Collection Patterns

Dorin Sandu [1]
sandu@acm.org

August 4, 2001

[1]School of Computer Science – Carleton University, 5302 Herzberg Laboratories, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6

Contents

CHAPTER 1

Collections

Collections play an important role in application development by providing reusable data structures to store similar elements and behavior to store, retrieve, and manipulate these elements. However, programmers still have to decide which type of collection to use for the job at hand and, in some situations, which collection implementation. In this chapter we introduce patterns to help selecting the right collection for a given design problem.

1.1 Introduction

Collections are objects which keep track of groups of related objects. The Java Collections Framework defines a series of classes which can be used to store arbitrary objects. For example, an array can be used to store a fixed sequence of objects, a `Vector` can be used to store a variable sequence of objects and a `Hashtable` can be used to store a keyed sequence of objects. These classes provide protocol to:

- add items to the collection.
- remove items from the collection.
- access items in the collection.
- iterate over the items in the collection.
- sort the items in the collection.
- search for an item in the collection.

By providing all this functionality, the collections framework allows programmers to concentrate on the application they are implementing, as opposed to how data is actually stored. Ideally, a well designed object can change how it represents its data

internally just by changing the type of collection used, without affecting the clients of that object. This increases coding speed, quality, and reuse since the collections framework provides well-tested collection implementations.

The Java Collections Framework comprises three entities. First, there are the interfaces, which provide a contract between the collection clients and the collection implementations, then the actual collection implementations and finally, a series of algorithms which help with sorting and searching.

However, while the Java Collections Framework provides a lot of functionality, there still remains the problem of choosing the right collection for the job at hand. Given a particular implementation problem, one must decide which collection to choose in order to satisfy the requirements of that particular job, whether the collections are to increase accessing speed, minimize storage space, or both. In this chapter, we introduce a couple of patterns that attempt to make your decision easier.

1.1.1 Pattern Summary

Table 1.1 summarizes the problems addressed by the patterns in our pattern language and their solutions. We describe *List*, *Fixed List*, *Set*, and *Map* as specializations of the structural pattern *Collection*, followed by the *Constructor* creational pattern, and ending with *Iterator*, *Sorter*, and *Converter* as behavioral patterns.

Pattern	Problem	Solution
Collection	How do you represent a one-to-many relationship?	Use a collection object. Ideally, the collection should be an implementation of either `Collection` or `Map` interfaces, or one of their subclasses.
List	How do you represent an indexed sequence of an unknown number of possibly duplicated elements?	Use a variable-array implementation of the `List` interface.
Fixed List	How do you represent an indexed sequence of a known number of possibly duplicated elements?	Use a fixed-array implementation of the `List` interface.
Set	How do you represent a non-indexed sequence of an unknown number of unique elements?	Use an implementation of the `Set` interface.

Pattern	Problem	Solution
Map	How do you represent a non-indexed sequence of an unknown number of elements that need to be accessed by some arbitrary key?	Use an implementation of the `Map` interface.
Constructor	How do you construct a collection when you know approximately how many elements it will hold?	Construct the collection with the initial capacity the number of elements you think it will store during its lifetime.
Iterator	How do you enumerate the elements of a collection?	Use the iterator object returned by the `iterator()` method.
Sorter	How do you sort the elements of a collection?	Use `Collections.sort()` to sort a `List` view of your collection. Provide the sort criteria in an implementation of the `Comparator` interface.
Converter	How do you convert from one collection type to another?	Use the target collection constructor with the original collection as argument.

Table 1.1: Pattern Problems and Solutions.

The rest of the chapter is structured as follows:

- In Section 1.2 we describe how one-to-many relationships between two objects can be represented by using implementations of the `Collection` interface provided the Java Collection Framework. We describe the *List*, *Fixed List*, *Set*, and *Map* minipatterns in Sections 1.2.1, 1.2.2, 1.2.3, and 1.2.4.

- In Section 1.3 we describe *Constructor* can be used to create and initialize collections when the number of elements can be estimated in advance.

- In Section 1.4 we describe how *Iterator* can be used to traverse collections one element at a time.

- In Section 1.5 we describe how *Sorter* can be used to sort the elements in a collection according to some arbitrary criteria.

- In Section 1.6 we describe how *Converter* can be used to convert between collection types.

1.2 Collection

An instance of a class you are implementing is related to a number of other objects in the system. This one-to-many relationship is important and you decide to capture it in code. The decision is based on the fact that the behavior of your object may also involve the related objects, for example, a drawing object should be able to access all its constituent figures such that, when updating the display, it can tell each figure to update its own region of the display.

How do you represent a one-to-many relationship?

One solution that comes to mind is to have all objects on the *many* side of the relationship reference the object on the *one* side. For the drawing example mentioned above, each figure object has to store the drawing it belongs to in an instance variable. Classes `Drawing` and `Figure` have to be implemented as follows:

```
public class Drawing {
  ...
}

public class Figure {
  private Drawing drawing;
  ...
}
```

This approach doesn't quite work because, while each figure knows to which drawing it belongs to, the drawing has no access to its figures and therefore, cannot invoke their display update method. We can solve this problem by having the object on the *one* side of the relationship, the drawing, reference all objects on the *many* side, the figures. One possible way to achieve this is to change the implementation such that the drawing references the first figure, which references the next figure in the relationship, and so on. The source for `Drawing` and `Figure` is as follows:

```
public class Drawing {
  private Figure figure;
  ...
}

public class Figure {
  private Figure next;
  ...
}
```

So now the drawing can access its figures by traversing a linked list of figures, until the accessor method for the `next` instance variable of some figure returns `nil` indicating that the end of the figures list has been reached. Other possible implementations exist. For example, we can replace the linked list structure with a more efficient list data structure, possibly a doubly-linked list [4], so `Drawing` and `Figure` will look as follows:

```
public class Drawing {
  private Figure figure;
  ...
}

public class Figure {
  private Figure next, previous;
  ...
}
```

Alternatively, instead of a list, we can use an array of figures:

```
public class Drawing {
  private Figure[] figures;
  ...
}

public class Figure {
  ...
}
```

Since arrays are primitive Java types, they can be faster than other data structures. However, this speed comes at a cost in flexibility because arrays can only hold a fixed number of objects which has to be specified at creation time. A better approach is to use specially-provided collections, such as `Vector` and `Hashtable`, provided in the `java.util` package. Both collections store their elements in an internal array which they grow to accomodate more elements when needed. Then `Drawing` and `Figure` can be represented as follows:

```
public class Drawing {
  private Vector figures;
  ...
}

public class Figure {
  ...
}
```

Can we do better than this? By setting the type of `figures` to `Vector` we make `Drawing` dependent on the implementation of the `Vector` class. Should we decide later to change `figures` to a `Hashtable`, we would actually have to modify the source code to change the types. While this seems a minor issue at first, consider that you may have to change the argument and return types of all the methods that directly handle the instance variable `figures`.

