Chapter 13

Introduction to Data Types and Structures

Abstract Data Types and the Java Collections Framework

Outline

Abstract data types

Implementing an ADT

Java Collections Framework (JCF)

`Collection<E>` **and** `Set<E>` **interfaces**

Set implementations and examples

`List<E>` **and** `ListIterator<E>` **interfaces**

List implementations and examples

Map data type

`Map<K,V>` **interface**

Map implementations and examples

Recursion examples using maps

`Collections` **utility class**

Sorting examples

13.1 Introduction

In this chapter we consider abstract data types and their implementations. Simple examples include a fixed size bag ADT, a dynamic size bag ADT and a dynamic size array ADT. In each case simple versions of these ADTs are designed using Java interfaces and implemented using array data structures.

Next we give an overview of some of the important ADTs such as sets, lists and maps that are part of the Java Collections Framework (JCF). Here we concentrate on using the ADTs and not on how they are implemented, which is left for a course on data structures.

13.2 Abstract data types

A **data type** is a set of values (the data) and a set of operations defined on the data. An **implementation** of a data type is an expression of the data and operations in terms of a specific programming language such as Java or C++. An **abstract data type** (ADT) is a specification of a data type in a formal sense without regard to any particular implementation or programming language. Finally, a **realization** of an ADT involves two parts

- the interface, specification, or documentation of the ADT: what is the purpose of each operation and what is the syntax for using it.

- the implementation of the ADT: how is each operation expressed using the **data structures** and statements of a programming language.

The ADT itself is concerned only with the specification or interface details, not the implementation details. This separation is important. In order to use an ADT the client or user needs to know only what the operations do, not how they do it. Ideally this means that the implementation can be changed, to be more efficient for example, and the user does not need to modify programs that use the ADT since the interface has not changed.

With object-oriented programming languages such as Java and C++ there is a natural correspondence between a data type and a class. The class defines the set of operations that are permissible: they are the public methods of the class. The data is represented by the instance data fields. Each object (instance of the class) encapsulates a particular state: set of values of the data fields.

In Java the separation of specification and implementation details can easily be obtained using the Javadoc program which produces the specification (public interface) for each class. The user can simply read this documenation to find out how to use the class. It is also possible to use a Java interface for the specification of an ADT since this interface contains no implementation details, only method prototypes: any class that implements the interface provides a particular implementation of the ADT.

13.2.1 Classification of ADT operations

The various operations (methods) that are defined by an ADT can be grouped into several categories, depending on how they affect the data of an object:

Create operation

It is always necessary to create an object before it can be used. In Java this is done using the class constructors.

Copy operation

The availability of this operation depends on the particular ADT. In many cases it is not needed or desired. If present, the meaning (semantics) of the operation also depends on the particular ADT. In some cases copy means make a true copy of the object and all its data fields, and all their data fields, and so on, and in other cases it may mean to simply make a new reference to an object. In other words, the reference to the object is being copied, not the object itself. In this case there is only one object and it is shared among all the references to it. This makes sense for objects that occupy large amounts of memory and in many other cases as well. Both types of operation can even be included in the same ADT. In some languages the copy operation can have explicit and implicit versions. In Java the implicit operation, defined by assignment or method argument passing, always copies references but it is possible to make other kinds of explicit copies using a copy constructor or by overriding the `clone` method inherited from the `Object` class.

Destroy operation

Since objects take up space in memory it is necessary to reclaim this space when an object is no longer needed. This operation is often called the *destroy* operation. In Java there is no explicit destroy operation since the built-in garbage collector takes on this responsibility: when there are no more references to an object it is eventually garbage-collected.

Modification operations

Every object of an ADT encapsulates data and for some ADTs we need operations that can modify this data. These operations act on objects and change one or more of their data fields. Sometimes they are called **mutator** operations. If an ADT has no mutator operations then the state cannot be changed after an object has been created and the ADT is said to be **immutable**, otherwise it is **mutable**.

Inquiry operations

An inquiry operation inspects or retrieves the value of a data field without modification. It is possible to completely hide all or part of the internal state of an object simply by not providing the corresponding inquiry operations.

13.2.2 Pre- and post-conditions

To document the operations of an ADT pre-conditions and post-conditions can be used.

Pre-conditions They are the conditions that must be true before an operation is executed in order that the operation is guaranteed to complete successfully. These conditions can be expressed in terms of the state of the object before the operation is applied to the object. A pre-condition may or may not be needed.

Post-conditions They are the conditions that will be true after an operation completes successfully. These conditions can be expressed in terms of the state of the object after the operation has been applied to the object.

Together the pre- and post-conditions form a contract between the implementer of the method and the user of the method.

13.2.3 Simple ADT examples

The simplest examples of ADTs are the numeric, character, and boolean types. Most programming languages have realizations of them as fundamental types which are used to build more complex structured ADTs. Some typical types in these categories are

An integer ADT

Mathematically the data values here can be chosen as all integers n such that $-\infty \le n \le \infty$. Another possibility is to consider only non-negative integers n satisfying $0 \le n \le \infty$.

A typical set of operations might be the standard arithmetic operations *add*, *subtract*, *multiply*, *integer quotient* and *integer remainder*, boolean valued operations such as *equal*, *notEqual*, and the relational operators $<$, $\le$, $>$, $\ge$. An assignment operation would also be needed.

These are infinite data types since there are an infinite number of integers. Therefore any realization would need to restrict the data values to a finite subset. Some common possibilities are 8-bit, 16-bit, 32-bit, or 64-bit representations which may be signed or unsigned (non-negative values).

For example, in Java there is an 8-bit `byte` type with range $-2^7 \le n \le 2^7 - 1$, a 16-bit `short` type with range $-2^{15} \le n \le 2^{15} - 1$, a 32-bit `int` type with range $-2^{31} \le n \le 2^{31} - 1$, and a 64-bit `long` type with range $-2^{63} \le n \le 2^{63} - 1$.

A floating point ADT

Here the data values are floating point numbers. In scientific notation a floating point number would have the form $x = m \times 10^e$ where m is the mantissa and e is the exponent.

A typical set of operations would be similar to those for integers except the divide operation is now a floating point division. An assignment operation would also be needed.

For example, in Java there is a single precision 32-bit `float` type and a double precision 64-bit `double` type. The standard IEEE representation is complicated but necessary to ensure that floating point arithmetic is portable. Most processors support this standard. A single precision number x is either 0, $-3.40 \times 10^{38} \le x \le -1.40 \times 10^{-45}$ or $1.40 \times 10^{-45} \le x \le 3.40 \times 10^{38}$. A double precision number x is either 0, $-1.80 \times 10^{308} \le x \le -4.94 \times 10^{-324}$ or $4.94 \times 10^{-324} \le x \le 1.80 \times 10^{308}$.

A character ADT

Here the data is the set of characters from some character set such as ASCII or Unicode. Internally each character is represented by an unsigned integer n in the range $0 \leq n \leq N$ for some N.

A typical set of operations might include operations to convert from upper case to lower case and vice versa, operations to compare two characters to see if they are equal or to see if one precedes another in the lexicographical ordering defined on the characters, or an assignment operation.

For example, in Java the `char` type is an unsigned 16-bit integer type with Unicode character code n satisfying $0 \leq n \leq 65535$.

A boolean ADT

Here there are only two data values which can be denoted by false and true. Other possibilities are to use 0 for false and 1 for true, or 0 for false and any non-zero number for true.

A typical set of operations would be an assignment operation, an operation to test for false and one to test for true.

13.2.4 Some common structured ADTs

A structured ADT is one that is defined in terms of another ADT using to some data structure. For example, an array of integers would be defined in terms of an integer ADT and a string ADT would be defined in terms of a character ADT. These two structured ADTs are the most common and are available in most programming languages.

The array ADT

An array consists of n elements $[a_0, a_1, \ldots, a_{n-1}]$. Here the data consists of these arrays and each array element a_k belongs to some other ADT. The subscript k is called the array index. The starting index may be 0, 1, or user defined. In C++ and Java array indices begin at index 0.

The basic array operations are to *get* the value of the k-th element and *set* a new value for the k-th element. In C++ and Java the *get* operation is denoted by `x = a[k]` and the *set* operation is denoted by `a[k] = x`. This also means that an array is a mutable ADT.

The standard array ADT is of fixed size: once created its size cannot be changed. The standard arrays in C++ and Java are of this type. However we will see that it is easy to create a dynamic array ADT (resizable) which can be expanded in size if needed to accommodate more elements.

The string ADT

Strings are like arrays of characters but the operations can be quite different. Both mutable and immutable string ADTs are common. For example, in Java the `String` class represents an immutable fixed size ADT and the `StringBuilder` class represents a dynamic mutable ADT.

Some immutable string operations are to *get* the k-th character, construct a substring, construct upper case or lower case versions, and compare two strings using the lexicographical order defined on the underlying character set.

Some mutable operations are to *set* the k-th character to a new value, and *append* a character or string to the end of a string.

13.2.5 User defined ADT examples

We are not limited to the standard ADTs that have implementations already available in a computer language or a system defined library of ADTs. We can write our own specifications for an ADT and implement it in any language. Here we give two examples. We will show how to implement them in Java.

A dynamic array ADT

Here the data elements are arrays $[a_0, a_1, \ldots, a_{n-1}]$. This is a mutable ADT and the basic operations would be *get*, to get the k-th array element, and *set*, to set a new value for the k-th array element. Also the array size can be increased automatically as needed (doubled in size when full, for example) or by applying some expand operation that increases the array size by a specified amount.

A bag ADT

Here the data elements are bags. Each bag is a container that holds a collection of elements of some type. There is no defined order on the bag elements as there are for arrays. In mathematics a bag is often called a multi-set (no order, but duplicate elements are allowed) in contrast to sets for which there can be no duplicates.

Bags are usually designed to be mutable and dynamic so a basic set of operations are *add*, to add another element to a bag, *remove*, to remove a specified element from a bag, and *contains* which tests if a specified element is in a bag.

13.3 Implementing an ADT

We now show how to implement the bag and dynamic array ADTs. The first step is to write a specification or design of the data type, indicating what each operation does. This could be done with a Java interface followed by the design of the class implementing the interface, indicating each constructor and method body by $\{\ldots\}$.

Whether an interface is being used or not the class design should always include constructor prototypes since they are never included in an interface.

Once the design is finished it is possible to write some statements that use the ADT to 'try out' the syntax of the operations as given by the instance method prototypes. Finally, the implementation must be written (data fields, constructor and method bodies). This involves choosing some data structure to represent the data encapsulated by the objects.

In Java all data types except for the eight primitive ones (`byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`) are expressed as objects from some class. This presents a problem in the design of a generic type since generic types must be object types (reference types) and we cannot

directly use the `int` type as a generic type. To allow primitive types to be used as objects there are wrapper classes in Java for each primitive type. For example the `Integer` class can be used as an object version of the `int` type. In Java 5 auto boxing and unboxing make this easy.

Finally, when the implementation is complete, its operations must be tested.

13.3.1 Implementation of the `Bag<E>` ADT

First we write a fixed size implementation of the bag ADT called `FixedBag<E>` using the generic type `E` for the elements in the bag. This means that once constructed for a given maximum size (number of elements) this size cannot be changed. Then we will make a simple modification to obtain a dynamic version called `DynamicBag<E>`.

