all twenty six letters.

The quick brown fox jumps over the lazy dog.

And now for a few more words just to increase the
number of distinct words. dog. dog.  dog. dog. dog.
```

the program input and output is as follows:

```
prompt >>
prompt >> java TestHashSet < input
Print up to the first 20 distinct words
=======================================
to
shortest
fox
for
letters.
quick
sentence
words.
contains
dog.
distinct
six
of
The
over
more
words
Here
all
just
=======================================
36 distinct words. 0 milliseconds.
=======================================
prompt >>
```

Points to note are as follows:

1. Let us assume that the script of text is stored in a file called input.txt. A simple way of executing the program from a command prompt in the terminal window is:

```
prompt >> java TestHashSet < input.txt
```

Here, the unix redirection symbol (<) tells the scanner to read input from input.txt instead of the console.

2. The command:

```
Set<String> words = new HashSet<String>();
```

creates a hashset object to store the words that appear in input.txt. Notice, that we simply record the word and not the number of times it appears in the tezt file.

Example 2. Create Sets of Enumerated Data Types

An enumerated data type is a type whose fields consist of a fixed set of constants. Common examples include compass directions (e.g., North, South, East, and West), days of the week, months of the year, sexes of organisms, letter grades in a course, and so forth. A key benefit in the use of enumerated data type include enhanced readability of code – typical uses include assignment statements, comparison expressions, iteration, selection with switch statements, and in java collections such as maps and sets.

In this example we create sets of enumerated data types representing various days of the week. The source code for days of the week is as follows:

source code

```
/*
 *  ========================================================
 *  Day.java: Enumerated data type for days of the week.
 *  ========================================================
 */

public enum Day {
  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday;
}
```

Now we can sets of enumerated data types for “days I go into work” and “days I don’t go into work,” i.e.,

source code

```
/*
 *  ============================================================
 *  TestDaysSet.java: Create sets for days of the week.
 *  ============================================================
 */

import java.util.EnumSet;
import java.util.Iterator;

public class TestDaysSet {
    public static void main(String[] args) {
        EnumSet weekDays = EnumSet.of(   Day.Monday, Day.Tuesday, Day.Wednesday,
                                         Day.Thursday, Day.Friday );
        EnumSet weekEnd  = EnumSet.of( Day.Saturday, Day.Sunday );

        System.out.println("I go into work during the week!");
        System.out.println("==============================");

        for( Iterator it = weekDays.iterator(); it.hasNext(); ) {
            Day day = (Day) it.next();
            System.out.println(day);
        }
```

```
        System.out.println("");
        System.out.println("And stay home on the weekend!");
        System.out.println("==============================");

        for( Iterator it = weekEnd.iterator(); it.hasNext(); ) {
            Day day = (Day) it.next();
            System.out.println(day);
        }

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

The program output is as follows:

```
prompt >> java TestDaysSet
I go into work during the week!
==============================
Monday
Tuesday
Wednesday
Thursday
Friday

And stay home on the weekend!
==============================
Sunday
Saturday
==============================
prompt >>
prompt >>
```

Points to note:

1. An EnumSet is a specialized Set implementation for use with enum types. All of the elements in an enum set must come from a single enum type that is specified (e.g., Day), explicitly or implicitly, when the set is created. An iterator traverses the set elements in their natural order.

2. Enum sets are represented internally as bit vectors. This representation is extremely compact and efficient. The space and time performance of this class should be good enough to allow its use as a high-quality, typesafe alternative to traditional int-based ”bit flags.” Even bulk operations (such as containsAll and retainAll) should run very quickly if the specified collection is also an enum set.

11.9 Modeling Association Relationships

When we model a system, certain concepts will be related to one another, and these relationships need to be modeled. An association represents the ...

... static relationship shared among the objects of two classes.

Binary associations (with two ends) are normally represented as a line. An association defines the multiplicity between two objects, e.g., one-to-one, one-to-many, many-to-one, and many-to-many. Associations can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties.

Uni-Directional Association

Definition. In a uni-directional association, two classes are related, but only one class knows that the relationship exists.

Example. In this example, we model the uni-directional association between a customer and a book. The customer owns a book, but the book is not aware of the customer, i.e.,

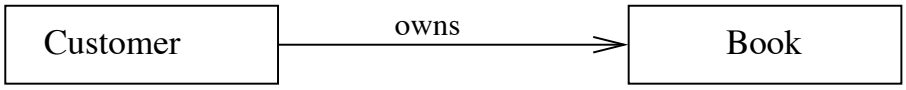

Figure 11.13. Uni-directional relationship between a customer and book.

Here we say that the customer owns a book, but do not say that a book is owned by a customer. Hence, only the owning side of the relationship (i.e., the customer) determines the updates between a customer and book.

The details of Book.java and Customer.java are as follows:

source code

```
/*
 *  ============================================================
 *  Book.java: A simple book object ...
 *  ============================================================
 */

public class Book {
    // Data attributes

    private String   name;
    private String author;

    // Constructor

    public Book () {}
```

```
    // String representations  ...

    public String toString() {
       String s = "Book name: " + getName()   + "\n" +
                  "     author: " + getAuthor() + "\n";

       return s;
    }

    // Set and access attributes

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
}
```

and

source code

```
/*
 *  ================================================================
 *  Customer.java: A customer object has an association with a book.
 *  ================================================================
 */

public class Customer {
    // Data attributes

    private String name;

    // String representations  ...

    public String toString() {
       String s = "Customer: " + getName() + "\n";
       return s;
    }

    // Association attributes

    public Book book;
```

```
    // Attribute accessors

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    // Association accessors

    public Book getBook() {
        return book;
    }

    public void setBook(Book book) {
        this.book = book;
    }
}
```

The file TestRelationship.java assembles Customer and Book objects and assembles the uni-directional association relationship, i.e.,

source code

```
/*
 *  ========================================================================
 *  TestRelationship: Create a undirectional relationship between a customer
 *  and a book object.
 *
 *  Written by: Mark Austin                                    December 2009
 *  ========================================================================
 */

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

      // Create book and customer objects.

      Book book = new Book();
      book.setName("The Cat in the Hat");
      book.setAuthor("Dr Seuss");

      Customer customer = new Customer();
      customer.setName("Angela Austin");

      // Create unidirectional customer-book relationship.

      customer.setBook( book );

      // Retrieve and print customer-book relationship.
```

```
      System.out.println ( customer.toString() );
      System.out.println ( customer.getBook().toString() );
   }
}
```

The test program input/output is as follows:

```
prompt >> java TestRelationship
Customer: Angela Austin

Book name: The Cat in the Hat
     author: Dr Seuss

prompt >>
```

The important point with this model is that ...