Assuming we had control over the implementation of the classes in `java.util` package, one solution would be to actually subclass `Vector` and `Hashtable` from a common abstract class, and make the type of `figures` be the abstract class. Then `figures` can hold either a `Vector` or `Hashtable` instance. Upon close inspection however, we realize that other collections may already be in a hierarchy of their own and, since Java does not support multiple inheritance, it is probably not feasible to force a class relationship between all collections.

Alternatively, we can use the Java interface mechanism. Interfaces are named sets of methods definitions. Regular classes can support different interfaces by providing actual implementations for the methods defined by the respective interfaces. Moreover, interfaces can be used as types, for example, they can be used in variable declarations, or as method argument and return types. Therefore, for our implementation, we can have both `Vector` and `Hashtable` implement some collection interface, and declare `figures` to be of that interface type.

On one hand, we have the need for performance and, on the other hand we have the need to encapsulate data and behavior in order to minimize code and maximize reuse, without having to refactor the existing source code too much. Therefore, in order to represent a one-to-many relationship:

Use a collection object. Ideally, the collection should be an implementation of either `Collection` or `Map` interfaces, or one of their subclasses.

In the one-to-many relationship you are trying to capture, the *one* part is the instance of the class you are implementing, which has an instance variable holding the *many* part or the relationship, in this case, a collection. Consider, for example, the case of the drawing and its figures, which can now be implemented as follows:

```
public class Drawing {
  private Collection figures;
  ...
}

public class Figure {
  ...
}
```

The `figures` instance variable is now of type `Collection`, an interface declared in `java.util` that specifies method definitions to deal with adding, removing, and traversing the collection elements. The advantage of using `Collection` is that the `figures` object implementation can be varied to suit different needs, for example, should the number of figures be small, a memory implementation can be used, should the number of figures increase, a database-collection can be used.

By expressing the one-to-many relationship as an implementation of one of the `Collection` or `Map` interfaces, you achieve the maximum flexibility possible. First, you minimize the code by using the well defined interface protocol and second, you maximize code reuse by using an already implemented collection.

The `Collection` interface is the most basic collection interface introduced by the Java Collections Framework. It provides protocol to add and remove elements without any assumption about whether the collection is indexed, ordered, or whether it allows duplicates:

```
public interface Collection {
  boolean add(Object object);
  boolean addAll(Collection collection);
  boolean remove(Object object);
  boolean removeAll(Collection collection);
```

```
  ...
}
```

In contrast to the `Collection` interface, which makes no assumptions about how the elements are represented, the `Map` interface assumes that the elements are mapped to some other objects:

```
public interface Map {
  Object put(Object key, Object value);
  Object remove(Object key);
  ...
}
```

Java Collection Framework specifies protocol for the basic collection data types shown in Figure 1.1, and provides concrete implementations of these interfaces in hierarchies rooted at `AbstractCollection` and `AbstractMap` classes. Additionally, it provides a series of algorithms that perform useful computations, like sorting and searching, on the provided collection implementations or on any other collection implementing the given interfaces. The implementations provide *reusable data structures* and the algorithms provide *reusable behavior*.

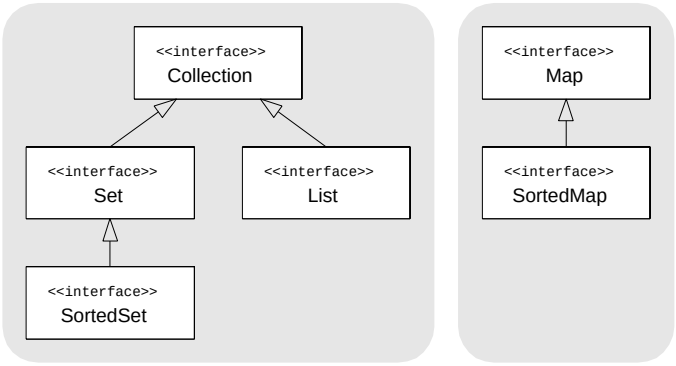

Figure 1.1: Collection framework interfaces.

Given the collection interfaces and their implementations, the question arises, which interface do you choose and, for that interface, which implementation? The minipatterns presented in the next sections attempt to answer the first question, while the second question is answered later in the chapter.

Also known as *Collection*, not to be confused with the Java `Collection` interface, this pattern has been initially developed by Beck [2] for the Smalltalk collections framework, and has later been mapped under the same name to Java by Langr [5].

1.2.1 List

You are in a position to use the *Collection* pattern to represent a one-to-many relationship between instances of the class you are implementing and some other objects in the

system. Instances of your class will reference a variable number of objects and some of these objects participate more than once in the relationship. In the implementation, you will identify the objects by their position in the relationship.

How do you represent an indexed sequence of an unknown number of possibly duplicated elements?

At first glance, you can use plain arrays because they provide indexed access to elements. However, you do not know how many elements there will be so you have to provide code to manage the array size as elements get added or removed. Going back to the drawing example, class `Drawing` can be defined as:

```
public class Drawing extends Object {
  private Figure[] figures;
  private int figuresCount;
  ...
}
```

We implement the drawing as an array of figures, represented by the `figures` instance variable, initialized to some arbitrary-sized array. The other instance variable, `figuresCount` stores how many figures the drawing currently has. It is initialized to zero since the drawing contains zero figures initially. We implement the methods `get()` and `set()` to provide indexed access to the figures. The methods do nothing but check for a valid index and access the figures array at that index:

```
public Figure get(int index) {
  if(index < 0 || index > figures.length)
  throw new java.lang.RuntimeException("index out of bounds");
  return figures[index];
}

public Figure set(int index, Figure figure) {
  if(index < 0 || index > figures.length)
  throw new java.lang.RuntimeException("index out of bounds");
  Figure previous = figures[index];
  figures[index] = figure;
  return previous;
}
```