Designing the `Bag<E>` ADT

Here we illustrate the use of an interface to specify the design of an ADT. Both the fixed size and dynamic versions of the ADT will implement the following interface.

Interface `Bag<E>`

book-project/chapter13/bags

```
package chapter13.bags;
/**
 * A simple mutable generic bag ADT.
 * @param <E> type of elements in the bag
 */
public interface Bag<E>
{
   /**
    * Return current number of elements in this bag.
    * @return current number of elements in this bag
    */
   int size();

   /**
    * Return true if this bag is empty else false.
    * @return true if this bag is empty else false
    */
   boolean isEmpty();

   /**
    * Add another element to this bag if there is room.
    * @param element the element to add
    * @return true if add was successful else false.
    */
   boolean add(E element);

   /**
    * Remove a given element from this bag.
    * @param element the element to remove
```

```
     * @return true if the element was removed.
     * A false return value occurs if element was
     * not in this bag.
     */
   boolean remove(E element);

   /**
     * Check if a given element is in this bag.
     * @param element the element to check
     * @return true if element is in this bag else false
     */
   boolean contains(E element);
}
```

We have not included the `public` modifier on the method prototypes in the interface. It is redundant since all methods in an interface are public.

Designing a fixed size implementation

The fixed size bag implementation has the form

```
public class FixedBag<E> implements Bag<E>
{
   // instance data fields will go here

   public FixedBag(int bagSize) {...}
   public FixedBag() {...}
   public FixedBag(FixedBag<E> b) {...}

   public int size() {...}
   public boolean isEmpty() {...}
   public boolean add(E element) {...}
   public boolean remove(E element) {...}
   public boolean contains(E element) {...}

   public String toString() {...}
}
```

Javadoc comments have been omitted. They are shown later in the final version of the class. Here we have three constructors. The first specifies the maximum number of elements that can be added to the bag and the no-arg constructor gives a bag with a maximum size of 10 elements. The third constructor is called a **copy constructor**. Its purpose is to construct a copy of the bag given by the argument `b`.

The `toString` method is used to return a string representation of the elements in the bag. We didn't need to include the `toString` prototype in the `Bag<E>` interface since every class inherits a `toString` method.

Also, for this fixed size implementation the `add` method would return false if the bag is already full.

According to this design we can construct a bag containing a maximum of 5 integers and add the integers 1, 2, and 3 to it using the statements

```
Bag<Integer> b = new FixedBag<Integer>(5);
b.add(1); b.add(2); b.add(3);
System.out.println(b);
```

Autoboxing is being used here: the compiler understands that `b.add(1)` means to replace `1` by the wrapper class object `new Integer(1)` and use `b.add(new Integer(1))`.

It is important to use the interface type on the left side of the constructor statement. This makes it easier to switch to another implementation class, such as a dynamic one in this case. This is sometimes called "programming to an interface".

Our bag design is minimal. For example it is not possible with this design to take a bag of integers and remove all even integers or display the bag elements one per line. This would require an iterator and will be discussed later.

■ EXAMPLE 13.1 **(Filling a fixed size bag)** The statements

```
Bag<Integer> bag = new FixedBag<Integer>(10);
for (int k = 1; k <= 10; k++)
   bag.add(k);
```

construct a fixed bag of size 10 and fill it with the numbers 1 to 10. ■

■ EXAMPLE 13.2 **(Filling a fixed size bag)** The statements

```
Bag<Integer> bag = new FixedBag<Integer>(10);
int k = 1;
while (bag.add(k))
   k++;
```

construct a fixed bag of size 10 and fill it with the numbers 1 to 10 using the `add` method to detect when the bag is full. ■

Choosing a data structure

The next step is to choose a data structure to hold the bag elements. Here we choose a fixed size array called `data` such that if the number of elements currently in the bag is `size` then these elements are stored in `data[0]`, `data[1]`, ..., `data[size-1]` and the remaining array elements `data[size]`, ..., `data[data.length-1]` are free for storing more elements. Therefore we choose the following instance data fields for the bag data.

```
private E[] data;
private int size;
```

As elements are added to the bag they are stored in the next available place in the array. Thus at any stage the array consists of two parts: the used part `data[0]` to `data[size-1]` and the unused part `data[size]` to `data[data.length-1]`.

Implementing the constructors

The first constructor implementation is

```
public FixedBag(int bagSize)
{
   data = (E[]) new Object[bagSize];
   size = 0;
}
```

and the second constructor calls this one. When constructing an array of generic type it is necessary to use the actual `Object` type for the array elements and typecast it to the type `E`. For various technical reasons related to the way generic types were added to the Java language the statement

```
data = (E[]) new E[bagSize];
```

is illegal.

Finally, the copy constructor is given by

```
public FixedBag(FixedBag<E> b)
{
   size = b.size();
   data = (E[]) new Object[b.data.length];
   for (int k = 0; k < size; k++)
      data[k] = b.data[k];
}
```

Here we first construct an array of the same maximum size `b.data.length` of the array `b`. Then the bag elements in `b` are copied into this array.

Implementing the methods

The `add` method first checks if there is room for the new element. Since `size` represents the number of elements currently in the `data` array then the new element can use `data[size]`. The implementation is

```
public boolean add(E element)
{
   if (size == data.length) // full bag
      return false;
   data[size] = element;
   size = size + 1;
   return true;
}
```

The `remove` method needs to use a loop to search for the element to remove. If the element is found at position `k` in the array then the obvious way to remove it is to use a for-loop to copy the array elements `data[k+1]`, ..., `data[size-1]` down one location to overwrite the element at position `k`. This requires another loop.

A more efficient way is to realize that a bag is not an ordered structure so the array ordering does not need to be preserved. Therefore we can just overwrite the element at position `k` with the last array element at position `size-1`. This effectively removes the element at position `k`. This gives the implementation

```
public boolean remove(E element)
{
   for (int k = 0; k < size; k++)
   {
      if (data[k].equals(element))
      {
         data[k] = data[size-1];
         size = size - 1;
         return true;
      }
   }
   return false; // not found
}
```

It is necessary to use the `equals` method defined for element type `E` to properly test for element equality. The test `data[k] == element` will not work. A class that does not have a properly defined `equals` method can not be used as the element type. The wrapper classes and the `String` class all have `equals` methods.

The remaining methods are easily implemented and the complete implementation class is

Class `FixedBag<E>`

book-project/chapter13/bags

```
package chapter13.bags;
/**
 * A simple fixed size bag implementation.
 * @param <E> type of elements in the bag
 */
public class FixedBag<E> implements Bag<E>
{
   // This version uses a fixed array for the bag

   private E[] data;
   private int size;

   /**
    * Create a bag for a given maximm number of elements.
    * @param bagSize the maximum number of elements
    */
   public FixedBag(int bagSize)
   {
      data = (E[]) new Object[bagSize];
      size = 0;
   }
```

```
/**
 * Create a default bag for a maximum of 10 elements
 */
public FixedBag()
{
   this(10);
}

/**
 * Construct a bag that is a copy of a given bag.
 * The copy has the same maximum size as bag b.
 * @param b the bag to copy
 */
public FixedBag(FixedBag<E> b)
{
   size = b.size();
   data = (E[]) new Object[b.data.length];
   for (int k = 0; k < size; k++)
      data[k] = b.data[k];
}

public int size()
{
   return size;
}

public boolean isEmpty()
{
   return size == 0;
}

public boolean add(E element)
{
   if (size == data.length)
      return false;
   data[size] = element;
   size = size + 1;
   return true;
}

public boolean remove(E element)
{
   for (int k = 0; k < size; k++)
   {
      if (data[k].equals(element))
      {
         // nice trick
         data[k] = data[size-1];
         size = size - 1;
         return true;
      }
```

```
      }
      return false; // not found
   }

   public boolean contains(E element)
   {
      for (int k = 0; k < size; k++)
         if (data[k].equals(element))
            return true;
      return false; // not found
   }

   /**
    * Return a string representation of this bag.
    * @return a string representation of this bag.
    */
   public String toString()
   {
      StringBuilder sb = new StringBuilder();
      sb.append("[");
      if (size != 0)
      {
         sb.append(data[0]);
         for (int k = 1; k < size; k++)
         {
            sb.append(",");
            sb.append(data[k]);
         }
      }
      sb.append("]");
      return sb.toString();
   }
}
```

We have not included comments for the interface methods since they are already given in the `Bag<E>` interface.

Converting to a dynamic implementation

We now convert the fixed size implementation to a dynamic one. This can easily be done by modifying the `add` method to automatically expand the `data` array whenever it it is full. The new version of `add` is

```
public boolean add(E element)
{
   if (size == data.length)
      resize();
   data[size] = element;
   size = size + 1;
   return true;
}
```

Here we call a `resize` method that increases the capacity as follows: (1) make a new data array twice the size of the current one, (2) copy the current data array to the beginning of the new one, (3) reassign the `data` reference to the new array (the old one will be garbage collected).

This gives the following private method.

```
private void resize()
{
   int newCapacity = 2 * data.length;
   E[] newData = (E[]) new Object[newCapacity]; // step 1
   for (int k = 0; k < data.length; k++) // step 2
      newData[k] = data[k];
   data = newData; // step 3
}
```

Here is the complete implementation.

Class `DynamicBag<E>`

book-project/chapter13/bags

```
package chapter13.bags;
/**
 * A simple dynamic bag implementation.
 * @param <E> the type of elements in the bag
 */
public class DynamicBag<E> implements Bag<E>
{
   private E[] data;
   private int size;

   /**
    * Create a bag with a given initial capacity.
    * @param initialCapacity the initial capacity of this bag
    */
   public DynamicBag(int initialCapacity)
   {
      data = (E[]) new Object[initialCapacity];
      size = 0;
   }

   /**
    * Create a default bag with an initial capacity of 10 elements.
    */
   public DynamicBag()
   {
      this(10);
   }

   /**
    * Construct a bag that is a copy of a given bag.
    * The copy has the same current maximum size as bag b.
```

```
     * @param b the bag to copy
     */
   public DynamicBag(DynamicBag<E> b)
   {
      size = b.size();
      data = (E[]) new Object[b.data.length];
      for (int k = 0; k < size; k++)
         data[k] = b.data[k];
   }

   public int size() {...} // same as for FixedBag
   public boolean isEmpty() {...} // same as for FixedBag

   public boolean add(E element)
   {
      if (size == data.length)
         resize();
      data[size] = element;
      size = size + 1;
      return true;
   }

   public boolean remove(E element) {...} // same as for FixedBag
   public boolean contains(E element) {...} // same as for FixedBag

   private void resize()
   {
      // Make a new array twice as big as current one,
      // copy data to it and make data reference the new one.

      int newCapacity = 2 * data.length;
      E[] newData = (E[]) new Object[newCapacity];
      for (int k = 0; k < data.length; k++)
         newData[k] = data[k];
      data = newData;
   }

   public String toString() {...} // same as for FixedBag
}
```

13.3.2 Implementation of the `DynamicArray` ADT

We have written a `FixedBag<E>` ADT but we will not consider a `FixedArray<E>` ADT since the built-in `array` type is a fixed size implementation.

Unlike a bag, an array is an ordered ADT. There is a first element, a second element, and so on so there is an index associated with each array element.