... the customer refers to a book, but a book does refer to a customer.

Bi-Directional Association

Definition. By default, associations are assumed to be bi-directional. This means that ...

... both classes are aware of each other and their relationship.

Example. Let's recode the customer-book association so that the customer owns a book and the book is owned by a customer, i.e.,

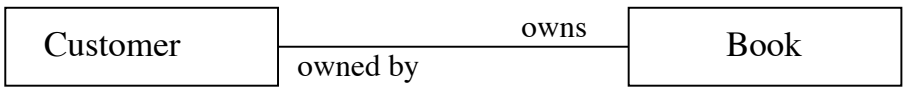

Figure 11.14. Bi-directional relationship between a customer and book.

The modified details of Book.java and Customer.java are as follows:

source code

```
/*
 *  ================================================================
 *  Book.java: Create a book object with an association to a customer
 *  ================================================================
 */

public class Book {
    // Data attributes
```

```
    private String   name;
    private String author;

    // Set relationship ....

    private Customer owner;

    // Constructor

    public Book () {
       this.owner = null;
    }

    // String representations  ...

    public String toString() {
       String s = "Book   name: " + getName()   + "\n" +
                  "     author: " + getAuthor() + "\n";

       if (this.owner != null )
           s += "   owned by: " + owner.getName() + "\n";

       return s;
    }

    // Set and access attributes

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    // Association accessor methods ...

    public Customer getCustomer() {
        return owner;
    }

    public void setCustomer( Customer owner ) {
        if (owner != null)
            this.owner = owner;
    }
}
```

and

source code

```
/*
 *  ========================================================================
 *  Customer.java: Create a Customer object with an association to a book.
 *  ========================================================================
 */

public class Customer {
    private String name;         // Data attributes
    public Book book;            // Association attributes

    // Constructors ....

    public Customer () {
       this.book = null;
    }

    // String representation ...

    public String toString() {
       String s = "";

       if (this.book != null )
           s += "Customer: " + getName() + " owns " + book.getName() + "\n";
       else
           s += "Customer: " + getName() + " doesn't own a book \n";

       return s;
    }

    // Attribute accessors

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    // Association accessors

    public Book getBook() {
        return book;
    }

    public void setBook( Book book ) {
        if (book != null)
            this.book = book;
    }
```

```
}
```

The file TestRelationship.java assembles Customer and Book objects and assembles the bi-directional association relationship. Details of the association relationship are then printed from the customer and book perspectives.

source code

```
/*
 *  ========================================================================
 *  TestRelationship: Create a bi-directional relationship between a customer
 *  and a book object.
 *
 *  Written by: Mark Austin                                    December 2011
 *  ========================================================================
 */

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

      // Create book and customer objects.

      Book book = new Book();
      book.setName("The Cat in the Hat");
      book.setAuthor("Dr Seuss");

      Customer owner = new Customer();
      owner.setName("Angela Austin");

      // Create a bidirectional customer-book association.

      owner.setBook( book );
      book.setCustomer( owner );

      // Retrieve and print owner-book relationship.

      System.out.println ( "Retrieve and print owner-book relationship" );
      System.out.println ( "=======================================" );

      System.out.println ( owner.toString() );

      // Retrieve and print book-owner relationship.

      System.out.println ( "Retrieve and print book-owner relationship" );
      System.out.println ( "=======================================" );

      System.out.println (  book.toString() );
   }
}
```

The test program input/output is as follows:

```
prompt >> java TestRelationship
Retrieve and print owner-book relationship
==========================================
Customer: Angela Austin owns The Cat in the Hat

Retrieve and print book-owner relationship
==========================================
Book   name: The Cat in the Hat
     author: Dr Seuss
   owned by: Angela Austin

prompt >>
```

One-to-Many Associations

Definition. In a one-to-many relationship, ...

> **... a single object can be related to many relating objects.**

In a one-to-many relationship between classes A and B, each object of type A is linked to 0, 1 or many instances of object B. For example, if A and B represent company and employees, generally speaking, a specific company will have many employees.

Example. In this example we assemble a one-to-many association relationship between an academic department (CEE) and five students enrolled in the department.

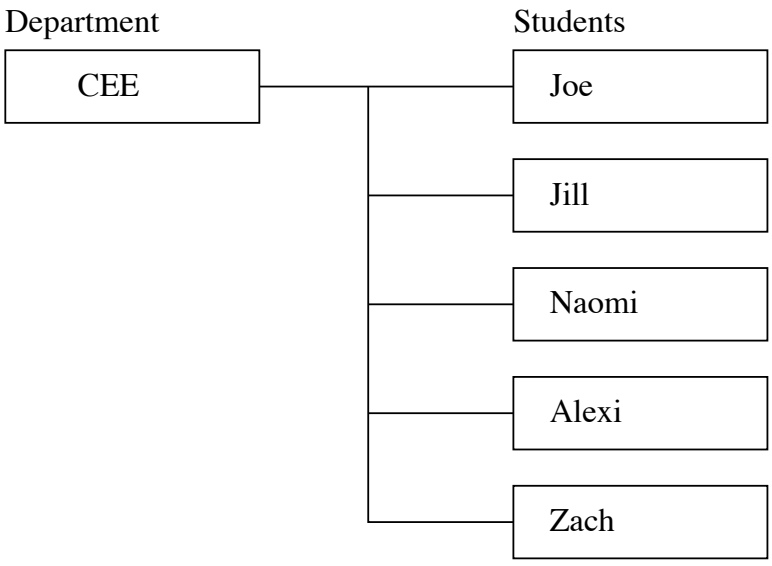

Figure 11.15. One-to-many relationship between an academic department and students.

The source code is organized into three files: Student.java, Department.java and Main.java. The details of Student.java are as follows:

source code

```
/*
 *  ================================================================
 *  Student.java: Create student objects ...
 *  ================================================================
 */

public class Student {
    private int id;
    private String name;

    private Department department;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department) {
        this.department = department;
    }

    public String toString() {
       String s = "Student: " + getName() + " id: " + getId() + "\n";
       if ( getDepartment() != null )
          s = s + "Department: " + getDepartment() + "\n";

       return s;
    }
}
```

Notice that each student will be associated with one, and only one department. Set() and get() methods are provided to manage the student-department relationship.