In addition to indexed access, we must provide methods that manage the figures array size. The `add()` method takes care of adding a new figure to the drawing and, when the array fills, it allocates a larger array and copies the existent figures over:

```
public boolean add(Figure figure) {
  if(figures.length == figuresCount) {
    Figure[] newFigures = new Figure[figures.length * 3 / 2];
    for(int i=0; i<figures.length; i++) {
      newFigures[i] = figures[i];
    }
    figures = newFigures;
  }
  figures[figuresCount] = figure;
  figuresCount = figuresCount + 1;
  return true;
}
```

In a similar way, the `remove()` method removes the given figure from the drawing and takes care to left-shift by one all the figures to the right of the index of the removed figure:

```
public boolean remove(Figure figure) {
  int i = 0;
  for(; i<figuresCount; i++) {
    Figure current = (Figure)figures[i];
    if(current.equals(figure)) break;
  }
  if(figuresCount == i) return false;
  for(; i<figuresCount-1; i++) {
    figures[i] = figures[i+1];
  }
  figures[figuresCount-1] = null;
  figuresCount = figuresCount - 1;
  return true;
}
```

This solution has two major drawbacks. First, the methods that add and remove figures implement additional behavior to grow the figures array as needed. This behavior should be implemented separately, either in private methods in a superclass or in a totally different class that can be reused in other parts of the system. Second, the use of arrays imposes a strong coupling between `Drawing` and the `figures` array. For example, because arrays do not implement any of the `Collection` or `Map` interfaces, should you decide later to represent figures as a linked list, you have to change both `add()` and `remove()`, and for that matter all methods referencing `figures`, to work with linked list protocol.

So you need to balance the need to implement a collection that allows indexed access to an unknown number of possibly duplicated elements with the need to reuse some default implementation. Even if you decide to implement your own collection, in order to maximize reuse, you should provide an implementation of the `List` interface.

Use a variable-array implementation of the `List` interface.

Because, as shown in Figure 1.1, `List` is a subclass of `Collection`, implementors must provide all `Collection` protocol plus `List`-specific methods. These methods provide indexed element access:

- `Object get(int index)`. This method returns the element at the specified index in the receiver.
- `Object set(int index, Object element)`. This method sets the element at the specified index in the receiver to the specified element and returns the element previously stored at that index or `null` if there was no element present.
- `void add(int index, Object element)`. This method sets the element at the specified index in the receiver to the specified element. Additionally, it shifts existing elements to the right of the specified index.

- `Object remove(int index)`. This method removes the element a the specified position and returns it. Additionally, it shifts existing elements to the left of the specified index.

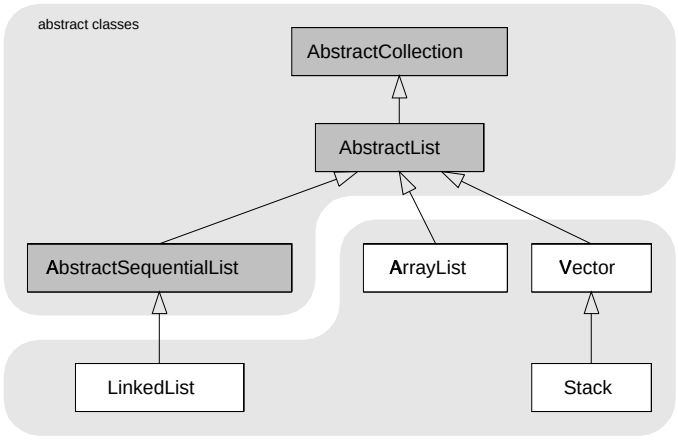

Figure 1.2: `List` interface implementors.

Implementors of the `List` interface grow to accomodate as many elements as possible. Java Collections Framework provides several implementations organized in the hierarchy shown in Figure 1.2. They are:

AbstractList provides a skeleton implementation of the `List` interface which uses an underlying random-access data structure to store its elements. This basic implementation is intended to be subclassed with specialized implementations for the rest of the `List` protocol.

AbstractSequentialList doesn't implement the `List` interface but it supports its protocol because it subclasses `AbstractList`. It provides a basic implementation intended for sequential-access which can be extended in subclasses.

LinkedList implements all optional `List` operations using a backing linked-list data structure. In addition to the `List` protocol, it provides standard linked-list methods to access, add, and remove the head and tail of the list.

ArrayList implements all optional `List` operations using a backing array data structure whose size is increased dynamically, as more elements are added.

Vector is conceptually equivalent with `ArrayList` and is only provided for backward compatibility to the collections provided before Java 2.

Stack doesn't actually provide an implementation for the `List` interface but, because it subclasses from `Vector` it supports the `List` protocol. Additionally, it provides the traditional stack methods for pushing, popping, peeking, and checking whether the stack is empty.

By using one of these collection to store the figures inside a drawing object, we can rewrite the above example as follows:

```
public class Drawing extends Object {
  private List figures;

  public Drawing() {
    figures = new ArrayList();
  }

  public Drawing(int maxFigures) {
    figures = new ArrayList(maxFigures);
  }

  public Figure get(int index) {
    return (Figure)figures.get(index);
  }

  public Figure set(int index, Figure figure) {
    return (Figure)figures.set(index, figure);
  }

  public boolean add(Figure figure) {
    return figures.add(figure);
  }

  public boolean remove(Figure figure) {
    return figures.remove(figure);
  }
}
```

Written this way, the drawing implementation looks much simpler. This is because the methods simply delegate their behavior. Both `get()` and `set()` methods delegate to the corresponding methods provided by `List`. The same goes for the `add()` and `remove()` methods. Additionally, since now the `figures` is of type `List` we can use any implementation of the `List` interface, in this case an `ArrayList`, and keep drawing methods unchanged.

This pattern is also known as *OrderedCollection* [2] and as *List* or *LinkedList* [5].

1.2.2 Fixed List

You are in a position to use the *Collection* pattern to represent a one-to-many relationship between instances of the class you are implementing and some other objects in the system. However, you realize that instances of your class will always reference at most a certain number of objects and some of these objects participate more than once in the relationship. In the implementation, you will identify the objects by their position in the relationship.

How do you represent an indexed sequence of a known number of possibly duplicated elements?

Right away, you realize you can use arrays because they provide indexed access to elements and have fixed size. However, on second thought, arrays are one of the Java

built-in primitives and do not adhere to the collection interfaces promoted by the Java Collections Framework. Of course you can use the *Converter* pattern to get a collection type you need but that should be the last resort. Objects similar to the array data type exist:

- First, class `java.util.Arrays` contains methods to manipulate arrays. There are methods to convert the array to a `List`, sort, search, fill, and test for equality.
- Second, class `java.sql.Array` is the mapping of the SQL type ARRAY to Java.
- Third, class `java.lang.reflect.Array` which provides static methods to dynamically create and access Java arrays. This behavior is used to access the fields and methods of Java objects and classes.