Designing the `Array` ADT

As for the Bag ADT we can use the following interface to design a simple array ADT

Interface `Array<E>`

book-project/chapter13/arrays

```
package chapter13.arrays;
/**
 * A simple generic array ADT.
 * @param <E> type of elements in the array
 */
public interface Array<E>
{
   /**
    * Return current number of elements in this array.
    * @return current number of elements in this array
    */
   int size();

   /**
    * Return true if this array is empty else false.
    * @return true if this array is empty else false
    */
   boolean isEmpty();

   /**
    * Add another element to end of this array.
    * @param element the element to add to end at position size().
    * @return true if add was successful else false.
    */
   boolean add(E element);

   /**
    * Get the element at a given index (0,1,...).
    * @param index the index of the element
    * @return the element at the index
    * @throws ArrayIndexOutOfBoundsException if the
    * index is out of the range 0 <= index < size()
    */
   E get(int index);

   /**
    * Set a new value for a given array element.
    * @param index the index of the array element
    * @param element the new value of the element
    * @throws ArrayIndexOutOfBoundsException if the
    * index is out of the range 0 <= index < size()
    */
   void set(int index, E element);
}
```

Here we have an `add` method that adds an element at the end of the array (position `size()`). It is important that we specify that the element be added at the end of the array. This was not necessary for the bag ADT.

The element at position k can be obtained using the get method and the set method can be used to give a new value to the object associated with position k. If an index k is outside the range 0 <= k < size() then an ArrayIndexOutOfBounds exception is thrown.

The operations defined for an array ADT are quite different than those for a bag ADT since the array ADT is an ordered collection of elements and there is no assumed order for the elements in the bag. The get and set methods were not part of the bag ADT since there is no concept of an index for the elements in a bag.

This is a minimal array interface and there are many other methods such as a remove method that removes the element at a given index, and indexOf that returns the index of a given elememt.

Designing a dynamic implementation

The dynamic array implementation has the form

```
public class DynamicArray<E> implements Array<E>
{
   private E[] data;
   private int size;

   public DynamicArray(int initialCapacity) {...}
   public DynamicArray() {...}
   public DynamicArray(DynamicArray<E> a) {...}

   public int size() {...}
   public boolean isEmpty() {...}
   public boolean add(E element) {...}
   public E get(int index) {...}
   public void set(int index, E element) {...}
   public String toString() {...}
}
```

Here we use the same data structure, a fixed array, as for the bag implementations. The constructors are very similar to the DynamicBag constructors.

Using the design

Now we can try out some statements for our dynamic array design.

■ EXAMPLE 13.3 **(Resizing a dynamic array)** The following statements test that an array is resized when it becomes full. Autoboxing is used to convert integers to the Integer object type.

```
Array<Integer> a = new DynamicArray<Integer>(3);
a.add(1); a.add(2); a.add(3); a.add(4);
System.out.println("Array size is " + a.size());
System.out.println(a);
```

Here the initial capacity is 3. When we add the 4-th number the capacity is doubled to 6 and the number 4 is added to the array, which now has size 4 and room for two more elements. ■

■ EXAMPLE 13.4 **(Summing the elements in a dynamic array)** Unlike the bag we can loop over the elements in the array by using the `get` method. Here we construct an integer array and sum its elements using the following statements.

```
Array<Integer> a = new DynamicArray<Integer>(3);
a.add(1); a.add(2); a.add(3); a.add(4);
int sum = 0;
for (int k = 0; k < a.size(); k++)
   sum = sum + a.get(k);
System.out.println("The sum of the elements is " + sum);
```

Compare these statements with the following ones that do the same thing with a standard array:

```
int[] a = new int[4];
a[0] = 1; a[1] = 2; a[2] = 3; a[3] = 4;
int sum = 0;
for (int k = 0; k < a.length; k++)
   sum = sum + a[k];
```

Here we need to use the exact size of 4. ■

■ EXAMPLE 13.5 **(Swapping two elements of an array)** Assuming that `str` is an array of strings, the statements

```
String temp = str.get(i);
str.set(i, str.get(j));
str.set(j, temp);
```

swap the strings at positions `i` and `j`. ■

Implementing the constructors and methods

This is the same as for `DynamicBag<E>` and the implementation of the `get` and `set` methods are simple so we have the following class.

Class `DynamicArray<E>`

book-project/chapter13/arrays

```
package chapter13.arrays;
/**
 * A simple dynamic array implementation.
 * @param <E> type of elements in the array
 */
public class DynamicArray<E> implements Array<E>
{
   private E[] data;
   private int size;
```

```
/**
 * Create an array for a given initial capacity.
 * @param initialCapacity the initial capacity
 */
public DynamicArray(int initialCapacity)
{
   data = (E[]) new Object[initialCapacity];
   size = 0;
}

/**
 * Create a default array for an initial capacity of 10 elements.
 */
public DynamicArray()
{
   this(10);
}

/**
 * Construct an array that is a copy of a given array.
 * The copy has the same capacity as array a.
 * @param a the array to copy
 */
public DynamicArray(DynamicArray<E> a)
{
   size = a.size();
   data = (E[]) new Object[a.data.length];
   for (int k = 0; k < size; k++)
      data[k] = a.data[k];
}

public int size()
{
   return size;
}

public boolean isEmpty()
{
   return size == 0;
}

public boolean add(E element) {...} // same as for DynamicBag

public E get(int index)
{
   if (0 <= index && index < size)
      return data[index];
   else
      throw new ArrayIndexOutOfBoundsException("index out of bounds");
}

public void set(int index, E element)
```

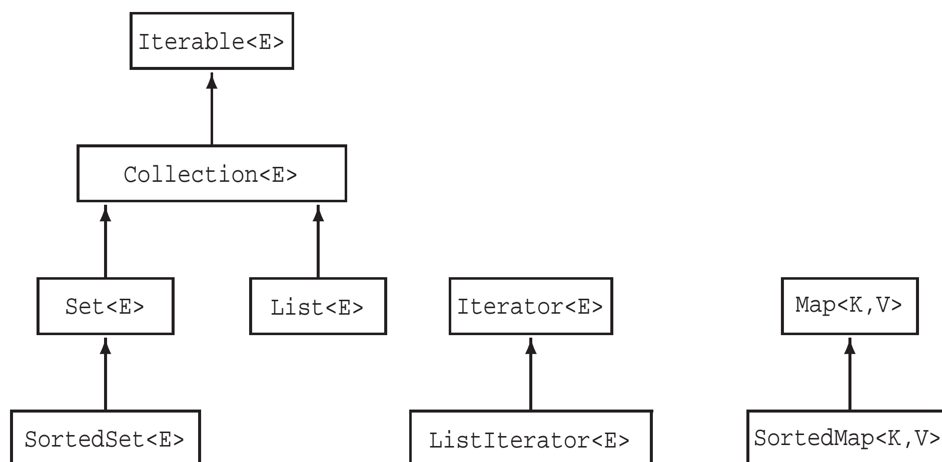

Figure 13.1: JCF related interface hierarchy

```
   {
      if (0 <= index && index < size)
         data[index] = element;
      else
         throw new ArrayIndexOutOfBoundsException("index out of bounds");
   }

   private void resize() {...} // same as for DynamicBag
   public String toString() {...} // same as for FixedBag
}
```

13.4 Java Collections Framework (JCF)

Many ADTs collect together elements of some data type. The simplest examples we have considered are the bag ADT and the array ADT. We define a **collection** as a data type that organizes a group of related objects called the elements of the collection and provides operations on them. There are often restrictions on the elements that belong to a specific kind of collection and on the way the elements can be accessed.

13.4.1 Interface hierarchy

In Java collections are represented by classes that implement the `Collection<E>` interface or one of its extended interfaces such as `Set<E>` or `List<E>`. These interfaces and others make up what is called the JCF (Java Collections Framework) and their relationship is shown in Figure 13.1. Here the arrow means "extends". For example the `Set<E>` interface extends `Collection<E>`.

A set is an example of a collection whose elements have the following two properties: (1) no defined order and (2) duplicate elements are not allowed. This corresponds to the mathematical definition of a set.

```
public interface Collection<E> extends Iterable<E>
{
   // Query operations
   int size();
   boolean isEmpty();
   boolean contains(Object obj);
   Iterator<E> iterator();
   Object[] toArray();
   <T> T[] toArray(T[] a);

   // Modification Operations
   boolean add(E element); // optional
   boolean remove(Object obj); // optional

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

   // Comparison and hashing
   boolean equals(Object obj);
   int hashCode();
}
```

Figure 13.2: The `Collection<E>` interface

A bag is another example of a collection that, like a set, imposes no defined order on its elements but does allow duplicate elements. In mathematics a bag is called a multi-set. The bag is the simplest kind of collection class since it imposes no restrictions or structure on its elements.

Arrays and lists are collections in which the elements do have a defined order. There is a first element, a second element, and so on, and duplicates are allowed. In mathematics an array or list is often called a sequence.

We shall give a survey of the most important classes in the Java Collections Framework (JCF). Our goal is not to understand the implementation of these classes, which is left to a data structures course, but to learn how to use them. Of course, we should not need to understand implementation details in order to use a class.

The most important interface in the JCF is the `Collection<E>` interface which represents the basic design and methods any collection class should have. A class that implements this interface "is a" collection. A summary of this interface is given in Figure 13.2. It also extends another interface called `Iterable<E>`, given in Figure 13.3 and this interface contains one method called `iterator` which returns an object from a class that implements the `Iterator<E>` interface shown in Figure 13.4. We now discuss these three interfaces.

```
public interface Iterable<E>
{
   Iterator<E> iterator();
}
```

Figure 13.3: The `Iterable<E>` interface

```
public interface Iterator<E>
{
   boolean hasNext();
   E next();
   void remove(); // optional
}
```

Figure 13.4: The `Iterator<E>` interface

13.4.2 Traversing a collection with an iterator

An important operation on a collection is to be able to traverse it. This means to examine or process elements in the collection one at a time using some kind of loop. This is the purpose of an **iterator**.

Our simple `Bag<E>` interface did not define an iterator so for classes such as `FixedBag<E>` and `DynamicBag<E>` there was no way to process the elements one by one in some order. We could do this for the `DynamicArray<E>` class only because we had an indexed collection so we could use a standard for-loop to traverse an array as shown in Example 13.4.

In the JCF an iterator is an object of some class that implements the `Iterator<E>` interface shown in Figure 13.4. A collection class will normally not implement this interface directly. Instead it will provide an `iterator()` method that returns an object of some class that implements the `Iterator<E>` interface. This is the case for the `Collection<E>` interface shown in Figure 13.2 (under query operations).

In the `Iterator<E>` interface the `hasNext()` method is used to stop the iteration process and the `next()` method returns the current element in the collection and advances to the next one. This means that we can call `next()` repeatedly as long as `hasNext()` returns true.

■ EXAMPLE 13.6 **(Using an iterator to traverse a collection)** We can use statements such as the following to process the elements.

```
Collection<E> c = new ACollectionClass<E>(...);
c.add(e1); c.add(e2); c.add(e3); // ...
Iterator<E> iter = c.iterator();
while(iter.hasNext())
{
   E element = iter.next();
   // do something here with element
}
```

Here `ACollectionClass` is any class that implements the `Collection<E>` interface. ■

The `Iterator<E>` interface also contains a `remove` operation which is listed as optional. If an implementing class does not support the removal of elements from the collection then an `UnsupportedOperationException` will be thrown. Such an iterator is said to be immutable.