A department is modeled with a name, id and a Map to a collection of students, i.e.,

source code

```
/*
 *  ================================================================
 *  Department.java: Create academic department objects ...
 *  ================================================================
 */

import java.util.Map;
import java.util.HashMap;

public class Department {
    private int id;
    private String name;

    private Map<String, Student> students;

    public Department() {
       students = new HashMap<String, Student>();
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String deptName) {
        this.name = deptName;
    }

    public void addStudent(Student student) {
        if (!getStudents().containsKey(student.getName())) {
            getStudents().put(student.getName(), student);
            if (student.getDepartment() != null) {
                student.getDepartment().getStudents().remove(student.getName());
            }
            student.setDepartment(this);
        }
    }

    public Map<String,Student> getStudents() {
        return students;
    }

    public String toString() {
        return "Department id: " + getId() + ", name: " + getName();
    }
}
```

In Main.java we systematically assemble the one-to-many relationship graph illustrated in Figure 11.15, i.e.,

source code

```
/*
 *  ====================================================================
 *  Main.java: Systematically assemble a one-to-many relationship graph
 *             between an academic department and students.
 *  ====================================================================
 */

import java.util.Map;
import java.util.HashMap;
import java.util.TreeMap;

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

    // Create student objects ...

    Student student01 = new Student();
    student01.setName("Joe");
    student01.setId( 001 );
    Student student02 = new Student();
    student02.setName("Jill");
    student02.setId( 002 );
    Student student03 = new Student();
    student03.setName("Naomi");
    student03.setId( 003 );
    Student student04 = new Student();
    student04.setName("Alexi");
    student04.setId( 004 );
    Student student05 = new Student();
    student05.setName("Zack");
    student05.setId( 005 );

    // Print details of individual students (not yet enrolled in a department) ...

    System.out.println("List of students (not yet enrolled)");
    System.out.println("===================================");

    System.out.println ( student01 );
    System.out.println ( student02 );
    System.out.println ( student03 );
    System.out.println ( student04 );
    System.out.println ( student05 );

    // Create department object ...

    Department dept = new Department();
    dept.setName( "CEE" );
    dept.setId( 004 );

    // Add students to department ...
```

```
    dept.addStudent( student01 );
    dept.addStudent( student02 );
    dept.addStudent( student03 );
    dept.addStudent( student04 );
    dept.addStudent( student05 );

    System.out.println("List of students enrolled in CEE   ");
    System.out.println("==================================");

    Map students = dept.getStudents();
    System.out.println ( students );

    // Organize students into a tree map ...

    System.out.println("Ordered list of students enrolled in CEE   ");
    System.out.println("==========================================");

    Map organizedStudents = new TreeMap( students );
    System.out.println ( organizedStudents );
  }
}
```

The (slightly edited) program input/output is:

```
prompt >> java Main
List of students (not yet enrolled)
==================================
Student: Joe id: 1
Student: Jill id: 2
Student: Naomi id: 3
Student: Alexi id: 4
Student: Zack id: 5

List of students enrolled in CEE
==================================
{  Naomi=Student: Naomi id: 3 Department: Department id: 4, name: CEE,
   Zack=Student: Zack id: 5 Department: Department id: 4, name: CEE,
   Alexi=Student: Alexi id: 4 Department: Department id: 4, name: CEE,
   Jill=Student: Jill id: 2 Department: Department id: 4, name: CEE,
   Joe=Student: Joe id: 1 Department: Department id: 4, name: CEE
}
Ordered list of students enrolled in CEE
==========================================
{  Alexi=Student: Alexi id: 4 Department: Department id: 4, name: CEE,
   Jill=Student: Jill id: 2 Department: Department id: 4, name: CEE,
   Joe=Student: Joe id: 1 Department: Department id: 4, name: CEE,
   Naomi=Student: Naomi id: 3 Department: Department id: 4, name: CEE,
   Zack=Student: Zack id: 5 Department: Department id: 4, name: CEE
}
prompt >>
```

Many-to-Many Associations

Definition. The many-to-many relationship between classes A and B exists when ...

... multiple objects of type A associated with multiple objects of type B, and vise versa.

Here are a few examples:

1. In most schools each teacher teaches multiple students and each student can be taught by multiple teachers.
2. An author can write several books, and a book can be written by several authors.

Example. We now extend the one-to-many example, and assume that students may be enrolled in multiple departments.

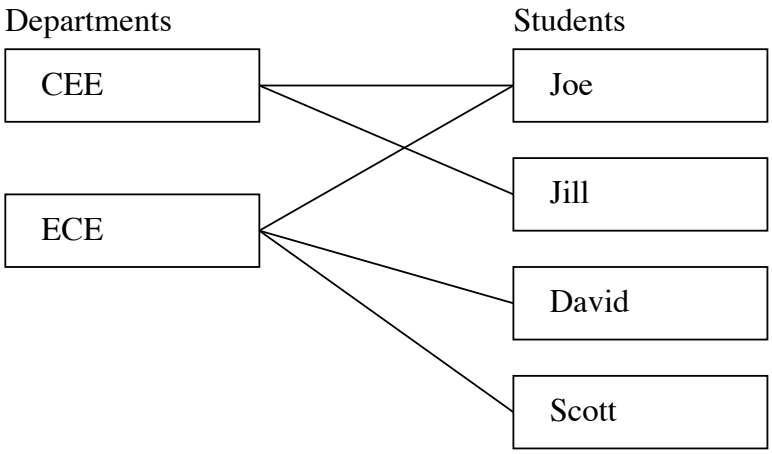

Figure 11.16. Many-to-many association relationships among academic departments and students.

The following scripts of code assemble the many-to-many relationship between students and departments illustrated in Figure 11.16. The source code is organized into three files: Student.java, Department.java, and SimSchool.java.

The details of Student.java are as follows:

source code

```
/*
 *  ========================================================
 *  Student.java; Create student objects ...
 *  ========================================================
 */

import java.util.Collection;
import java.util.ArrayList;
import java.util.Iterator;
```

```
public class Student {
   private int id;
   private String name;

   // Collection of departments in which the student enrols.

   private Collection<Department> departments;

   // Constuctor method ...

   public Student() {
      departments = new ArrayList<Department>();
   }

   // Setup student Ids ...

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   // Set student name ...

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   // Student enrols in a department ...

   public void addDepartment(Department department) {

     // Add new department to students resume ..

     if ( getDepartments().contains(department) == false ) {
          getDepartments().add(department);
     }

     // Synchronize with list of students enrolled in the department ..

     if ( department.getStudents().contains(this) == false ) {
          department.getStudents().add(this);
     }
   }

   // Student drops-out of a department ...

   public void removeDepartment(Department department) {
```

```
      // Remove department from students resume ..

      if ( getDepartments().contains(department) == true ) {
           getDepartments().remove(department);
      }

      // Synchronize with list of students enrolled in the department ..

      if ( department.getStudents().contains(this) == true ) {
           department.getStudents().remove(this);
      }
   }

   // Return collection of departments ...

   public Collection<Department> getDepartments() {
      return departments;
   }

   public void setDepartment( Collection <Department> departments) {
      this.departments = departments;
   }

   // Create String representation of student object ...

   public String toString() {
      String s = "Student: " + name + "\n";

      s = s + "Departments: ";
      if ( departments.size() == 0 )
         s = s + " none \n";
      else {
         Iterator iterator1 = departments.iterator();
         while ( iterator1.hasNext() != false ) {
            Department dept = (Department) iterator1.next();
            s = s + dept.getName() + " ";
         }
         s = s + "\n";
      }

      return s;
   }
}
```