You can convert the array to a `List` instance using the first approach, or even use one of the `List` implementations directly, but the resulting list does not have a fixed size. You probably should not used the other two approaches either, as `java.sql.Array` and `java.lang.reflect.Array` are meant to be used in the context of SQL queries and object reflection and may be subject to protocol changes in the future. The solution seems to rely on a user implementation, you either have to implement a new collection or subclass `ArrayList` or `LinkedList` and override the appropriate methods. The following implements a new collection, `FixedList`, as a subclass of `AbstractList`:

```
public class FixedList extends AbstractList {
  private Object[] elements;
  private int elementCount;

  public FixedList() {
    elements = new Object[10];
    elementCount = 0;
  }

  public FixedList(int maxElements) {
    elements = new Object[maxElements];
    elementCount = 0;
  }

  public int size() {
    return elements.length;
  }

  public Object[] toArray() {
    return (Object[])elements.clone();
  }

  public Object get(int index) {
    if(index < 0 || index > elements.length)
    throw new java.lang.RuntimeException("index out of bounds");
    return elements[index];
  }
```

```
  public Object set(int index, Object element) {
    if(index < 0 || index > elements.length)
    throw new java.lang.RuntimeException("index out of bounds");
    Object previous = elements[index];
    elements[index] = element;
    return previous;
  }

  public boolean add(Object element) {
    if(elements.length == elementCount)
      return false;
    elements[elementCount] = element;
    elementCount = elementCount + 1;
    return true;
  }

  public boolean remove(Object element) {
    int i = 0;
    for(; i<elementCount; i++) {
      Object current = elements[i];
      if(current.equals(element)) break;
    }
    if(elementCount == i) return false;
    for(; i<elementCount-1; i++) {
      elements[i] = elements[i+1];
    }
    elements[elementCount-1] = null;
    elementCount = elementCount - 1;
    return true;
  }
}
```

The bulk of the code is identical to the first approach we took for the *List* pattern. Internally, the fixed list stores its elements in an array and uses a count to keep track of how many elements are added. The `get()` and `set()` methods provide indexed access to the elements array and methods `add()` and `remove()` add and remove elements to and from the elements array. In contrast to the approach we took for the *List* pattern, the `add()` method does not grow the elements array when it becomes full.

The other solution would be to subclass `FixedList` from either `ArrayList` or `LinkedList` and only override the `add()` method to behave as above. By subclassing from `ArrayList` you become dependent on its implementation which may change. The same applies to `LinkedList` along with the fact that linked lists use `Entry` objects to wrap each element. Therefore:

Use a fixed-array implementation of the `List` interface.

You can provide your own implementation of a fixed-array `List` as above or you can use the one that Java provides. The implementation provided by Java can be used as follows:

```
// ...inside class Arrays
public static List asList(Object[] a) {
```

```
return new ArrayList(a);
}
```

So, to create a fixed-array List, one has to invoke `Arrays.asList(new Object[size])` where `size` is the size of the primitive array object used by the implementation. This method actually returns an `ArrayList` object, where `ArrayList` is an inner-class private to the `Arrays` class, not to be confused with the `AbstractList` subclass by the same name which uses a variable-array implementation.

```
// Arrays
private static class ArrayList
extends AbstractList
implements java.io.Serializable {
  private Object[] a;

  ArrayList(Object[] array) {
    a = array;
  }

  public int size() {
    return a.length;
  }

  public Object[] toArray() {
    return (Object[]) a.clone();
  }

  public Object get(int index) {
    return a[index];
  }

  public Object set(int index, Object element) {
    Object oldValue = a[index];
    a[index] = element;
    return oldValue;
  }
}
```

Besides the unfortunate name collision, you have to decide which implementation to use, as the default does not provide `add()` and `remove()` which results in `UnsupportedOperationException` being triggered every time these two are invoked.

This pattern is also known as *Array* [5].

1.2.3 Set

You are in a position to use the *Collection* pattern to represent a one-to-many relationship between instances of the class you are implementing and some other objects in the system. Instances of your class will reference a variable number of objects which do not participate more than once in the relationship. Also, the order of the objects in the relationship is not important, for example, you will not reference any of the objects by index.

How do you represent a non-indexed sequence of an unknown number of unique elements?

As in the *List* pattern, the first data structure that comes to mind is arrays. However, arrays do not really match the requirements because of two problems. First, they are indexed data structures and second, they do allow for duplicate elements. You can still use them but you have to provide behavior to remove duplicate elements, grow the array as needed, and use the *Converter* pattern to provide a Java Collections Framework interface to clients.

`Stack` is too specialized for the given requirements so you may choose `Vector` for synchronized access to its elements or between `LinkedList` and `ArrayList` based on the relative benefits of using a backing list or array implementation and non-synchronized access to stored elements. In all cases however, you have to deal with indexed access and have to write additional behavior to remove duplicate elements. The only advantage over arrays is that all these classes provide the adequate interfaces.

To solve these problems, the Java Collections Framework provides the required access protocol in the `Set` interface. Therefore:

Use an implementation of the `Set` interface.

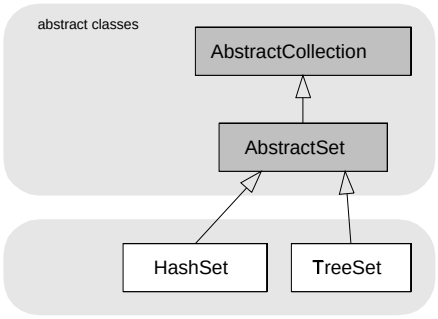

Figure 1.3: `Set` interface implementors.

The `Set` interface, as shown in Figure 1.1 is a subclass of `Collection`. In addition to `Collection` protocol, `Set` places additional constraints on the constructors to create set objects without duplicate elements, and adds the following methods:

- `boolean add(Object element)`. This method adds the given element to the receiver it is not already present.
- `boolean addAll(Collection elements)`. This method adds all elements in the given collection to the receiver if they are not already present.
- `boolean equals(Object set)`. This method returns a boolean indicating whether the receiver and the given set are equal. In contrast to the other interfaces, implementors of `Set` must first check whether the receiver and given

set have the same size and then whether each element in the receiver is present in the given set (or equivalently, every element in the given set should be in the receiver).

This requirement for `Set` implementors makes impossible for objects to implement both `List` and `Set` interfaces.