■ EXAMPLE 13.7 **(Using an iterator as a filter)** The following statements show how an iterator can be used as a **filter** by removing elements from the collection that satisfy some condition.

```
Collection<E> c = new ACollectionClass<E>(...);
c.add(e1); c.add(e2); c.add(e3); // ...
Iterator<E> iter = c.iterator();
while(iter.hasNext())
{
   E element = iter.next();
   if (removal condition is true)
   {
      iter.remove();
   }
}
```

Here it is important that the `remove()` method is used after a call to `next()`. ■

■ EXAMPLE 13.8 **(Using an iterator as a filter without remove)** If removal is not supported then a filter can be written by creating a new collection containing only the elements that were not removed:

```
Collection<E> c = new ACollectionClass<E>(...);
c.add(e1); c.add(e2); c.add(e3); // ...
// create a new empty collection
Collection<E> newCollection = new ACollectionClass<E>();
Iterator<E> iter = c.iterator();
while(iter.hasNext())
{
   E element = iter.next();
   if (removal condition is NOT true)
      newCollection.add(element);
}
```

Here the original collection is not changed. ■

An important property of an iterator is that it does not expose any internal details of the collection and the data structures used in the implementation. This is important since it means that the implementation of the collection class can be changed without changing the iterator.

13.4.3 Iterable<E> interface

The Iterable<E> interface is related to the for-each loop introduced in Java 5. If a class implements this interface then it provides an iterator() method defining an iterator and the for-each loop can be applied as follows

■ EXAMPLE 13.9 **(Using a for-each loop as an immutable iterator)** The for-each loop has the syntax

```
for (E element : c)
{
   // do something here with element
}
```

Here c is any object from a class that implements the Iterable<E> interface. In particular it can be of type Collection<E>. The for-each loop cannot access the remove() method so it can only be used for immutable traversals. ■

■ EXAMPLE 13.10 **(Using an iterator with a standard array type)** The built-in array type also implements Iterable<E> so it is possible to process an array using statements such as

```
String[] s = new String[3];
s[0] = "one"; s[1] = "two"; s[2] = "three";
for (String str : s)
{
   // do something here with the string str
}
```

This is useful as a replacement for the standard for-loop that does not actually use its index in the body of the loop. The for-each loop requires no index. ■

13.5 Collection<E> and Set<E> interfaces

13.5.1 Collection<E> interface

We now summarize the methods in the Collection<E> interface in Figure 13.2. For more complete descriptions see the Java API documentation. As shown in the figure the operations can be divided into four categories: (1) Query operations, (2) Modification operations, (3) Bulk operations, and (4) Comparison and hashing.

Some methods are optional. If a class does not want to implement an optional method the method must throw an UnsupportedOperationException if it is called. Note that the optional operations are precisely the ones which may modify this collection, so if a class implements none of these methods then it is implementing immutable collections. Here is a summary of the Collection<E> methods.

Note that the contains and remove methods have an argument of type Object instead of E. This is conventional since these methods do not add new elements to the collection. However, the

add method must have an argument of type E to guarantee that the collection will only contain elements of type E.

- **int size()**

  Return the number of elements in **this** collection.

- **boolean isEmpty()**

  Returns true if there are no elements in **this** collection else returns false.

- **boolean contains(Object obj)**

  Returns true if **this** collection contains element **obj** else returns false.

- **Iterator<E> iterator()**

  Return an iterator of type **Iterator<E>** for **this** collection. This is the method that is necessary to implement the **Iterable<E>** interface.

- **Object[] toArray()**

  Convert the elements in **this** collection to an array of **Object** type.

  For example, if c is a collection of strings then the statement

  ```
  Object[] s = c.toArray();
  ```

  converts the collection of strings to the array s of objects s[0], ..., s[s.length-1]. To recover the strings it is necessary to use a typecast on each component such as

  ```
  String str = (String) s[k];
  ```

- **<T> T[] toArray(T[] a)**

  This is a parametrized method for type **T** that returns an array **T[]** of type **T**.

  If the parametrized type of the collection is **T** as indicated by the argument **a** then this method converts the elements of **this** collection to an array of type **T** which is the run-time type of the array. If the collection does not contain elements of type **T** an exception is thrown.

  For example, if c is a collection of strings then the statement

  ```
  String[] s = c.toArray(new String[c.size()]);
  ```

  converts the collection of strings to the array s of strings s[0], ..., s[s.length-1].

- **boolean add(E element)**

  Returns true if **this** collection was changed (**element** was added) after calling the method else returns false. This is an optional operation.

- **boolean remove(Object obj**

  Returns true if **this** collection was changed (**obj** was found and removed) after calling the method else returns false. This is an optional operation.

```
public interface Set<E> extends Collection<E>
{
   // The Collection<E> interface methods can go here
   // The Set<E> interface introduces no new methods
}
```

Figure 13.5: The `Set<E>` interface

- **`boolean containsAll(Collection<?> c)`**

  Returns true if **`this`** collection contains all the elements in collection **`c`** else returns false. The notation **`Collection<?>`** means a collection of any type (**`?`** is a wild card).

- **`boolean addAll(Collection<? extends E> c)`**

  Adds all of the elements of **`c`** to **`this`** collection. Returns true if **`this`** collection was modified after calling the method else returns false. The notation **`Collection<? extends E>`** means a collection of any type that extends or implements the type **`E`**. In this context **`extends`** means "extends or implements". This is an optional operation.

- **`boolean removeAll(Collection<?> c)`**

  Returns true if **`this`** collection was modified (one or more elements of **`c`** were removed from **`this`** collection) after calling the method else returns false. This is an optional operation.

- **`boolean retainAll(Collection<?> c)`**

  Retains only the elements in **`this`** collection that are also in **`c`**. Returns true if this collection was modified after calling the method else returns false. This is an optional operation.

- **`void clear()`**

  Remove all elements of **`this`** collection to give an empty collection. This is an optional operation.

- **`boolean equals(Object obj)`**
  `int hashCode()`

  These are methods in the **`Object`** class that can be overridden. The **`equals`** method tests if two collections have the same elements.

13.5.2 `Set<E>` interface

The `Collection<E>` interface describes what is called a bag or multi-set since there is no structure imposed on the elements in the collection.

The `Set<E>` interface is given in Figure 13.5. It extends `Collection<E>` but does not introduce any new methods. However the documentation of some of the methods changes since a set is a collection that does not contain duplicates. For example, the `contains` method will return false

if the element `obj` is already in `this` set and the `add` method will not change the collection if the element `obj` is already in `this` set.

Similarly the `addAll` method will only add to `this` set the elements of the collection `c` that are not already in `this` set.

Set theory interpretation of the bulk set methods

The bulk `Set<E>` methods can be used to implement the basic set theory operations of subset, set difference, intersection, and union.

> **subset/superset** If a and b are two sets then $a \subseteq b$ (or equivalently $b \supseteq a$) means that a is a subset of b (or equivalently b is a superset of a). In other words every element in a is also an element of b.
>
> This can be expressed using `containsAll`. If `a` and `b` are two sets (objects from a class that implements `Set<E>`) then `a.containsAll(b)` returns true only if $a \supseteq b$, so `containsAll` is the superset operation.
>
> **set difference** If a and b are two sets then $a - b$ is the difference: set of all elements in a that are not in b. A destructive version is represented by `a.removeAll(b)`, which replaces a by $a - b$.
>
> **set union** If a and b are two sets then $a \cup b$ is their union: set of all elements in a or b or both. A destructive version is represented by `a.addAll(b)`, which replaces a by $a \cup b$.
>
> **set intersection** If a and b are two sets then $a \cap b$ is their intersection: set of all elements that are in a and in b. A destructive version is represented by `a.retainAll(b)`, which replaces a by $a \cap b$.

To obtain non-destructive versions (`a` is not changed) it is necessary to make a copy of `a` and apply the operation to the copy.

13.6 Set Implementations and examples

The JCF includes several implementations of the `Set<E>` interface. We will consider three of them: `HashSet<E>`, `LinkedHashSet<E>`, and `TreeSet<E>`. The `HashSet<E>` implementation is the fastest but if a total order can be defined on the elements of the set then `TreeSet<E>` can be used to maintain the set in sorted order unlike `HashSet<E>` which maintains no order. If the element order is not important use `HashSet<E>`. The `LinkedHashSet<E>` class maintains the elements in the order they were added to the set.

13.6.1 `HashSet<E>` implementation of `Set<E>`

A summary of the `HashSet<E>` implementation is given in Figure 13.6. We will not discuss any implementation details. There are four constructors. The first constructor with no arguments

```
public class HashSet<E> extends AbstractSet<E>
   implements Set<E>, Cloneable, Serializable
{
   public HashSet() {...}
   public HashSet(int initialCapacity) {...}
   public HashSet(Collection<? extends E> c) {...}
   public HashSet(int initialCapacity, float loadFactor) {...}

   public Object clone() {...}

   // implementations of Set interface methods go here
}
```

Figure 13.6: The `HashSet<E>` class

```
public class LinkedHashSet<E> extends HashSet<E>
   implements Set<E>, Cloneable, Serializable
{
   public LinkedHashSet() {...}
   public LinkedHashSet(int initialCapacity) {...}
   public LinkedHashSet(Collection<? extends E> c) {...}
   public LinkedHashSet(int initialCapacity, float loadFactor) {...}

   public Object clone() {...}

   // implementations of Set interface methods go here
}
```

Figure 13.7: The `LinkedHashSet<E>` class

constructs an empty set with a default initial capacity of 16 elements. The second constructor specifies a given initial capacity.

The third one is called a **conversion constructor** and is very useful. It creates a set of element type `E` from any given collection `c` which may have any element type which extends or implements the type `E`. This constructor can also be used as a copy constructor if `c` has type `E`.

We will not use the fourth constructor. It is used to optimize the hash table implementation.

13.6.2 `LinkedHashSet<E>` implementation of `Set<E>`

A summary of the `LinkedHashSet<E>` implementation is given in Figure 13.7. The constructors are identical to the ones in `HashSet<E>`.

13.6.3 `TreeSet<E>` implementation of `SortedSet<E>` and `Set<E>`

A summary of the `TreeSet<E>` implementation is given in Figure 13.8. Note that `TreeSet<E>`

```
public class TreeSet<E> extends AbstractSet<E>
   implements SortedSet<E>, Cloneable, Serializable
{
   public TreeSet() {...}
   public TreeSet(Collection<? extends E> c) {...}
   public TreeSet(Comparator<? super E> c) {...}
   public TreeSet(SortedSet<E> s){...}

   public Object clone() {...}

   // implementations of SortedSet interface methods go here
   // SortedSet extends the Set interface
}
```

Figure 13.8: The TreeSet<E> class

implements the SortedSet<E> interface which extends the Set<E> interface so TreeSet<E> also extends Set<E>. We will not need the extra methods provided by the SortedSet<E> interface.

There are four constructors. The first provides an empty set. As elements are added they will sorted according to the natural order of the elements of type E (E must implement Comparable<E>).

The second is a **conversion constructor** similar to the one in HashSet<E>. It creates a sorted set of element type E from any given collection c which may have type E or any element type which extends or implements the type E.

The third constructor provides a Comparator argument which has type E or any type that is a super type of E. It's purpose is to define the total order to be used by TreeSet<E>. If this constructor is not used then the natural ordering defined by the element type E is used. In this case the type E must implement the Comparable<E> interface.

The last constructor is a copy constructor which makes a copy of any sorted set.