The details of Department.java are as follows:

source code

```
/*
 *  ================================================================
 *  Department.java; Create simple model of an academic department.
 *  ================================================================
```

```
 */

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class Department {
   private int id;
   private String name;

   // Setup collection of students ...

   private Collection<Student> students;

   // Constructor method ....

   public Department(){
      students = new ArrayList<Student>();
   }

   // Set/get the department Id.

   public int getId() {
      return id;
   }

   public void setId(int id) {
        this.id = id;
   }

   // Methods to deal with the department name ...

   public String getName() {
        return name;
   }

   public void setName(String deptName) {
        this.name = deptName;
   }

   // Add a student to department ....

   public void addStudent(Student student) {

      if ( getStudents().contains(student) == false ) {
           getStudents().add(student);
      }

      // Synchronize with departments on the student side ....

      if ( student.getDepartments().contains(this) == false ) {
           student.getDepartments().add(this);
      }
   }
```

```
   public Collection<Student> getStudents() {
      return students;
   }

   public void setStudent( Collection<Student> students ) {
      this.students = students;
   }

   // Create a String representation for the department ...

   public String toString() {
      String s = "Department: " + name + "\n";

      s = s + "Students: ";
      if ( students.size() == 0 )
         s = s + " none \n";
      else {
         Iterator iterator1 = students.iterator();
         while ( iterator1.hasNext() != false ) {
            Student student = (Student) iterator1.next();
            s = s + student.getName() + " ";
         }
         s = s + "\n";
      }

      return s;
   }
}
```

The graph of many-to-many relationships is assembled in SimSchool.java, i.e.,

source code

```
/*
 *  ====================================================================
 *  SimSchool.java: Simulate many-to-many relationships between students
 *                  and the departments in which they enroll.
 *  ====================================================================
 */

import java.util.List;

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

      // Create student objects ...

      Student student01 = new Student();
      student01.setName("Joe");

      Student student02 = new Student();
      student02.setName("Jill");
```

```
        Student student03 = new Student();
        student03.setName("David");

        Student student04 = new Student();
        student04.setName("Scott");

        // Add students to civil and environmental engineering ...

        Department civil = new Department();
        civil.setName("CEE");
        civil.addStudent( student01 );
        civil.addStudent( student02 );

        // Add students to electrical and computer engineering ...

        Department eecs = new Department();
        eecs.setName("ECE");
        eecs.addStudent( student01 );
        eecs.addStudent( student03 );
        eecs.addStudent( student04 );

        // Print details of student-department assocations ...

        System.out.println( "Part 1: Summary of Student-Department Associations" );
        System.out.println( "================================================" );

        System.out.println( student01 );
        System.out.println( student02 );
        System.out.println( student03 );
        System.out.println( student04 );

        // Print details of department-student assocations ...

        System.out.println( "Part 2: Summary of Department-Student Associations" );
        System.out.println( "================================================" );

        System.out.println( civil );
        System.out.println( eecs );

        // David drops out of ECE to concentrate on CEE ...

        System.out.println( "Part 3: David switches from ECE to CEE" );
        System.out.println( "============================================" );

        student03.removeDepartment( eecs );
        student03.addDepartment( civil );

        // Validate David's enrollment in CEE and ECE ...

        System.out.println( student03 );
        System.out.println( civil );
        System.out.println( eecs );
    }
}
```

The (slightly edited) program input/output is:

```
prompt >> java SimSchool
Part 1: Summary of Student-Department Associations
================================================
Student: Joe
Departments: CEE ECE

Student: Jill
Departments: CEE

Student: David
Departments: ECE

Student: Scott
Departments: ECE

Part 2: Summary of Department-Student Associations
================================================
Department: CEE
Students: Joe Jill

Department: ECE
Students: Joe David Scott

Part 3: David switches from ECE to CEE
============================================
Student: David
Departments: CEE

Department: CEE
Students: Joe Jill David

Department: ECE
Students: Joe Scott

prompt >>
```

A few key points:

1. Figure 11.17 shows the relationship among classes needed to support the modeling of many-to-many relationships.

Both the Student and Depatment classes employ Arraylists for the storage of their counterpart associations. On the student side we have:

```
private Collection<Department> departments;

public Student() {
   departments = new ArrayList<Department>();
}
```

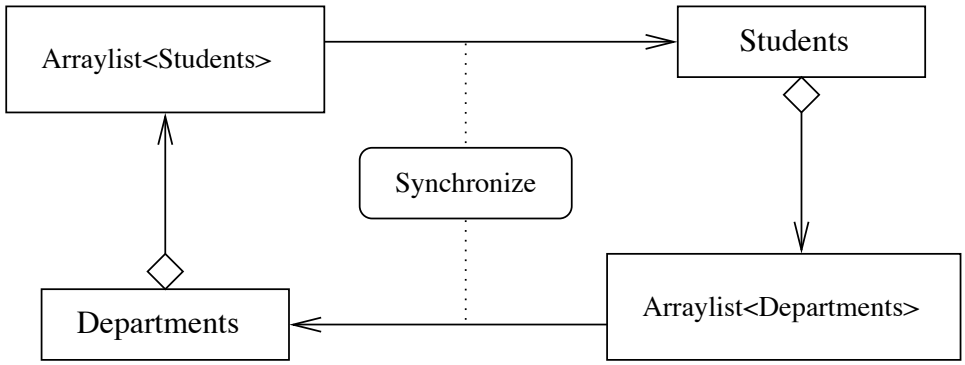

Figure 11.17. Relationship among classes in modeling of many-to-many association relationships.

And on the department side we have:

```
private Collection<Student> students;

public Department(){
   students = new ArrayList<Student>();
}
```

HashMaps of the form:

```
private Collection<String,Department>
        departments = new HashMap<String,Department>();
private Collection<String,Student>
        students = new HashMap<String,Student>();
```

would also work as well. The interesting part of the code is in the methods `addStudent()`, `removeStudent()`,`addDepartment()`,and `removeDepartment()` where code is written to synchronize association relationships from both the department and student viewpoints.