- `int hashCode()` This method computes and returns the hash value of the receiver. The hash value must be computed as the sum of all the elements hash values, where the hash value of `null` is defined to be zero.

  This requirement for `Set` implementors makes impossible for objects to implement both `List` and `Set` interfaces.

Java Collections Framework provides several implementations for the `Set` interface, as shown in Figure 1.3. They are:

AbstractSet provides a skeleton implementations of the `Set` interface. This basic implementation is designed to be subclassed by more specialized implementations.

HashSet implements the `Set` interface using an underlying hash-table implementation. This implementation does not provide any guarantees for the order of the elements in the set.

TreeSet doesn't implement the `Set` interface directly but it provides the `Set` protocol by subclassing from class `AbstractSet`. This implementation provides guarantees on how the elements are sorted, ascending order by default or the order specified by a `Comparator` implementation provided in the constructor.

1.2.4 Map

You are in a position to use the *Collection* pattern to represent a one-to-many relationship between instances of the class you are implementing and some other objects in the system. Instances of your class will reference a variable number of objects which may participate more than once in the relationship. Additionally, you need to access the elements in the collection by some object. Also, the order of the objects in the relationship is not important, for example, you will not reference any of the objects by index.

How do you represent a non-indexed sequence of an unknown number of elements that need to be accessed by some arbitrary key?

At first glance you can use two arrays, one for the keys and the other one for the values. Given the key object, you can find the element it is mapped to by first finding the index of the key in the keys array and returning the object at that index from the values array. This approach poses the same problems outlined in the previous patterns. First, we have to provide behavior to remove duplicate keys and their corresponding values. Second, as arrays are fixed size, we need to provide behavior to resize them as more elements get added. Third, arrays do not provide collection interfaces.

Since arrays do not provide standard collection interfaces, we can try the above approach with `List` implementations. However, we still have to provide additional behavior to remove duplicate keys and their values.

The only data structures that do not provide any order on their elements, and grow as more elements are added, are implementors of the `Set` interface. At first glance we can solve all above problems using sets. We cannot use two sets, one for keys and one for values, because sets are not ordered and cannot identify their elements by their index. The only approach we can use to provide a pair object that associates the key and value and computes the hash value based on the key and value hash values. However, this looks like extra work, we still have to provide extra behavior to wrap keys and their values into pairs and to add the pairs to the set.

To solve these problems, the Java Collections Framework provides the required access protocol in the `Map` interface. Therefore:

Use an implementation of the `Map` interface.

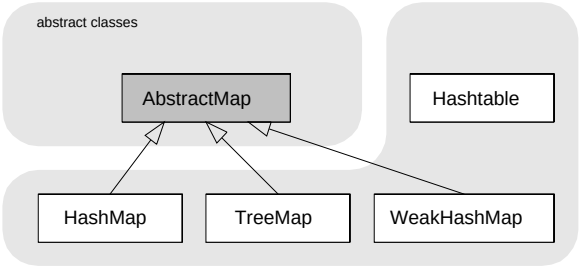

Figure 1.4: `Map` interface implementors.

Because, as shown in Figure 1.1, `Map` is a subclass of `Collection`, implementors must provide all `Collection`

- `Object get(Object key)`. This method returns the value for the specified key.
- `Object put(Object key, Object value)`. This method associates the specified value with the specified key. It returns the value that was previously associated with the key or `null` if there was none.
- `Object remove(Object key)`. This method removes the value associated with the specified key. It returns the value that was previously associated with the key or `null` if there was none.

Additionally, `Map` provides protocol for views on the underlying implementation:

- `Set entrySet()`. This method returns a set view of the mappings in the receiver. The elements in this view are instances of `Map.Entry`.

- `Set keySet()`. This method returns a set view of the keys in the receiver.
- `Collection values()`. This method returns a collection view of the values contained in the receiver.

Java Collection Framework provides several implementations for the `Map`, organized in the hierarchy shown in Figure fig:mapimplementors. They are:

AbstractMap provides a skeleton implementation of the `Map` interface. This basic implementation is designed to be subclassed by more specialized implementations.

HashMap implements the `Map` interface using an underlying hash-table implementation. This implementation does not provide any guarantees for the order of the elements in the set.

TreeMap doesn't implement the `Map` interface directly but it provides the `Map` protocol by subclassing from class `AbstractMap`. This implementation provides guarantees on how the elements are sorted, ascending order by default or the order specified by a `Comparator` implementation provided in the constructor.

WeakHashMap is conceptually equivalent with `HashMap` and it provides *weak* keys. Weak keys imply that the value will be removed if no other object but the map itself references the key.

Hashtable is conceptually equivalent with `HashMap` and is only provided for backward compatibility to the collections provided before Java 2.

1.3 Constructor

You are in a position to use the *Collection* pattern to represent a one-to-many relationship between instances of the class you are implementing and some other objects in the system. Instances of your class will reference a variable number of objects and you know the approximate number of objects that will be stored in the collection. This number does not have to be exact, more or less objects can be added. You decide to use an array-based implementation, such as `ArrayList`, `HashSet`, or `HashMap`.

How do you construct a collection when you know approximately how many elements it will hold?

Consider the example of filling an `ArrayList` with some objects. The following code does just that by adding one million `dummy` objects to a list. The operation creates the `ArrayList` instance with the default constructor and finishes adding the objects in 3164 milliseconds:

```
public class ListTest {
  public static void main(String[] args) {
    List list = new ArrayList();
    long startTime, stopTime;
```

```
    startTime = System.currentTimeMillis();
    for(int i=0; i<1000000; i++) {
      Object dummy = new ListTest();
      list.add(dummy);
    }
    stopTime = System.currentTimeMillis();

    System.out.println(stopTime - startTime);
  }
}
```

This seems rather expensive. Can we do better? We should first look at the implementation for the method `add`:

```
public boolean add(Object o) {
  ensureCapacity(size + 1);
  elementData[size++] = o;
  return true;
}
```

Internally, `ArrayList` uses an array instance variable, `elementData`, to store its elements. It uses another instance variable, `size` to keep track of the number of elements in the array. A new element is always inserted at index `size + 1`. To make sure the array has enough space to store the element to be added, `ensureCapacity()` is invoked before the new element is added:

```
public void ensureCapacity(int minCapacity) {
  modCount++;
  int oldCapacity = elementData.length;
  if (minCapacity > oldCapacity) {
    Object oldData[] = elementData;
    int newCapacity = (oldCapacity * 3)/2 + 1;
    if(newCapacity < minCapacity)
      newCapacity = minCapacity;
    elementData = new Object[newCapacity];
    System.arraycopy(oldData, 0, elementData, 0, size);
  }
}
```

This method checks to see if the current array length `oldCapacity` is smaller than the minimum capacity required, `minCapacity`. Should this be the case, the method allocates a new array of larger size and copies the old elements into it. The newly allocated array is one and half times larger than the previous array so, if we go back to our example, adding one million objects to an `ArrayList` caused its backing array to be grown and copied many times. The default constructor for `ArrayList` initializes the backing array to size 10 so, the 11th element causes the array to grow to size 16 and later to 25, 38, 58, 88, 133, etc. To avoid growing the array and copying its elements over so many times, it would help tremendously if we could preallocate the array to size one million from the beginning. Therefore:

Construct the collection with the initial capacity the number of elements you think it will store during its lifetime.