13.6.4 Simple set examples

■ EXAMPLE 13.11 **(Removing duplicates from a collection)** Suppose we have a collection c of strings and we want to obtain a new collection that is c with duplicates removed. The following statement does this

```
Set<String> noDups = new HashSet<String>(c);
```

using the conversion constructor. ■

■ EXAMPLE 13.12 **(Random sets of elements)** The following statements create a set of 10 integers generated randomly in the range 1 to n where $n > 9$.

```
Random random = new Random();
Set<Integer> randomSet = new TreeSet<Integer>();
while (randomSet.size() < 10)
{
```

```
      randomSet.add(random.nextInt(n) + 1);
   }
```

Here we simply try to add elements until the set has size 10. It is important to have $n > 9$ or the loop will be infinite since there are no sets of size 10 containing only numbers in the range $1 \le k \le 9$. ■

■ EXAMPLE 13.13 **(Using HashSet to compute set union)** The statements

```
Set<String> s1 = new HashSet<String>();
s1.add("one"); s1.add("two"); s1.add("three");
Set<String> s2 = new HashSet<String>();
s2.add("four"); s2.add("five"); s2.add("six");
```

define two sets of strings and the statements

```
Set<String> union = new HashSet<String>(s1);
union.addAll(s2);
System.out.println(union);
```

create a copy of `s1` and use `addAll` to compute the union of the two sets without modifying either `s1` or `s2`. The result displayed is

```
[one, two, five, four, three, six]
```

The output shows there is no specific order.

If you replace `HashSet` by `LinkedHashSet` everywhere the result displayed is

```
[one, two, three, four, five, six]
```

Now the order is the same as the order in which the strings were added to the set.

If you replace `HashSet` by `TreeSet` everywhere the result displayed is

```
[five, four, one, six, three, two]
```

Now the elements appear in alphabetical order. ■

■ EXAMPLE 13.14 **(Using an iterator as a filter)** The statements

```
Set<Integer> s = new HashSet<Integer>();
s.add(1); s.add(2); s.add(3); s.add(3); s.add(4); // [1,2,3,4]
Iterator<Integer> iter = s.iterator(); // ask s for an iterator
while (iter.hasNext())
{
   int k = iter.next();
   if (k % 2 == 0)
      iter.remove();
}
System.out.println(s);
```

use an iterator to remove all the even integers from the set s of integers. The print statement displays [1,3]. ■

■ EXAMPLE 13.15 **(Use an iterator as a filter)** The following statements

```
Set<Integer> s = new HashSet<Integer>();
s.add(1); s.add(2); s.add(3); s.add(3); s.add(4); // [1,2,3,4]
Iterator<Integer> iter = s.iterator(); // ask s for an iterator
Set<Integer> evenSet = new HashSet<Integer>();
Set<Integer> oddSet = new HashSet<Integer>();
while (iter.hasNext())
{
   int k = iter.next();
   if (k % 2 == 0)
      evenSet.add(k);
   else
      oddSet.add(k);
}
System.out.println(evenSet);
System.out.println(oddSet);
```

use an iterator to create two new sets from s, one containing the even integers in s and the other containing the odd integers in s. The print statements display [2,4] and [1,3] ■

13.6.5 Removing duplicates from a list of words

Using sets we can easily write a program that removes duplicate words in a list of words. Simply read the words and add them to a set. Any duplicates will not be added.

Class RemoveDuplicateWords

book-project/chapter13/sets

```
package chapter13.sets;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Scanner;
import java.util.Set;

/**
 * Remove duplicate words from a file of words.
 */
public class RemoveDuplicateWords
{
   public void doTest() throws FileNotFoundException
   {
      Scanner input = new Scanner(new File("files/words.txt"));
```

```
      Set<String> uniqueSet = new HashSet<String>();
      Iterator<String> iter = input;
      while(iter.hasNext())
      {
         uniqueSet.add(iter.next());
      }
      input.close();
      System.out.println(uniqueSet.size() + " unique words found:");
      System.out.println(uniqueSet);
   }

   public static void main(String[] args) throws FileNotFoundException
   {
      new RemoveDuplicateWords().doTest();
   }
}
```

Here we use the fact that the `Scanner` class implements the `Iterator<String>` interface. As each word is read an attempt is made to add it to the set. You can try this program using a file such as

```
all all
words words
are are duplicated duplicated
```

The output is

```
4 unique words found:
[words, all, duplicated, are]
```

You may get a different order since we are using a `HashSet`. For output in alphabetic order use `TreeSet`. For a related problem see Exercise 13.10.

13.7 `List<E>` and `ListIterator<E>` interfaces

A list is a collection of elements arranged in some linear order. It has a first element, a second element and so on. According to Figure 13.1 the `List<E>` interface extends `Collection<E>` so you can think of a list as an ordered collection of elements. The `List<E>` interface is summarized in Figure 13.9. As for the `Collection<E>` interface the operations that can modify a list are indicated as optional so an implementation for immutable lists would not implement these operations.

For traversing lists the `Iterator<E>` interface has been extended to provide a two way iterator called `ListIterator<E>` summarized in Figure 13.10.

13.7.1 `List<E>` interface

The methods from the `Collection<E>` class have basically the same meaning in the `List<E>` interface except that the `add` and `addAll` methods now specify that these operations append the elements to the end of the list and the `remove` method specifically removes the first occurrence of the element.

```
public interface List<E> extends Collection<E>
{
   // The Collection<E> interface methods can go here

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

   // Search Operations
   int indexOf(Object obj);
   int lastIndexOf(Object obj);

   // List Iterators
   ListIterator<E> listIterator();
   ListIterator<E> listIterator(int index);

   // View
   List<E> subList(int fromIndex, int toIndex);
}
```

Figure 13.9: The `List<E>` interface

```
public interface ListIterator<E> extends Iterator<E>
{
   // Query Operations
   boolean hasNext();
   E next();
   boolean hasPrevious();
   E previous();

   int nextIndex();
   int previousIndex();

   // Modification Operations
   void remove(); // optional
   void set(E element); // optional
   void add(E element); // optional
}
```

Figure 13.10: The `ListIterator<E>` interface

We now summarize the extra methods introduced by the `List<E>` interface of Figure 13.9. The additional methods fall into four categories: (1) positional access operations that locate list elements using an index, (2) search operations that find a list element given its index, (3) list iterators that begin at the start of a list or at some other position, and (4) a view operation that returns a sublist.

- **`E get(int index)`**

  Return the element in **`this`** list at position given by **`index`**. If **`index < 0`** or **`index >= size()`** an index out of bounds exception is thrown.

- **`E set(int index, E element)`**

  Replace the element at position **`index`** by the given element. The element being replaced is returned. If **`index < 0`** or **`index >= size()`** an `IndexOutOfBoundsException` is thrown. This is an optional operation.

- **`void add(int index, E element)`**

  Add a new element to **`this`** list at position **`index`**. The elements originally beginning at position **`index`** are moved up to higher indices to accommodate the new element. If **`index < 0`** or **`index > size()`** an index out of bounds exception is thrown. Note that **`index = size()`** is allowed here, corresponding to adding after the last element. This is an optional operation.

- **`boolean addAll(int index, Collection<? extends E> c)`**

  Add all the elements in the given collection **`c`** to **`this`** list beginning at the given position **`index`**. The elements originally beginning at position **`index`** are moved up to higher indices to accommodate the new elements. The restrictions on **`index`** are the same as for the **`add`** method. This is an optional operation.

- **`int indexOf(Object obj)`**

  Return the index of the first occurrence of the given object **`obj`** in **`this`** list. If **`obj`** was not found then -1 is returned.

- **`int lastIndexOf(Object obj)`**

  Return the index of the last occurrence of the given object **`obj`** in **`this`** list. If **`obj`** was not found then -1 is returned.

- **`ListIterator<E> listIterator()`**
  `ListIterator<E> listIterator(int index)`

  Returns a **`ListIterator<E>`** object. For the no-arg version the iterator will start at the beginning of **`this`** list. The second version will start at position **`index`** in **`this`** list. The restrictions on **`index`** are the same as for **`get`**.

- **`List<E> subList(int fromIndex, int toIndex)`**

  Returns a sublist of `this` list beginning and ending at the given indices. If the indices are not in range an `IndexOutOfBoundsException` is thrown.

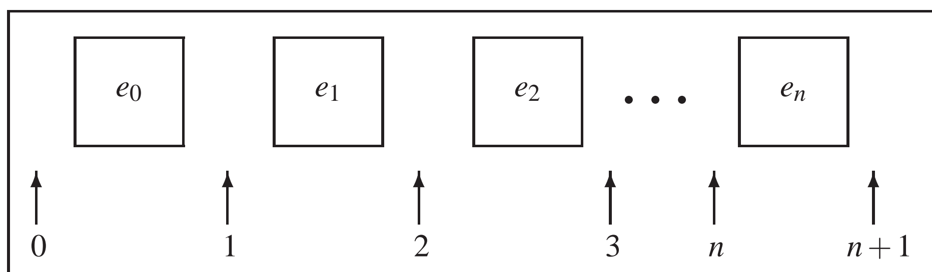

Figure 13.11: Indices for the list $[e_0, e_1, e_2, \ldots, e_n]$ lie between elements.

13.7.2 `ListIterator<E>` interface

As shown in Figure 13.10 the `ListIterator<E>` interface extends `Iterator<E>` so that the list can be traversed in either direction. The `Iterator<E>` part provides iteration in the forward direction using `hasNext()` and `next()` and the new methods provide iteration in the backward direction using `hasPrevious()` and `previous()`.

During iteration the `add`, `remove`, and `set` methods are available. They operate on the current element of the list (last element returned by `next()` or `previous()`). For `add` the element is inserted immediately before the next element that would be returned by `next()`, if any, and after the next element that would be returned by `previous()`.

When using a list iterator it is helpful to think of list indices as lying between the list elements as shown in Figure 13.11. Thus, a call to `next()` returns the element to the right of the index and advances to the next higher index. Similarly, a call to `previous()` returns the element to the left of the index and advances to the next lower index.

13.8 `List<E>` implementations and examples

The JCF includes two general purpose implementations of the `List<E>` interface: `ArrayList<E>` and `LinkedList<E>`.

13.8.1 `ArrayList<E>` implementation of `List<E>`

The `ArrayList<E>` class implements a dynamic array ADT and is the best implementation if you need positional access to the list using a 0-based index. Accessing an element given its index is an $O(1)$ operation. Thus this is a random access structure like the built-in `array` class. The `ArrayList<E>` class is summarized in Figure 13.12.

There are three constructors. The no-arg constructor provides a resizable list with initial space for 10 elements and the second constructor provides a resizable list with the specified initial capacity.

The third constructor is a conversion constructor that creates an `ArrayList<E>` from the given collection `c` in the order defined by the collection's iterator.

The dynamic increase in the size of the list occurs automatically as needed. Two methods are

```
public class ArrayList<E> extends AbstractList<E>
   implements List<E>, RandomAccess, Cloneable, Serializable
{
   // Constructors
   public ArrayList() {...}
   public ArrayList(int initialCapacity) {...}
   public ArrayList(Collection<? extends E> c) {...}

   // Implementation of List<E> interface methods go here

   // Extra methods
   public Object clone() {...}
   public void ensureCapacity(int minCapacity) {...}
   public void trimToSize() {...}
}
```

Figure 13.12: The `ArrayList<E>` class

supplied for resizing the list under program control. The `ensureCapacity` method can be used to expand the size to a specified amount if necessary and the `trimToSize` method can be used to downsize the list so that its capacity is the same as its size.