2. SimSchool assembles the graph of Students and Depatmental associations shown in Figure 11.15. The simulation presents summaries of student-department and department-student associations.

Then, student David switches from department ECE to department CEE. This action is handled by the pair of method calls:

```
student03.removeDepartment( eecs );
  student03.addDepartment( civil );
```

Of course, a department might also decide to terminate a students enrollment, in which case the method call would be something like:

```
civil.removeStudent( student03 );
```

11.10 Working with Java Generics

As we have seen, a Java collection is a flexible data structure that can hold heterogeneous objects where the elements may have any reference type. It is ...

> **... your responsibility, however, to keep track of what types of objects your collections contain.**

Consider, for example, the task of adding a double to a collection.

- Since you cannot have collections of primitive data types, you must convert the double to the corresponding reference type (i.e., Double) before storing it in the collection.
- Then when the element is extracted from the collection, an Object is returned that must be cast to an Double in order to ensure type safety.
- If the programmer accidentally makes an error (e.g., casts the returned value to a String), then a run-time error will occur.

The manual overhead in ensuring type safety makes ...

> **... this aspect of Java programming more difficult than it needs to be.**

To address this problem, J2SE 5.0 has added a new core language feature known as generics (also known as parameterized types), that provides ...

> **... compile-time type safety for collections and eliminate the drudgery of casting.**

Introduction to Programming with Generics

Broadly speaking generic programming is ...

> **... a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters.**

Generic programming techniques date back the the early 1980s, and the development of Ada. From a software development standpoint, having an ability to ...

> **... write common functions or types that differ only in the set of types on which they operate when used,**

reduces the need for duplication.

Software entities created using generic programming are known as:

1. Tenerics in Ada, Eiffel, Java, C#, and Visual Basic .NET.,
2. Templates in C++. and,
3. Parameterized types in the influential 1994 book Design Patterns.

Use of Generics in Java

Support for the generics, or containers-of-type-T, subset of generic programming were ...

... added to the Java programming language in 2004 as part of J2SE 5.0.

In Java, generics are checked at compile time for type correctness. The generic type information is then removed via a process called type erasure, and is unavailable at runtime.

For example, a List<String> is converted to the raw type List. The compiler inserts type casts to convert the elements to the String type when they are retrieved from the list.

Using generics, a collection is no longer treated as a list of Object references, but you would be able to differentiate between a collection of references to Integers and collection of references to Bytes. A collection with a generic type has a type parameter that specifies the element type to be stored in the collection.

Example 1. Use of Generics in a LinkedList

Consider the following segment of code that creates a linked list and adds an element to the list (source: http://www.oracle.com/technetwork/articles/javase/generics-136597.html):

```
LinkedList list = new LinkedList();
list.add(new Integer(1));
Integer num = (Integer) list.get(0);
```

When an element is extracted from the list it must be cast. The casting is safe as it will be checked at runtime, but if you cast to a type that is different from, and not a supertype of, the extracted type then a runtime exception, ClassCastException will be thrown.

Using generic types, the previous segment of code can be written as follows:

```
LinkedList<Integer> list = new LinkedList<Integer>();
list.add(new Integer(1));
Integer num = list.get(0);
```

Here we say that LinkedList is a generic class that takes a type parameter, Integer in this case.

The benefit in using generics is that you no longer need to cast to an Integer since the get() method would return a reference to an object of a specific type (Integer in this case). If you attempt to assign an extracted element to a different type, then ...

... the error would be at compile-time instead of run-time.

This early static checking increases the type safety of the Java language.

To reduce the clutter, the above example can be rewritten as follows...using autoboxing:

```
LinkedList<Integer> list = new LinkedList<Integer>();
list.add(1);
int num = list.get(0);
```

Example 2: Avoiding Run-Time Failure of an ArrayList

Consider the following class, Ex1, which creates a collection of two Strings and one Integer, and then attempts to print out the collection:

source code

```
/*
 *  ================================================================
 *  Ex1.java
 *  ================================================================
 */

import java.util.*;

public class Ex1 {

   private void testCollection() {
      List list = new ArrayList();
      list.add(new String("Hello world!"));
      list.add(new String("Good bye!"));
      list.add(new Integer(95));
      printCollection(list);
   }

   private void printCollection(Collection c) {
      Iterator i = c.iterator();
      while(i.hasNext()) {
         String item = (String) i.next();
         System.out.println("Item: "+item);
      }
   }

   public static void main(String argv[]) {
      Ex1 e = new Ex1();
      e.testCollection();
   }
}
```

Notice that an explicit cast is required in the printCollection method. This class compiles fine, but throws a CLassCastException at runtime as it attempts to cast an Integer to a String, i.e.,

```
Item: Hello world!
Item: Good bye!
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer
        at Ex1.printCollection(Ex1.java:16)
        at Ex1.testCollection(Ex1.java:10)
        at Ex1.main(Ex1.java:23)
```

Now let's fix the problem by adding generics to the array list.

source code

```
/*
 *  ====================================================================
 *  Ex2.java
 *  ====================================================================
 */

import java.util.*;

public class Ex2 {

   private void testCollection() {
      List<String> list = new ArrayList<String>();
      list.add( new String("Hello world!") );
      list.add( new String("Good bye!") );
      list.add( new Integer(95) );
      printCollection(list);
   }

   private void printCollection(Collection c) {
      Iterator<String> i = c.iterator();
      while(i.hasNext()) {
         System.out.println("Item: "+i.next());
      }
   }

   public static void main(String argv[]) {
      Ex2 e = new Ex2();
      e.testCollection();
   }
}
```

Now, if you try to compile this code, ...

> **... a compile-time error will be produced informing you that you cannot add an Integer to a collection of Strings.**

So we see that use of generics enables more compile-time type checking, thereby reducing the likelihood of having to deal with run-time errors.

You may have already noticed the new syntax used to create an instance of ArrayList,i.e.,

```
List<String> list = new ArrayList<String>();
```

ArrayList is now a parameterized type.

Working with Parameterized Types

A parameterized type consists of a class or interface name E and a parameter section

```
<T1, T2, ..., Tn>,
```

which must match the number of declared parameters of E, and each actual parameter must be a subtype of the formal parameter's bound types. The following segment of code shows parts of the new class definition for ArrayList:

```
public class ArrayList<E> extends AbstractList<E> implements List<E>,
   RandomAccess, Cloneable, Serializable {
   // ...
}
```

Here E is a type variable, which is an unqualified identifier. It simply acts as a placeholder for a type to be defined when the list is used.