Java documentation for the collections framework recommends all collection interface implementations have two constructors. The first is the void (no arguments) constructor which creates an empty collection and the second, a constructor with another collection as its argument, which creates a collection with the same elements as its argument. Besides these constructors, array-based collections implementations provide a third collection that takes as argument an integer representing the initial capacity of the collection. Using this constructor for the example above, the code becomes:

```
public class ListTest {
  public static void main(String[] args {
    List list = new ArrayList(1000000);
    long startTime, stopTime;

    startTime = System.currentTimeMillis();
    for(int i=0; i<1000000; i++) {
      Object dummy = new ListTest();
      list.add(dummy);
    }
    stopTime = System.currentTimeMillis();
    System.out.println(stopTime - startTime);
  }
}
```

Preallocating the array almost triples the speed with which the `ArrayList` is filled. Similar speed improvements can be obtained by specifying a large initial capacity with `HashSet` and `HashMap`. We ran the above example to obtain the results shown in Table 1.2.

	default capacity	initial capacity
ArrayList	3215	1161
HashSet	19278	9594
HashMap	19157	9524

Table 1.2: Construction time in milliseconds.

In conclusion, when you are using an array-based dynamic collection and you have to add a large number of elements, you may want to create the collection with a large enough initial capacity in order to prohibit the collection to grow its backing array too many times.

This pattern has been developed initially by Auer and Beck as *Hypoth-a-Sized Collection*, one of their patterns for efficient Smalltalk programming [1]

1.4 Iterator

You are in a position to use the *Collection* pattern to represent a one-to-many relationship between instances of the class you are implementing and some other objects in the system. Clients of your object have to process all the objects in the collection and you need to provide them with access to the referenced objects. Processing order may or may not be important.

How do you enumerate the elements of a collection?

To enumerate through the elements stored into an array, you can use either a `for` or `while` construct. Consider the task of enumerating the elements in an array using a `for`-statement:

```
for(int i=0; i<array.length; i++) {
  Object element = array[i];
  // perform element operation
}
```

and a `while`-statement:

```
int i = 0;
while(i<array.length) {
  Object element = array[i++];
  // perform element operation
}
```

This approach will also work with primitive arrays and `List` implementations. For example, the following code creates an `ArrayList` holding one million dummy objects and enumerates through them in 130 milliseconds:

```
public class ListTest {
  public static void main(String[] args) {
    List list = new ArrayList(1000000);
    for(int i=0; i<1000000; i++) {
      Object dummy = new ListTest();
      list.add(dummy);
    }

    long startTime = System.currentTimeMillis();
    for(int i=0; i<list.size(); i++) {
      Object element = list.get(i);
      // perform element operation
    }
    long stopTime = System.currentTimeMillis();
    System.out.println(stopTime - startTime);
  }
}
```

The same code above will execute in 30 milliseconds for enumerating through objects stored in a primitive array of the same size. The difference in speed is because all operations on an `ArrayList` go through one level of indirection to operate on its backing array, for example, clients of an `ArrayList` call `get(int index)` which in turn accesses its backing array via the `[]` operator.

The `for` and `while` constructs can only be used on indexed data structures like primitive arrays and `List` implementations such as `ArrayList` and `LinkedList`. You probably have to convert `Set` and `Map` objects to arrays before you can enumerate their elements. This however poses another problem: the default implementation of `toArray()` in `AbstractCollection` creates a new array and copies the elements into it:

```
public Object[] toArray() {
  Object[] result = new Object[size()];
  Iterator e = iterator();
  for (int i=0; e.hasNext(); i++)
    result[i] = e.next();
  return result;
}
```

This method is inherited by most collection implementations. It is overriden by `ArrayList` to perform a slightly better array copy, and by `LinkedList` to deal with the links between entries.

Rather than having to deal with all this special cases, it would be nice to be able to enumerate the elements in a collection polymorphically. To do this for all types of collections, we have to move the traversal logic inside the collection itself. Assuming the client controls the traversal, we can write the following:

```
public class ArrayList implements List {
  private int i = 0;

  public boolean hasNext() {
    return i != size();
  }

  public Object next() {
    return get(i++);
  }
}
```

This strategy however, forces us to extend collections with methods dealing with different kinds of traversals. A much better way is to encapsulate the behavior of each traversal into an *Iterator* [3] and have the collections return such objects via *Factory Methods* [3]. Therefore:

Enumerate the elements of a collection using the iterator object returned by the `iterator()` method.

The Java Collections Framework implements iterators via `Enumeration` and `Iterator` interfaces. Iterators offer the same functionality with more appropriate method names and, additionally, permit clients to remove elements from the underlying collection during the iteration. The two interfaces are so similar that, in fact, `Collections.enumeration()` returns an enumeration wrapper for an iterator:

```
public static Enumeration enumeration(final Collection c) {
  return new Enumeration() {
    Iterator i = c.iterator();

    public boolean hasMoreElements() {
      return i.hasNext();
    }

    public Object nextElement() {
      return i.next();
    }
  };
}
```

We can rewrite the example mentioned above to use an iterator as follows:

```
public class ListTest {
  public static void main(String[] args) {
    List list = new ArrayList(1000000);
    for(int i=0; i<1000000; i++) {
      Object dummy = new ListTest();
      list.add(dummy);
    }

    Iterator iterator = list.iterator();
    long startTime = System.currentTimeMillis();
    while(enum.hasNext()) {
      Object element = iterator.next();
      // perform element operation
    }
    long stopTime = System.currentTimeMillis();
    System.out.println(stopTime - startTime);
  }
}
```

In addition to the regular iterator interface, `List` implementors must provide an additional iterator, called a `ListIterator`. This iterator is clients to insert or replace elements as they traverse a list object. Additionally, clients can traverse list objects in reverse, whithout having to reverse the list themselves. Also, list iterators can start iterating at any given index in a list object.