Our `DynamicArray<E>` class (see page 740) is a very simple version of `ArrayList<E>`.

13.8.2 `LinkedList<E>` implementation of `List<E>`

The `LinkedList<E>` implementation uses a linked list data structure (discussed in a data structures course). For random access (using an index) this implementation is inefficient ($O(n)$). If you mostly want to add and remove elements using the list iterator (access relative to the current element) then this implementation is efficient ($O(1)$) whereas the `ArrayList<E>` implementation would be inefficient. The `LinkedList<E>` class is summarized in Figure 13.13.

There are two constructors. The no-arg constructor creates an empty list. There is no need to specify a capacity since one of the properties of a linked list is that it can grow and shrink one element at a time in a very efficient manner.

The second constructor is a conversion constructor that creates a linked list from the given collection `c` in the order defined by the collection's iterator.

13.8.3 Simple list examples

■ EXAMPLE 13.16 **(Converting a collection to a list)** The statement

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

uses the conversion constructor to convert any collection `c` of strings to an `ArrayList` of strings in the order given by the collection's iterator. ■

■ EXAMPLE 13.17 **(Appending to a list)** The statement

```
public class LinkedList<E> extends AbstractSequentialList<E>
   implements List<E>, Queue<E>, Cloneable, Serializable
{
   // Constructors
   public LinkedList() {...}
   public LinkedList(Collection<? extends E> c) {...}

   // Implementation of List<E> interface methods go here
   // Queue<E> related methods go here

   // Extra methods
   public Object clone() {...}
   public void addFirst(E element) {...}
   public void addLast(E element) {...}
   public E getFirst() {...}
   public E getLast() {...}
   public E removeFirst() {...}
   public E removeLast() {...}
}
```

Figure 13.13: The `LinkedList<E>` class

```
list1.addAll(list2);
```

appends `list2` to the end of `list1`.

The statements

```
List<String> list3 = new ArrayList<String>(list1);
list3.addAll(list2);
```

append two lists to create a new list without modifying either `list1` or `list2`. ■

■ EXAMPLE 13.18 **(Swapping (exchanging) two list elements)** Given a list of strings the following statements

```
String temp = list.get(i);  // String temp = list[i];
list.set(i, list.get(j));   // list[i] = list[j];
list.set(j, temp);          // list[j] = temp;
```

use the indexed list operations `get` and `set` to swap the elements at positions `i` and `j`. The comments show the statements that would be used if `list` were an array instead of a list.

The polymorphic static method

```
public static <E> void swap(List<E> list, int i, int j)
{
   E temp = list.get(i);
   list.set(i, list.get(j));
   list.set(j, temp);
```

```
    }
```

can be used to swap two elements of any list. ■

13.8.4 Book inventory example

Here we create a simple book inventory system. Each book is represented as an object from a `Book` class and the books in the store are represented as a list of type `ArrayList<Book>`,

Each book has data fields for a title, author, price, and the number of books in stock. We want to process a list of books and remove books that are not in stock. The books removed can be stored in another reorder list. The `Book` class is given by

Class `Book`

book-project/chapter13/lists

```
package chapter13.lists;
/**
 * Book objects have a title, author, price, quantity in stock.
 * Books can also be ordered by increasing order of title.
 */
public final class Book implements Comparable<Book>
{
   private String title;
   private String author;
   private double price;
   private int inStock;

   /**
    * Construct a book from given data.
    * @param title the title of the book.
    * @param author the author of the book.
    * @param price the retail price of the book.
    * @param inStock the number of books in stock.
    */
   public Book(String title, String author, double price, int inStock)
   {
      this.title = title;
      this.author = author;
      this.price = price;
      this.inStock = inStock;
   }

   /**
    * Return the author of the book.
    * @return the author of the book.
    */
   public String getAuthor()
   {
      return author;
   }
```

```
/**
 * Return the number of books in stock.
 * @return the number of books in stock.
 */
public int getInStock()
{
   return inStock;
}

/**
 * Return the retail price of the book.
 * @return the retail price of the book.
 */
public double getPrice()
{
   return price;
}

/**
 * Return the title of the book.
 * @return the title of the book.
 */
public String getTitle()
{
   return title;
}

/**
 * Return a string representation of a book.
 * @return a string representation of a book.
 */
public String toString()
{
   return "Book[" + title + "," +
      author + "," + price + "," + inStock + "]";
}

/**
 * Compare this book to another book using the title.
 * @param b the book to compare with this book
 * @return negative, zero, positive results
 */
public int compareTo(Book b)
{
   return title.compareTo(b.title);
}

/**
 * Return true if this book has the same title as obj.
 * @param obj the book to compare with this book
 * @return true if this book has same title as obj
```

```
    */
   public boolean equals(Object obj)
   {
      if (obj == null || getClass() != obj.getClass())
         return false;
      return title.equals(((Book) obj).title);
   }

   public int hashCode()
   {
      return title.hashCode();
   }
}
```

We have implemented the `Comparable<Book>` interface that defines the natural order with the `compareTo` method to be alphabetical order by title. An `equals` method has also been provided and the corresponding `hashCode` is obtained using the hash code of the title string. Choosing hash codes is best left to a course on data structures. Here we use the hash code already defined in the `String` class.

The following static method can be used to produce the two lists.

```
public static List<Book> reOrderBooks(List<Book> list)
{
   List<Book> reOrderList = new LinkedList<Book>();
   Iterator<Book> iter = list.iterator();
   while (iter.hasNext())
   {
      Book b = iter.next();
      if (b.getInStock() == 0)
      {
         reOrderList.add(b);
         iter.remove();
      }
   }
   return reOrderList;
}
```

Here `list` is the given list to split. A `reOrderList` is created and the iterator `iter` is used to traverse the given list, removing elements with an in stock value of 0. Each element removed is added to `reOrderList` which is returned by the method. Note that we have used `Iterator<Book>` instead of `ListIterator<Book>` since the extra methods in `ListIterator<Book>` are not used here.

We have used `LinkedList` here instead of `ArrayList` since we access the list only relatively using the iterator's `add` and `remove` methods which are efficient.

Here is a short program that can be used to test the method.

Class `BookList`

book-project/chapter13/lists

```
package chapter13.lists;
import java.util.LinkedList;
import java.util.Iterator;
import java.util.List;

public class BookList
{
   /**
    * Modify original list so it contains only books
    * in stock and create a new list that contains books
    * which are out of stock.
    */
   public void processBookList()
   {
      List<Book> list = new LinkedList<Book>();
      list.add(new Book("Dead Souls", "Ian Rankin", 25.95 ,10));
      list.add(new Book("Stranger House", "Reginald Hill", 29.50 ,0));
      list.add(new Book("Not Safe After Dark", "Peter Robinson", 32.99 ,10));
      list.add(new Book("Original Sin", "P. D. James", 39.95 ,0));
      list.add(new Book("Fleshmarket Close", "Ian Rankin", 25.00 ,0));

      List<Book> reOrderList = reOrderBooks(list);
      System.out.println("Re-order list:");
      displayList(reOrderList);
      System.out.println("List in stock:");
      displayList(list);
   }

   /**
    * Create lists of books in stock and reorder list.
    * @param list the book list
    * @return the list of books to be ordered.
    * The original list now contains only books that are instock.
    */
   public static List<Book> reOrderBooks(List<Book> list)
   {
      List<Book> reOrderList = new LinkedList<Book>();
      Iterator<Book> iter = list.iterator();
      while (iter.hasNext())
      {
         Book b = iter.next();
         if (b.getInStock() == 0)
         {
            reOrderList.add(b);
            iter.remove();
         }
      }
      return reOrderList;
   }

   public static <E> void displayList(List<E> list)
   {
```

```
      for (E element : list)
         System.out.println(element);
   }

   public static void main(String[] args)
   {
      BookList books = new BookList();
      books.processBookList();
   }
}
```

A for-each loop is used to display the books, one per line and the output is

```
Re-order list:
Book[Stranger House,Reginald Hill,29.5,0]
Book[Original Sin,P. D. James,39.95,0]
Book[Fleshmarket Close,Ian Rankin,25.0,0]
List in stock:
Book[Dead Souls,Ian Rankin,25.95,10]
Book[Not Safe After Dark,Peter Robinson,32.99,10]
```

13.8.5 Insertion in a sorted list

An easy way to maintain a list in some sorted order is to start with an empty list and as elements are added to the list put them in the correct position so that the list remains sorted. In this way we avoid sorting altogether.

To develop the algorithm suppose that $[e_0, e_1, \ldots, e_n]$ is a list that is sorted in some order. If we want to add an element e to the list in its proper sorted position then we need to iterate through the list and compare e with each e_k. The iteration continues until we arrive at an element e_k such that $e \le e_k$. Then the proper place for e is before e_k. There are two special cases: (1) list is empty so create a one-element list, (2) we never find that $e \le e_k$ so the element e must be added at the end of the list.

Let us assume that we have a sorted list of integers. Then we can write the following method to do the insertion.

```
public static void
insertInSortedIntegerList(List<Integer> list, Integer newElement)
{
   ListIterator<Integer> iter = list.listIterator();

   if (!iter.hasNext()) // empty list so make a 1-element list
   {
      iter.add(newElement);
      return;
   }

   while(iter.hasNext())
   {
```

```
         int ek = iter.next();
         if (newElement <= ek)
         {
            iter.previous(); // backup
            iter.add(newElement);
            return;
         }
      }
      iter.add(newElement); // add after end of list
   }
```

It is important to note that `previous()` is needed since to find the correct position using `next()` we need to add the element at the position to its left so `previous()` backs up the iterator. If we come out of the while loop then we need to add the new element to the end of the list.

Statements such as the following can be used to test the method:

```
List<Integer> list = new ArrayList<Integer>();
list.add(4); list.add(6); list.add(8);
System.out.println(list);
insertInSortedIntegerList(list,9);
System.out.println(list);
```

The result is the list `[4,6,8,9]`.

We can convert this method to the following polymorphic generic one with type `E`.

```
public static <E extends Comparable<E>>
void insertInSortedList(List<E> list, E newElement)
{
   ListIterator<E> iter = list.listIterator();

   if (!iter.hasNext()) // empty list so make a 1-element list
   {
      iter.add(newElement);
      return;
   }

   while(iter.hasNext())
   {
      E element = iter.next();
      if (newElement.compareTo(element) <= 0)
      {
         iter.previous(); // backup
         iter.add(newElement);
         return;
      }
   }
   iter.add(newElement); // add after end of list
}
```

Here we specify that the generic type must extend or implement the `Comparable<E>` interface. Then instead of using `<=` we use the `compareTo` method of the `Comparable<E>` interface.

This example can also be done using a `LinkedList<E>`, which may be more efficient than an `ArrayList<E>` in this case, since any modifications to the input list are done using only relative access and the list iterator operations are $O(1)$.

Here is a short program that can be used to test the method for lists of type `String` and `Book` both of which implement the `Comparable` interface.