Implementing Generic Types

In addition to using generic types, you can implement your own. A generic type has one or more type parameters. Here is an example with only one type parameter called E. A parameterized type must be a reference type, and therefore primitive types are not allowed to be parameterized types.

```
interface List<E> {
   void add(E x);
   Iterator<E> iterator();
}

interface Iterator<E> {
   E next();
   boolean hasNext();
}

public class LinkedList<E> implements List<E> {
   // implementation
}
```

Here, E represents the type of elements contained in the collection. Think of E as a placeholder that will be replaced by a concrete type. For example, if you write ...

```
LinkedList<String>
```

then E will be replaced by String.

In some of your code you may need to invoke methods of the element type, such as Object's hashCode() and equals(). Here is an example that takes two type parameters:

```
class HashMap<K, V> extends AbstractMap<K, V> implements Map<K, V> {

   // ...

   public V get(Object k) {
      ...
      int hash = k.hashCode();
      ...
   }
   // ...
}
```

The important thing to note is that you are required to replace the type variables K and V by concrete types that are subtypes of Object.

Working with Generic Methods

Genericity is not limited to classes and interfaces, you can define generic methods. Static methods, nonstatic methods, and constructors can all be parameterized in almost the same way as for classes and interfaces, but the syntax is a bit different. Generic methods are also invoked in the same way as non-generic methods.

Before we see an example of a generics method, consider the following segment of code that prints out all the elements in a collection:

```
public void printCollection(Collection c) {
   Iterator i = c.iterator();
   for(int k = 0;k<c.size();k++) {
      System.out.println(i.next());
   }
}
```

Using generics, this can be re-written as follows. Note that the Collection<?> is the collection of an unknown type.

```
void printCollection( Collection<?> c ) {
   for(Object o:c) {
      System.out.println(o);
   }
}
```

This example uses a feature of generics known as wildcards.

Working with Wildcards

There are three types of wildcards:

1. "? extends Type": Denotes a family of subtypes of type Type. This is the most useful wildcard
2. "? super Type": Denotes a family of supertypes of type Type

3. "?": Denotes the set of all types or any

As an example of using wildcards, consider a `draw()` method that should be capable of drawing any shape such as circle, rectangle, and triangle. The implementation may look something like this. Here Shape is an abstract class with three subclasses: Circle, Rectangle, and Triangle.

```
public void draw(List<Shape> shape) {
   for(Shape s: shape) {
      s.draw(this);
   }
}
```

It is worth noting that the draw() method can only be called on lists of Shape and cannot be called on a list of Circle, Rectangle, and Triangle for example.

In order to have the method accept any kind of shape, it should be written as follows:

```
public void draw(List<? extends Shape> shape) {
   // rest of the code is the same
}
```

Here is another example of a generics method that uses wildcards to sort a list into ascending order. Basically, all elements in the list must implement the Comparable interface.

```
public static <T extends Comparable<? super T>> void sort(List<T> list) {
   Object a[] = list.toArray();
   Arrays.sort(a);
   ListIterator<T> i = list.listIterator();
   for(int j=0; j<a.length; j++) {
      i.index();
      i.set( (T)a[j] );
   }
}
```

11.11 Exercises

11.1 The left-hand side of Figure 11.18 shows the essential details of a domain familiar to many children. One by one, rectangular blocks are stacked as high as possible until they come tumbling down – the goal, afterall, is to create a spectacular crash!!

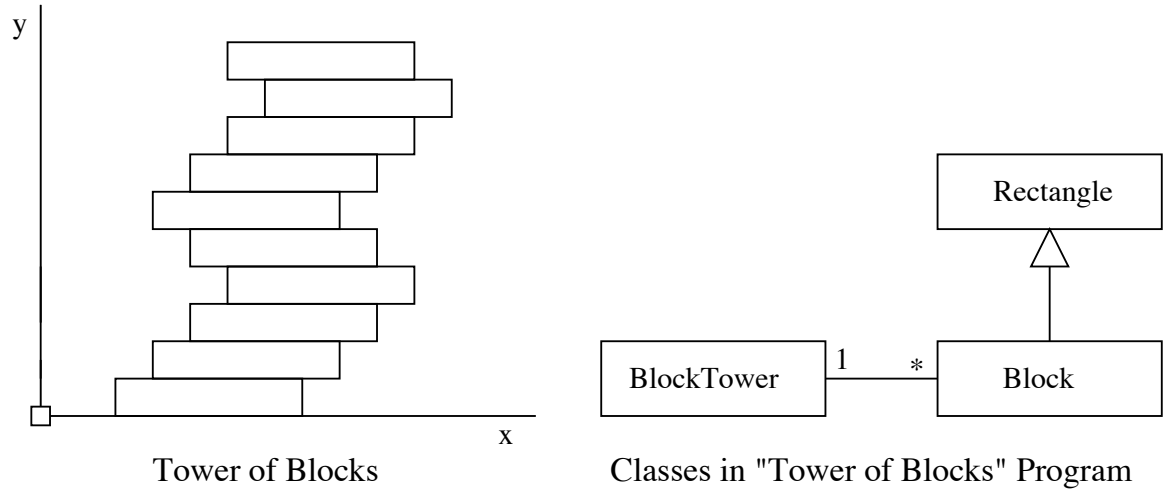

Figure 11.18. Schematic and classes for “Tower of Blocks”

Suppose that we wanted to model this process and use engineering principles to predict incipient instability of the block tower. Consider the following observations:

1. Rather than start from scratch, it would make sense to create a Block class that inherits the properties of Rectangle, and adds details relevant to engineering analysis (e.g., the density of the block).
2. Then we could develop a BlockTower class that systematically assembles the tower, starting at the base and working upwards. At each step of the tower assembly, analysis procedures should make sure that the tower is still stable.

The right-hand side of Figure 11.18 shows the relatioship among the classes. One BlockTower program (1) will employ many blocks, as indicated by the asterik (*).

Develop a Java program that builds upon the Rectangle class written in the previous questions. The class Block should store the density of the block (this will be important in determining its weight) and methods to compute the weight and centroid of each block. The BlockTower class will use block objects to build the tower. A straight forward way of modeling the block tower is with an ArrayList. After each block is added, the program should conduct a stability check. If the system is still stable, then add another block should be added. The simulation should cease when the tower of blocks eventually becomes unstable.

`Note.` To simplify the analysis, assume that adjacent blocks are firmly connected.

`Stability Considerations.` If the blocks are stacked perfectly on top of each other, then from a mathematical standpoint the tower will never become unstable. In practice, this never happens. There is always a small offset and, eventually, it's the accumulation of offsets that leads to spectacular disaster.