We've seen iterators work with `Collection` implementations. What about maps? It turns out we have to traverse maps a bit differently. We can traverse a map as a set of keys, collection of values, or set of key-value mappings. 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.

1.5 Sorter

You have used the *Collection* pattern to represent a one-to-many relationship. Clients of your object have to process the objects in the collection in a certain order and you need to either sort the collection directly or provide your clients with a sorted view of the objects in the collection.

How do you sort the elements of a collection?

You can provide your own sorting by implementing the appropriate methods in your class. For example, assuming that you want to sort the figures in a drawing by in increasing order of their name, you can provide a `sort()` in the drawing class to do just that:

```
public class Drawing {
  private List figures = new ArrayList();
  ...
```

```
  private void sort() {
    // sort figures here
  }
  ...
  public Iterator iterator() {
    sort();
    return figures.iterator();
  }
}
```

The `sort()` method sorts the `figures` collection in place just before an iterator is returned on it. The disadvantage of this approach is that you have to provide a sorting method in every class that needs to sort its internal collections.

Alternatively, you can provide a special implementation of the `Iterator` interface. This new implementation sorts a copy of the figures collection and then iterates over it. The code becomes:

```
public class FigureIterator implements Iterator {
  private Iterator iterator;
  public FigureIterator(List figures) {
    iterator = sort(figures).iterator();
  }
  private List sort(List figures) {
    // sort figures here
  }
  public boolean hasNext() {
    return iterator.hasNext();
  }
  public Object next() {
    return iterator.next();
  }
  public void remove() {
    iterator.remove() {
  }
}

public class Drawing {
  private List figures = new ArrayList();
  ...
  public Iterator iterator() {
    return new FigureIterator(figures);
  }
}
```

This doesn't seem appropriate either, as `FigureIterator` pretty much hardwires how the figures should be sorted in the `sort()` method. Since the ordering of the objects is dictated by how objects compare to each other, we can generalize `sort()` to simply compare two figures, and implement the `Comparable` interface on the figure object:

```
public class SortingIterator implements Iterator {
  ...
  private List sort(List objects) {
    List sortedList = new ArrayList();
    List unsortedList = objects.clone();
```

```
    while(!unsortedList.isEmpty()) {
      Object smallest = get(0);
      Iterator iterator = unsortedList.iterator();
      while(iterator.hasNext()) {
        Object current = iterator.next();
        int status = smallest.compareTo(current);
        if(status > 0) smallest = current;
      }
      unsortedList.remove(smallest);
      sortedList.add(smallest);
    }
    return sortedList;
  }
  ...
}

public class Figure implements Comparable {
  ...
  public int compareTo(Object object) {
    Figure figure = (Figure)object;
    return getName().compareTo(figure.getName());
  }
}
```

The `sort()` method above simply moves the smallest element from one collection to the end of another collection until the first collection becomes empty. It is not the most efficient algorithm but it serves the purpose to show that the smallest element picked is determined by the outcome of the `compareTo()` method implemented on the objects in the collection. With this approach, we have generalized the iterator, and we only have to implement the appropriate comparison methods on the objects that we plan to sort. Is this better? Not really, since objects can compare themselves with other objects in different ways.

In review, we have tried to place the sort behavior into the drawing itself, into the iterator, and ultimately on the objects stored in the collection. Neither approaches work well because we usually need to sort objects differently in different cases. To be able to switch sorting algorithms at runtime, it looks like we need to place the comparison behavior into separate objects that can be plugged in to some sorting algorithm. Therefore:

Use `Collections.sort()` to sort a `List` view of your collection. Provide the sort criteria in an implementation of the `Comparator` interface.

The interface `Comparator` is defined as follows:

```
package java.util;
public interface Comparator {
  int compare(Object anObject, Object anotherObject);
  boolean equals(Object anObject);
}
```

Method `compare(Object anObject, Object anotherObject)` compares `anObject` with `anotherObject`. This method is used by an implementation

of the `Comparator` interface to compare the elements of a collection, two at the time, to determine which comes first. It returns a negative integer if the receiver object is less than `anObject`, zero if the objects are equal, or a positive integer if the receiver object is greater than `anObject`. An implementation for the second method, `equals(Object anObject)` should normally not be provided, as it is inherited from `Object`.

Then, the sorting can be done in place, using an inner class as follows:

```
Collections.sort(figures, new Comparator() {
  public int compare(Object anObject, Object anotherObject) {
    Figure figure1 = (Figure)anObject;
    Figure figure2 = (Figure)anotherObject;
    return figure1.getName().compareTo(figure2.getName());
  }
})
```

As we can see, the sorting criteria can be plugged into the algorithm that sorts a given collection. The result is that we free all objects involved from providing the pairwise comparison behavior, and we force the object that actually needs the figures to be sorted to do the sorting. In this case, such an object can be a list box providing an alphabetical list of the figures in the drawing.

1.6 Converter

You are in a position to use the *Collection* pattern to represent a one-to-many relationship between instances of the class you are implementing and some other objects in the system. Your object has to interact with other parts of the system that may expect a different type of collection so you have to convert your collection to match the required type. More precisely, you may find yourself in one of these situations:

- You have to convert your collection to or from an array. This may happen when you have to invoke Java methods that return array primitive objects or take array primitives as arguments.
- You have to convert your collection to or from another collection type. This may happen because you have to deal with older collection types, such as `Vector` or `Hashtable` which, although implement the appropriate List and Map interfaces, are really slow due to the synchronized access. Alternatively, you may have to convert between two collection types.

How do you convert from one collection type to another?

First of all, in our opinion, you should not give your clients access to your collection directly. This is because your main goal is to promote encapsulation, in other words, you don't want to allow clients to change your object's internal state directly. If you create public accessor methods to return the collection instance variable, clients can

modify it at their discretion, not a really good thing if your object needs to keep track of the objects that get added to or removed from the collection.

To avoid exposing the internal state of your object, it is better to apply the *Iterator* pattern and provide sequential access to the objects in the collection. Additionally, your object should provide behavior to add and remove elements from the internal collection. This way, the collection does not get modified without your object knowing about it, providing a very effective firewall against change from outside.

Of course, there are those occasions beyond your control, such as a library written by somebody else that you don't have the code to, nor the time or desire to modify and you must make use of the given interface. Some of these interfaces require you to pass collections from object to object and, even more, convert between collection types. The first thing that comes to mind is to implement an Adapter or Bridge between the two types but this seems like overkill.