Class `SortedListExample`

book-project/chapter13/lists

```
package chapter13.lists;
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class SortedListExample
{
   public void doTest()
   {
      // Try it on a list of strings

      List<String> strList = new ArrayList<String>();
      strList.add("Fred"); strList.add("Jane"); strList.add("Mike");
      System.out.println(strList);
      insertInSortedList(strList, "Gord");
      System.out.println(strList);
      insertInSortedList(strList,"Carol");
      System.out.println(strList);
      insertInSortedList(strList,"Bob");
      System.out.println(strList);
      insertInSortedList(strList,"Susan");
      System.out.println(strList);

      // Try it on a list of books

      List<Book> list = new ArrayList<Book>();
      insertInSortedList(list, new Book("Dead Souls", "Ian Rankin", 25.95 ,10));
      insertInSortedList(list, new Book("Stranger House", "Reginald Hill", 29.50 ,0));
      insertInSortedList(list,
                  new Book("Not Safe After Dark", "Peter Robinson", 32.99 ,10));
      insertInSortedList(list, new Book("Original Sin", "P. D. James", 39.95 ,0));
      insertInSortedList(list, new Book("Fleshmarket Close", "Ian Rankin", 25.00 ,0));
      displayList(list);
   }

   public static <E extends Comparable<E>>
   void insertInSortedList(List<E> list, E newElement)
   {
      ListIterator<E> iter = list.listIterator();
```

```
      if (!iter.hasNext()) // empty list so make a 1-element list
      {
         iter.add(newElement);
         return;
      }
      // Note: when we know where to insert
      // the new element we have gone one
      // position too far so previous is needed.
      while(iter.hasNext())
      {
         E element = iter.next();
         if (newElement.compareTo(element) <= 0)
         {
            iter.previous(); // backup
            iter.add(newElement);
            return;
         }
      }
      iter.add(newElement); // add after end of list
   }

   public static <E> void displayList(List<E> list)
   {
      for (E element : list)
         System.out.println(element);
   }

   public static void main(String[] args)
   {
      SortedListExample example = new SortedListExample();
      example.doTest();
   }
}
```

The sorted output is

```
[Fred, Jane, Mike]
[Fred, Gord, Jane, Mike]
[Carol, Fred, Gord, Jane, Mike]
[Bob, Carol, Fred, Gord, Jane, Mike]
[Bob, Carol, Fred, Gord, Jane, Mike, Susan]
Book[Dead Souls,Ian Rankin,25.95,10]
Book[Fleshmarket Close,Ian Rankin,25.0,0]
Book[Not Safe After Dark,Peter Robinson,32.99,10]
Book[Original Sin,P. D. James,39.95,0]
Book[Stranger House,Reginald Hill,29.5,0]
```

13.9 Map data type

Maps are one of the most important data types. A map is a function f that associates elements of one set K called the domain of the map to elements of another set V called the range of the map. Each element of the domain is often called a **key** and the corresponding element of the range is often called the **value**.

A map can be denoted by $f : K \to V$ or as a set of **key-value pairs** (k, v) denoted in the finite case by the set

$$f = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}.$$

of n pairs. We can also denote the pair (k, v) by v_k which looks like array notation except the subscripts do not need to be integers.

The keys themselves form the set $K = \{k_1, k_2, \ldots, k_n\}$ since no two keys can be the same. Since two or more keys can be associated with the same value, the values do not form a set, they form a collection.

13.9.1 Name-age example

As a simple example consider a set of names as the domain and the set of ages as the range. Then the following map associates names of people with their age.

$$\text{age} = \{(\text{Jane}, 12), (\text{Fred}, 10), (\text{Mary}, 15), (\text{Bob}, 10)\}.$$

Then, for example, using standard function notation, $\text{age}(\text{Fred}) = 10$ and $\text{age}(\text{Mary}) = 15$. A map can be visualized as a two-column table as shown in Figure 13.14. Here the keys go in the first

Name	Age
Jane	12
Fred	10
Mary	15
Bob	10

Figure 13.14: A two-column representation of the name-age map

column and the corresponding values go in the second column.

13.9.2 Basic map operations

The basic operations on a map are

add Add a new key-value pair to the map (a map should be resizable).

delete Remove a key-value pair given its key.

replace Replace the value in a key-value pair with a new value given its key.

search Search for ("look up") the value associated with a given key.

The most important operation on a map is to be able to efficiently "look up" the value associated with a given key. A naive approach to this would be to use an array data structure to store the key-value pairs and, given a key, use a linear search to find the ordered pair containing this key and hence the value. This searching method would be $O(n)$.

A much better approach is to use a data structure called a hash table that uses a hash code to make lookup much more efficient than linear search. In fact look up is normally an $O(1)$ operation.

13.9.3 Hash tables and codes

We consider a very simple case of a hash table which is the implementation data structure for a map. In our case the keys and values are both integers. Suppose we have an array with indices 0 to 10 as shown in Figure 13.15 that can hold the key-value pairs. Here we assume that each array

v_{132}			v_{102}	v_{15}	v_5	v_{257}		v_{558}		v_{32}
0	1	2	3	4	5	6	7	8	9	10

Figure 13.15: A simple hash table of size 11 using $h(k) = k \bmod 11$. Here v_k is the value associated with key k.

location can hold one key-value pair and the notation v_k indicates that the value associated with key k is v_k and we assume that the values are non-negative integers. There is room for 11 pairs and some of them are shown in the figure. Empty array locations are unused.

For each key we need a function to transform the key into an array index which can then be used to obtain the value associated with this key.

In general the range of values (non-negative integers in this case) is much greater than the size of the array so we cannot just store the pair with key k in the location with index k. To be specific let us assume that each key k satisfies $0 \le k \le 1000$. What we need is a function $h(k)$ called a hash function that produces an integer hash code for each key k. This code can then be converted to an array index i in the range $0 \le i \le 10$ using $i = h(k) \bmod 11$. We consider only the simplest case which is $h(k) = k$ so that the array index of key k is $i = k \bmod 11$.

Suppose we start with an empty array and begin inserting pairs with keys 15, 558, 32, 132, 102, and 5. Then the corresponding array indices are $15 \bmod 11 = 4$, $558 \bmod 11 = 8$, $32 \bmod 11 = 10$, $132 \bmod 11 = 0$, $102 \bmod 11 = 3$, and $5 \bmod 11 = 5$, as shown in Figure 13.15.

No problems are encountered since all the remainders are different. However when we try to insert a pair with key 257 then $257 \bmod 11 = 4$ and location 4 is already occupied by the pair v_{15} having key 15. This is inevitable as we insert new pairs since there are many more keys than array indices. This situation is called a **collision** and we need a **collision resolution policy** to decide where to store the pair. The simplest policy is to find the next highest empty location and store the

pair there. In our example this means that pair v_{257}, which would have gone in the location with index 4, now goes in the location with index 6, as shown in Figure 13.15. In general we would assume that the array indices wrap around with index 0 following index 10. If there is no free location this means that the array is full and would need to be expanded by doubling its size for example.

13.10 The `Map<K,V>` interface

The JCF has a `Map<K,V>` interface that defines the basic operations on maps. This interface is parametrized with two generic types. The type `K` is the key type and the type `V` is the value type. They can be any object type. The methods in the `Map<K,V>` interface are shown in Figure 13.16. An interesting feature of this interface is that it contains an inner interface to represent the entries (pairs) in the map. Detailed descriptions of these operations are given in the Java API documentation which is summarized here.

- **`int size();`**

  Return the number of pairs (entries) currently stored in **`this`** map.

- **`boolean isEmpty();`**

  Return true if **`this`** map is empty (contains no entries).

- **`boolean containsKey(Object key);`**

  Return true if an entry with the given **`key`** is in **`this`** map.

- **`boolean containsValue(Object value);`**

  Return true if an entry with the given **`value`** is in **`this`** map.

- **`V get(Object key);`**

  Return the value associated with the given **`key`**. This is the "look up" operation. A return value of `null` either indicates that there is no entry with this key or there is an entry but its value is `null`.

- **`V put(K key, V value);`**

  Add a new pair (entry) to the map with given **`key`** and **`value`**. If the entry was already in **`this`** map then the old value is replaced by **`value`** and the old value is returned. Otherwise a new entry is added to **`this`** map and `null` is returned. This is an optional operation.

- **`V remove(Object key);`**

  If the entry with the given **`key`** is in **`this`** map then it is removed and its value is returned. Otherwise `null` is returned. This is an optional operation.

- **`void putAll(Map<? extends K, ? extends V> t);`**

  All the entries in the map **`t`** are put into **`this`** map. The types of the map **`t`** can be **`K`** and **`V`** or any types that extend or implement **`K`** and **`V`**. This is an optional operation.

```
public interface Map<K,V>
{
   // Query Operations
   int size();
   boolean isEmpty();
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   V get(Object key);

   // Modification Operations
   V put(K key, V value); // optional
   V remove(Object key); // optional

   // Bulk Operations
   void putAll(Map<? extends K,? extends V> t); // optional
   void clear(); // optional

   interface Entry<K,V>
   {
      K getKey();
      V getValue();
      V setValue(V value); // optional
      boolean equals(Object obj);
      int hashCode();
   }

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

   // Comparison and hashing
   boolean equals(Object obj);
   int hashCode();
}
```

Figure 13.16: `Map` interface

- **`void clear();`**

  Remove all the entries from **`this`** map. The result is the empty map. This is an optional operation.

- **`interface Entry<K,V>`**

  This is an inner interface that defines a map entry (pair). To refer to such an entry use the type `Map.Entry<K,V>`.

  - **`K getKey();`**
    Return the key of **`this`** entry.
  - **`V getValue();`**
    Return the value of **`this`** entry.
  - **`V setValue(V value);`**
    Set a new value for **`this`** entry. This is an optional operation.
  - **`boolean equals(Object obj);`**
    Return true if **`obj`** is equal to **`this`** entry.
  - **`int hashCode()`**
    Return the hash code of **`this`** entry.

- **`Set<K> keySet();`**

  Return the keys in **`this`** map as a set.

- **`Collection<V> values();`**

  Return the values in **`this`** map as a collection.

- **`Set<Map.Entry<K,V>> entrySet();`**

  Return the entries of **`this`** map as a set of elements of type **`Map.Entry<K,V>`**.

- **`boolean equals(Object obj);`**

  Return true if **`obj`** is a map equal to **`this`** map.

- **`int hashCode()`**

  Return the hash code of **`this`** map.

13.11 Map implementations and examples

The JCF has several implementations of the `Map<K,V>` interface. We will consider three of them that are similar to the corresponding ones for sets: `HashMap<K,V>`, `LinkedHashMap<K,V>`, and `TreeMap<K,V>`.

```
public class HashMap<K,V> extends AbstractMap<K,V>
   implements Map<K,V>, Cloneable, Serializable
{
   public HashMap() {...}
   public HashMap(int initialCapacity) {...}
   public HashMap(Map<? extends K,? extends V> m) {...}
   public HashMap(int initialCapacity, float loadFactor) {...}

   public Object clone() {...}

   // Implementations of Map interface methods go here
}
```

Figure 13.17: The HashMap<K,V> class

```
public class LinkedHashMap<E> extends HashMap<K,V>
   implements Map<K,V>, Cloneable, Serializable
{
   public LinkedHashMap() {...}
   public LinkedHashMap(int initialCapacity) {...}
   public LinkedHashMap(int initialCapacity, float loadFactor) {...}
   public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder) {...}
   public LinkedHashMap(Map<? extends K,? extends V> m) {...}

   public Object clone() {...}

   // Implementations of Map interface methods go here
   // Other methods go here
}
```

Figure 13.18: The LinkedHashMap<K,V> class

13.11.1 HashMap<K,V> implementation of Map<K,V>

The HashMap<K,V> implementation is the fastest but it does not maintain any order to the entries in the map. A class summary is shown in Figure 13.17.