For the purposes of this question, assume that blocks are five units wide and one unit high. When a new block is added, the block offset should be one unit. To make the question interesting, assume that four blocks are stacked with an offset to the right, then three blocks are added with an offset to the left, then four to the right, three to the left, and so forth. This sequence can be accomplished with the looping construct:

```
offset = (int) ( Math.floor ((BlockNo - 1)/5.0) + (BlockNo-1)%5 );
if ((BlockNo-1)%5 == 4 ) offset = offset - 2;
```

The tower will become unstable when the center of gravity of blocks above a particular level falls outside the edge of the supporting block.

11.2 This problem will give you practice at using abstract classes to and array lists simplify the implementation of engineering property (e.g., position of the centroid, moments of inertia, orientation of the principal axes) computations for element cross sections. The computation of these properties can be complicated by irregular section shapes and/or cross section shapes that change as a function of loading (e.g., crack patterns in a concrete beam).

Engineering Property Formulae. If the total number of shapes is denoted by N, then the total area of the grid, A, is given by

$$A = \sum_{i=1}^{N} A_i \tag{11.1}$$

The (x,y) coordinates of the grid centroid are defined by:

$$A\bar{x} = \sum_{i=1}^{N} x_i \cdot A_i \quad \text{and} \quad A\bar{y} = \sum_{i=1}^{N} y_i \cdot A_i \tag{11.2}$$

The area moments of inertia about the x- and y-axes are given by:

$$I_{xx} = \sum_{i=1}^{N} y_i^2 \cdot A_i \quad \text{and} \quad I_{yy} = \sum_{i=1}^{N} x_i^2 \cdot A_i \tag{11.3}$$

respectively. Similarly the cross moment of inertia is given by

$$I_{xy} = \sum_{i=1}^{N} x_i \cdot y_i \cdot A_i \tag{11.4}$$

The corresponding moments of inertia about the centroid are given by the parallel axes theorem. Finally, the orientation of the principle axes are given by

$$\tan(2\theta) = \left[\frac{2I_{xy}}{I_{xx} - I_{yy}}\right] \tag{11.5}$$

Things to do.

The computation of engineering properties for this spatial layout can be simplified if the basic algorithms for area, centroid, and inertia calculations are specified in terms of shapes. Java will take care of the details of calling the appropriate methods within each specific shape object.

1. Download, compile and run the abstract shape example (e.g., Shape.java, Location.java, TestShape.java) from the java examples page. Then download, compile and run the Triangle code from the java examples web page.

2. The computation of engineering properties depends on quantities such as the cross section area and centroid (i.e., (x,y) location). An algorithm will need to retrieve this information from each of the object types.

Extend the abstract shape class so that methods for retrieving the x and y coordinates are included. I suggest that you simply add the method declarations:

```
public abstract double getX();
public abstract double getY();
```

to Shape.java and then add concrete implementations of of the methods `getX()` and `getY()` to Circle.java, Rectangle.java and Triangle.java.

3. Modify the Triangle code so that it extends Shape, i.e.,

```
public class Triangle extends Shape { ....
```

Triangles are defined by the (x,y) coordinates of the three corner points. The center point of a triangle should be defined as the average of the three respective coordinate values, i.e.,

$$c.x = \left[\frac{x_1 + x_2 + x_3}{3}\right] \quad \text{and} \quad c.y = \left[\frac{y_1 + y_2 + y_3}{3}\right]. \tag{11.6}$$

4. Write a Java program that will initialize and position circle, rectangle and triangle shapes as shown on Figure 11.6, and then compute and print the grid area, x and y coordinates of the grid centroid, moments of inertia I_{xx}, I_{yy}, and I_{xy} computed about the axes/origin and, finally, the grid centroid. For details, see equations 11.1 through 11.4.

Hint. Notice that the layout of shapes in Figure 11.6 is symmetric about the line y = x + 1. Hence, you should expect that: (1) the centroid will lie along this line, and (2) the principal axes will be oriented along this line.

11.3 A footprint model simply defines the area that will be covered by an object. Footprint models of buildings are commonly used in the earliest stages of design and in high-level models of urban areas.

Figure 11.19 shows, for example, the AV Williams building footprint.

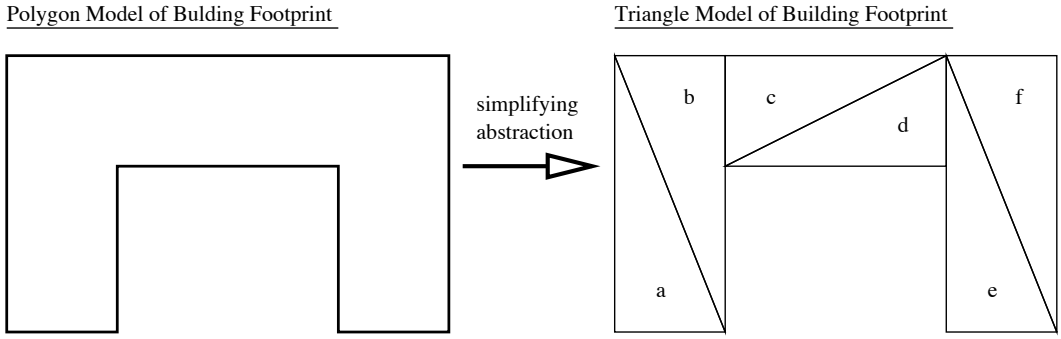

Figure 11.19. Polygon and triangle models for a building footPrint.

Modeling Footprints. Because the footprint area is defined by its perimeter, naturally, a general-purpose polygon model is the first approach that comes to mind. It turns out, however, that polygon operations (e.g., computing the area) can quickly become very complicated. Suppose that a building has an internal courtyard (i.e., the footprint contains a hole). What would you do then?

Many potentially difficult computational problems can be avoided by modeling the footprint as a collection of simple triangular regions, as shown on the right-hand side of Figure 11.19.

Coordinates of A.V. Williams Building Footprint. Let us assume that the AV Williams Building footprint can be modeled with six trianglar alements having geometry as shown in Figure 11.20. And Figure 11.21 shows the relationship among classes that would be used in a software implementation.

Simply put, Figure 11.21 says that one Footprint object will be composed of many Triangle objects. In turn, triangles will be defined in terms of Node and Edge objects. Nodes are an extension of Vector.

Properties of the building footprint (e.g., area, center of mass) will be computed and summed across the ensemble of triangles.

Things to do.