One thing for sure, in order to preserve encapsulation, you should return a shallow copy of the collection, and this shallow copy should be of the type expected by the conversion. This can be accomplished easily by iterating through the original collection and adding every element to a new instance of the required collection type. Consider the following example where we convert from a List type to a Set type:

```
public class ConverterTest {
  public static void main(String[] args) {
    List list = new ArrayList(1000000);
    for(int i=0; i<1000000; i++) {
      Object dummy = new ConverterTest();
      list.add(dummy);
    }

    Set set = new HashSet(2000000);
    Iterator iterator = list.iterator();
    long startTime = System.currentTimeMillis();
    while(iterator.hasNext()) {
      Object element = iterator.next();
      set.add(element);
    }
    long stopTime = System.currentTimeMillis();
    System.out.println(stopTime - startTime);
  }
}
```

We have to duplicate this code everywhere we need to convert between lists and sets. On one hand we definitely like to convert the list to a set, or vice-versa, on the other hand we would like to minimize the code we have to write. So, rather than use the copy-and-paste approach, we can encapsulate this behavior into a separate object. Java already provides this behavior in the form of *Parametrized Constructor* methods in the collection classes. Therefore:

Convert a collection to another collection type by using the target collection constructor with the original collection as argument.

The parametrized constructors take a `Collection` implementation as argument and construct a new collection of the type of the class the constructor belongs to. The above code can be written as:

```
public class ConverterTest {
  public static void main(String[] args) {
    List list = new ArrayList(1000000);
    for(int i=0; i<1000000; i++) {
      Object dummy = new ConverterTest();
      list.add(dummy);
    }

    long startTime = System.currentTimeMillis();
    Set set = new HashSet(list);
    long stopTime = System.currentTimeMillis();
    System.out.println(stopTime - startTime);
  }
}
```

Both approaches take around 14571 milliseconds to perform. However, by using the constructor approach, we make the code more readable as it is obvious we are building a set from a list. Another advantage of this approach is that conversions can be done on the fly, without introducing local variables:

```
doSomething(new HashSet(list));
```

A similar pattern, *Duplicate Removing Set* is used by Kent Beck [2] to convert between collections to sets for the sole purpose to remove duplicate objects from the original collection.

1.7 Patterns in Action

The following classes implement a simple contact manager. To compile and run the examples, execute:

```
javac Contact.java ContactManager.java
java ContactManager
```

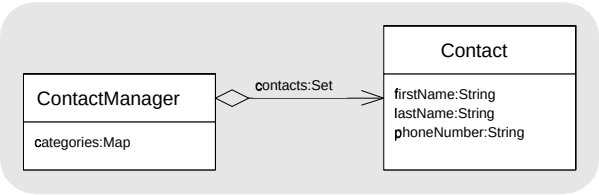

Figure 1.5: Contact manager application.

1.7.1 Class Contact

Class `Contact` models a contact. Contact objects keep track of the first and last names of the contact people, and their phone numbers:

```
package collections.example;

import java.lang.*;

public class Contact implements Comparable {
  private String firstName;
  private String lastName;
  private String phoneNumber;

  private Contact() {
    // do nothing
  }

  public Contact(String firstName, String lastName, String phoneNumber) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.phoneNumber = phoneNumber;
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public String getPhoneNumber() {
    return phoneNumber;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public void setPhoneNumber(String phoneNumber) {
    this.phoneNumber = phoneNumber;
  }

  public String toString() {
    return(lastName + ", " + firstName + "..." + phoneNumber);
  }
}
```

1.7.2 Class ContactManager

A `ContactManager` keeps track of a collection of contacts organized in different categories, for example, contacts are grouped by the business type they run. One of the requirements is to disallow duplicate contacts, so we chose to use the *Set* pattern to represent contacts. The categories simply label subsets of the contacts collection, so we chose *Map* to associate names to collections of contacts. Additionally, we used

Sorter to sort the contacts alphabetically before listing them on the console:

```
package collections.example;

import java.io.*;
import java.lang.*;
import java.util.*;

public class ContactManager {
  private Set contacts;
  private Map categories;

  public ContactManager() {
    contacts = new HashSet();
    categories = new HashMap();
  }

  public ContactManager(int contactsCapacity, int categoriesCapacity) {
    contacts = new HashSet(contactsCapacity);
    categories = new HashMap(categoriesCapacity);
  }

  public void addCategory(String category) {
    if(categories.containsKey(category)) return;
    categories.put(category, new ArrayList());
  }

  public void addContact(Contact contact) {
    contacts.add(contact);
  }

  public boolean addContact(Contact contact, String category) {
    addContact(contact);
    List contactList = (List)categories.get(category);
    if(contactList == null) return false;
    return contactList.add(contact);
  }

  public Iterator getCategories() {
    return categories.keySet().iterator();
  }

  public Iterator getContacts() {
    return contacts.iterator();
  }

  public String toString() {
    // sort
    List list = Arrays.asList(contacts.toArray());
    Collections.sort(list, new Comparator() {
      public int compare(Object object1, Object object2) {
        int result;
        Contact contact1 = (Contact)object1;
        Contact contact2 = (Contact)object2;

        result = contact1.getLastName().compareTo(contact2.getLastName());
        if(result != 0) return result;
```

```
          result = contact1.getFirstName().compareTo(contact2.getFirstName());
          if(result != 0) return result;

          result = contact1.getPhoneNumber().compareTo(contact2.getPhoneNumber());
          return result;
        }
      });

      // print
      StringWriter result = new StringWriter();
      PrintWriter writer = new PrintWriter(result);
      Iterator iterator = list.iterator();
      while(iterator.hasNext()) {
        Contact contact = (Contact)iterator.next();
        writer.println(contact);
      }
      return result.toString();
    }

    public static void main(String[] args) {
      ContactManager manager = new ContactManager();
      manager.addContact(new Contact("Elvis", "Presley", "342-4535"));
      manager.addContact(new Contact("Tom", "Jones", "345-4542"));
      manager.addContact(new Contact("Jimmy", "Dean", "374-2398"));
      System.out.println(manager);
    }
  }
```

Bibliography

[1] Ken Auer and Kent Beck. Patterns for Efficient Smalltalk Programming. *Pattern Languages of Program Design 2*, 1996.

[2] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1998.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[4] Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison Wesley, 1997.

[5] Jeff Langr. *Essential Java Style Patterns for Implementation*. Prentice-Hall, 2000.