1. Download, compile and execute the Triangle source code from the class website.
2. Write a class called **Footprint** to setup the simplified footprint model for the AV. Williams building, e.g.,

```
Footprint avw = new Footprint();
avw.setName("AV. Williams Buiklding");
```

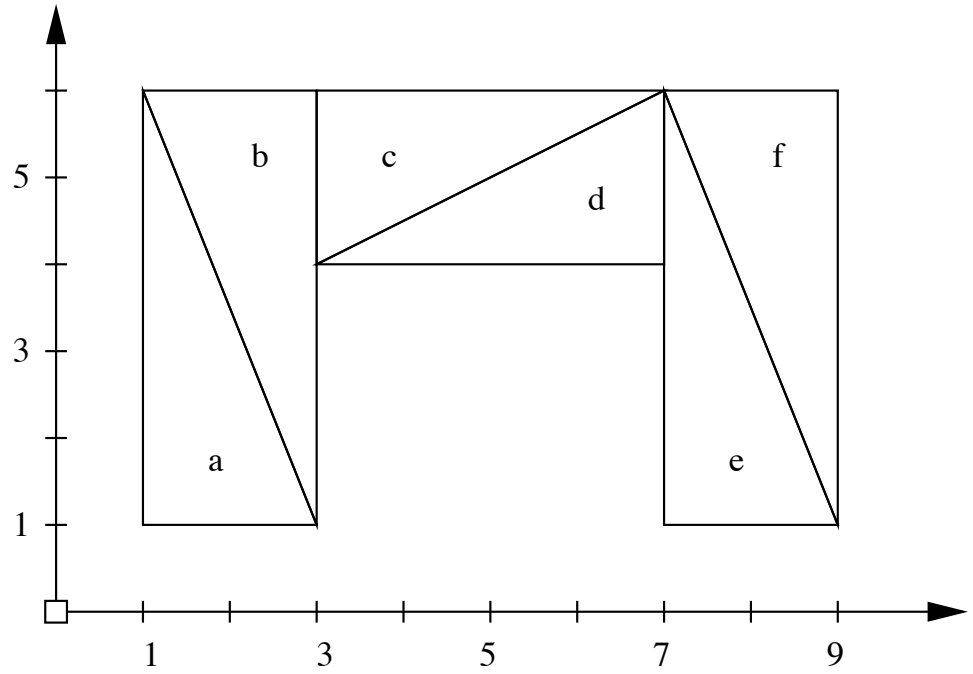

Figure 11.20. Geometric details of the footprint for AV Williams Building

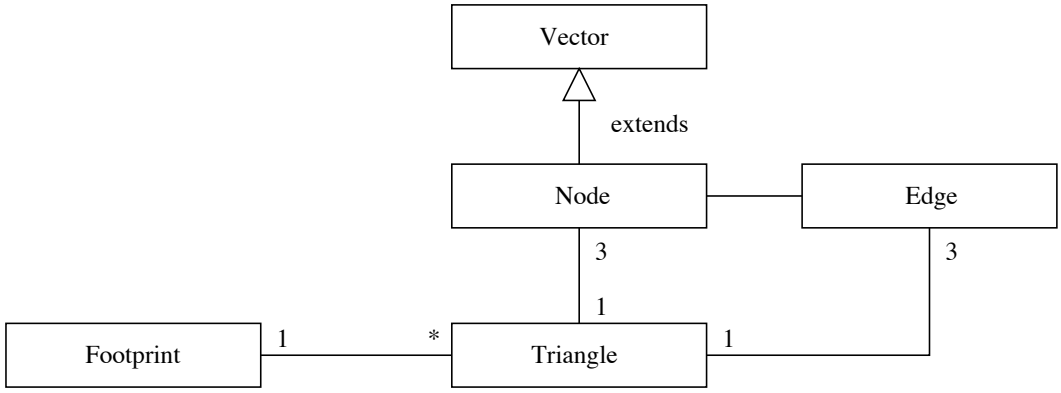

Figure 11.21. Class diagram for building footprint model.

Your program should create the six triangles defining the footprint, and then add them to an arraylist.

3. Write a method `toString()` to create a string representation of the building footprint.

4. Within **Footprint**, write a method called `area()` that will compute the building area by walking along the arraylist and summing the triangle areas.

5. Finally, write methods getCentroidX() and getCentroidY() to compute the x- and y- coordinates of the building centroid.

`Note.` For parts 4 and 5, most of what you need is already defined in Triangle.java.

11.4 A polyline defines a set of one or more connected straight line segments. Polyline abstractions can be found in many areas of Civil Engineering (e.g., road trajectories in transportation, the orange line on the DC Metro, rebar trajectories in structural engineering). As illustrated by these examples, polyline elements typically define open shapes.

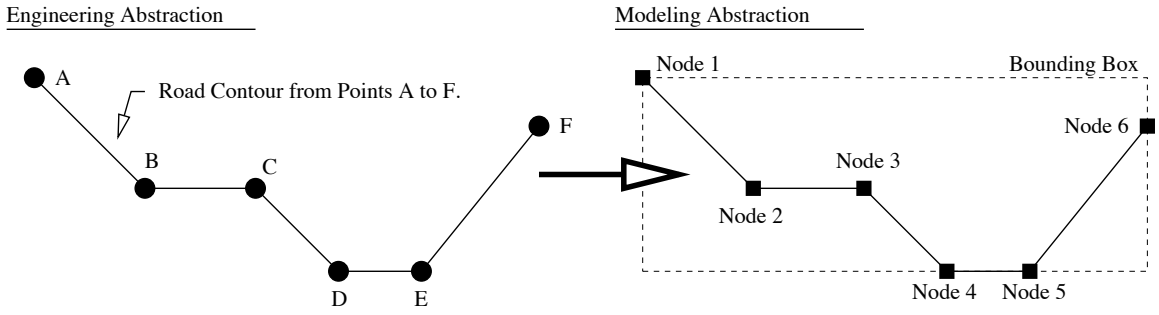

Figure 11.22. Real world and modeling abstractions for polylines

Polyline objects can be created by specifying the endpoints of each segment.

Insert problem description and UML figure soon ...

```
public class Polyline { ...  }

public class LineSegment { ...  }
```

11.5 A path is a continuous line composed of one or more line segments and/or curve segments.

```
public class Path { ... }

public class LineSegment { ... }
public class Curve { ... }
```

Bibliography

[1] Cormen T.H., Leiserson C.E., Rivest R.L. *Introduction to Algorithms*. The MIT Press, 1992.

[2] Liang Y.D. *Introduction to Java Programming (Comprehensive Version) (8th Edition)*. Prentice-Hall, 2011.

Index