There are four constructors. The first constructor with no arguments constructs an empty map with a default initial capacity of 16 elements. The second constructor specifies a given initial capacity. The third one is called a conversion constructor and can be used as a copy constructor. We will not use the fourth constructor. It is used to optimize the hash table implementation.

13.11.2 LinkedHashMap<K,V> implementation of Map<K,V>

The LinkedHashMap<K,V> implementation maintains the order in which keys are added to the map. A class summary is shown in Figure 13.18.

```
public class TreeMap<K,V> extends AbstractMap<K,V>
   implements SortedMap<K,V>, Cloneable, Serializable
{
   public TreeMap() {...}
   public TreeMap(Comparator<? super K> c) {...}
   public TreeMap(Map<? extends K,? extends V> m) {...}
   public TreeMap(SortedMap<K,? extends V> m) {...}

   public Object clone() {...}

   // implementations of SortedMap interface go here
   // SortedMap extends the Map interface
}
```

Figure 13.19: The `TreeMap<K,V>` class

13.11.3 `TreeMap<K,V>` implementation of `Map<K,V>`

The `TreeMap<K,V>` implementation provides a sorted order based on the natural ordering of the keys as given by the `Comparable<K>` interface implemented by `K`. A class summary is shown in Figure 13.19. The `SortedMap<K,V>` interface extends the `Map<K,V>` interface to provide extra methods related to the sort order (See Java API documentation).

13.11.4 Simple map examples

Here we give some simple examples to illustrate map operations using the name-age example.

■ EXAMPLE 13.19 **(Constructing a name-age map)** The statements

```
Map<String,Integer> age = new HashMap<String,Integer>();
age.put("Jane", 12);
age.put("Fred", 10);
age.put("Mary", 15);
age.put("Bob", 10);
System.out.println(age);
```

create the name-age map shown in Figure 13.15 using autoboxing from `int` to `Integer`. The no-arg constructor uses a default size of 16 entries for the map. The output is

```
{Bob=10, Jane=12, Fred=10, Mary=15}
```

and shows that the insertion order is not preserved by the `HashMap` implementation. If you change the implementation to `LinkedHashMap` then the output is

```
{Jane=12, Fred=10, Mary=15, Bob=10}
```

which is in the order of insertion into the map. Finally, if you change the implementation to `TreeMap` then the output is

```
{Bob=10, Fred=10, Jane=12, Mary=15}
```

which is sorted in increasing order of the names (keys). ■

■ EXAMPLE 13.20 **(Finding the age of a given person)** The statements

```
String name = "Mary";
int a = age.get(name);
System.out.println("Age of " + name + " is " + a);
```

return the age of Mary. ■

■ EXAMPLE 13.21 **(Using get if name is not in the map)** The statements

```
String name = "Gord";
int a = age.get(name);
```

throw a `NullPointerException`. Since Gord is not in the map `get` returns `null` which cannot be unboxed to an `int` so the exception is thrown. This only happens with the primitive types. Without the auto unboxing the statements

```
String name = "Gord";
Integer a = age.get(name);
System.out.println("Age of " + name + " is " + a);
```

return a `null` value for `a` and no exception is thrown. ■

■ EXAMPLE 13.22 **(Checking if a map contains a key)** The statements

```
String name = "Jill";
if (age.containsKey(name))
   System.out.println(name + " was found");
else
   System.out.println(name + " was not found");
```

show that Jill was not found in the map. ■

■ EXAMPLE 13.23 **(Update a value given its key)** The statements

```
String name = "Fred";
age.put(name, 15);
System.out.println("New age of " + name + " is " + age.get(name));
```

update the age of Fred from 10 to 15 and display it. The statements

```
String name = "Fred";
int currentAge = age.get(name);
age.put(name, currentAge + 1);
System.out.println("New age of " + name + " is " + age.get(name));
```

add 1 year to Fred's age and display it. ■

■ EXAMPLE 13.24 **(Deleting an entry given its key)** The statements

```
String name = "Fred";
if (age.containsKey(name))
   age.remove(name);
System.out.println(age);
```

delete Fred from the map and display the resulting map

```
{Bob=10, Jane=12, Mary=15}
```

which shows that Fred is no longer an entry in the map ■

■ EXAMPLE 13.25 **(Iterating over the keys of a map)** To get an iterator over the keys in a map we first get the keys as a set and then ask this set for an iterator. The statements

```
Set<String> keys = age.keySet();
Iterator<String> iter = keys.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   int a = age.get(name);
   System.out.println(name + " -> " + a);
}
```

use the iterator to display the name-age pairs using an "arrow" notation, one per line. ■

■ EXAMPLE 13.26 **(Iterating over the keys using a for-each loop)** The statements

```
for (String name : age.keySet())
{
   System.out.println(name + " -> " + age.get(name));
}
```

use the for-each loop to display the name-age pairs using an "arrow" notation, one per line. ■

■ EXAMPLE 13.27 **(Use the for-each loop to compute average age)** The statements

```
Set<String> keys = age.keySet();
double sum = 0.0;
for (String name : keys)
{
   sum += age.get(name);
}
System.out.println("Average age is " + sum / keys.size());
```

compute the average age. The `size` method is used to find the number of keys in the map. ■

■ EXAMPLE 13.28 **(Use an iterator and the `Map.Entry` interface)** The statements

```
Set<Map.Entry<String,Integer>> entries = age.entrySet();
Iterator<Map.Entry<String,Integer>> iter = entries.iterator();
while (iter.hasNext())
{
   Map.Entry<String,Integer> entry = iter.next();
   System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

iterate over the map entries. First we get the `entries` set of type `Map.Entry<String,Integer>` using the inner interface of the `Map<String,Integer>` interface. Then we ask it for an iterator over the entries. Each entry has `getKey()` and `getValue()` methods. The loop displays the entries using arrow notation.

The for-each loop

```
for (Map.Entry<String,Integer> entry : age.entrySet())
{
   System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

can be used as long as the mutable iterator operations are not required. ■

■ EXAMPLE 13.29 **(Adding 1 year to all the ages)** The statements

```
Set<Map.Entry<String,Integer>> entries = age.entrySet();
Iterator<Map.Entry<String,Integer>> iter = entries.iterator();
while (iter.hasNext())
{
   Map.Entry<String,Integer> entry = iter.next();
   entry.setValue(entry.getValue() + 1);
}
System.out.println(entries);
```

use the `entrySet()` iterator to add 1 to all the ages. ■

13.11.5 Hours worked example

As a useful example of a map suppose we have a file called `hours.txt` whose lines contain a person's name and the number of hours they have worked. An example might be

```
Fred:10
Gord:20
Fred:30
Mary:15
Gord:13
Mary:4
Mary:6
```

There can be more than one entry per person and we want to display the total hours worked by each person in the format

```
Fred -> 40.0
Mary -> 25.0
Gord -> 33.0
```

indicating that Fred has worked 40 hours (10 + 30), Gord has worked 33 hours (20 + 13), and Mary has worked 25 hours (15 + 4 + 6).

We can produce this list by reading the file into a map with the names as keys and the hours worked as the values. Each time we read a line we check if the name is already in the map. If it is not we create a new entry, and if it is already in the map we update the number of hours by adding the new value.

Before reading the file we create the following map:

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

If you want the names to be ordered alphabetically then replace `HashMap` by `TreeMap`.

Then if `name` and `hours` are the values read from the file the map is updated using the statements

```
if (map.containsKey(name)) // update hours worked
{
   double currentHours = map.get(name);
   map.put(name, currentHours + hours);
}
else // new entry
{
   map.put(name, hours);
}
```

To read the lines of the file we can use the `split` method in the `String` class, so if `line` is a line read from the file then

```
String[] s = line.split(":");
```

will read the name and hour values as strings into `s[0]` and `s[1]`, using colon as the delimiter. Here is the complete program.

Class `HoursWorked`

book-project/chapter13/maps

```
package chapter13.maps;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
```

```
/**
 * A map example: file contains names and hours worked in the format
 * name:hours
 * A person may appear several times in the file and we want to
 * determine the total hours each person has worked.
 *
 * We read this file a line at a time and separate the name and hours.
 * The name is used as the key in a hash table and the hours is the
 * value if the key is new else the hours are updated. The result
 * is a map containing the total hours worked for each person.
 *
 * If a TreeMap is used instead of a HashMap the names will be
 * ordered in increasing alphabetic order.
 */
public class HoursWorked
{
   private static final File IN_FILE = new File("files/hours.txt");

   public void processFile() throws IOException
   {
      Map<String,Double> map =
         new HashMap<String,Double>();
      BufferedReader in =
         new BufferedReader(new FileReader(IN_FILE));
      String line;

      while ( (line = in.readLine()) != null)
      {
         // Each line of the file contains a name and a number
         // of hours worked separated by a colon which can be
         // preceded by zero or more spaces.

         String[] s = line.split(":");
         String name = s[0].trim();
         double hours = Double.parseDouble(s[1].trim());

         // Echo for checking

         System.out.println(name + ":" + hours);

         // put entries in map and update hours

         if (map.containsKey(name)) // update hours worked
         {
            double currentHours = map.get(name);
            map.put(name, currentHours + hours);
         }
         else // new entry
         {
            map.put(name, hours);
         }
```

```
      }
      in.close();

      // Display the map, one entry per line

      System.out.println("Map is");
      for (String name : map.keySet())
      {
         double hours = map.get(name);
         System.out.println(name + " -> " + hours);
      }
   }

   public static void main(String[] args) throws IOException
   {
      HoursWorked tester = new HoursWorked();
      tester.processFile();
   }
}
```

13.11.6 Favorites map with maps as values

We now do an example of a map whose values are also maps. This example is extended in the end of chapter exercises.

In our case we want a map structure that can record the favorite song, food, golfer, etc, associated with each person. Thus, the key-value pairs of the primary map are names and references to favorite maps. The key-value pairs of each favorite map are the category names, such as food, song and golfer, and the values are the preferences.

Using set theory notation an example of such a map of maps is

$$\begin{aligned}
\text{favorites} &= \{(\text{Bob}, f_1), (\text{Fred}, f_2), (\text{Gord}, f_3)\} \\
f_1 &= \{(\text{food}, \text{salad}), (\text{golfer}, \text{Vijay Singh}), (\text{song}, \text{White Wedding})\} \\
f_2 &= \{(\text{food}, \text{steak}), (\text{golfer}, \text{Tiger Woods}), (\text{song}, \text{Satisfaction})\} \\
f_3 &= \{(\text{food}, \text{spaghetti}), (\text{golfer}, \text{Phil Mickelson}), (\text{song}, \text{Money})\}
\end{aligned}$$

This example is also shown using tables in Figure 13.20. It is easy to construct these maps in Java. The favorites map is given by

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

which is a map from strings to maps from strings to strings. Now the favorite maps are given by

```
Map<String,String> f1 = new HashMap<String,String>();
f1.put("golfer", "Vijay Singh");
f1.put("song", "White Wedding");
f1.put("food", "salad");

Map<String,String> f2 = new HashMap<String,String>();
```

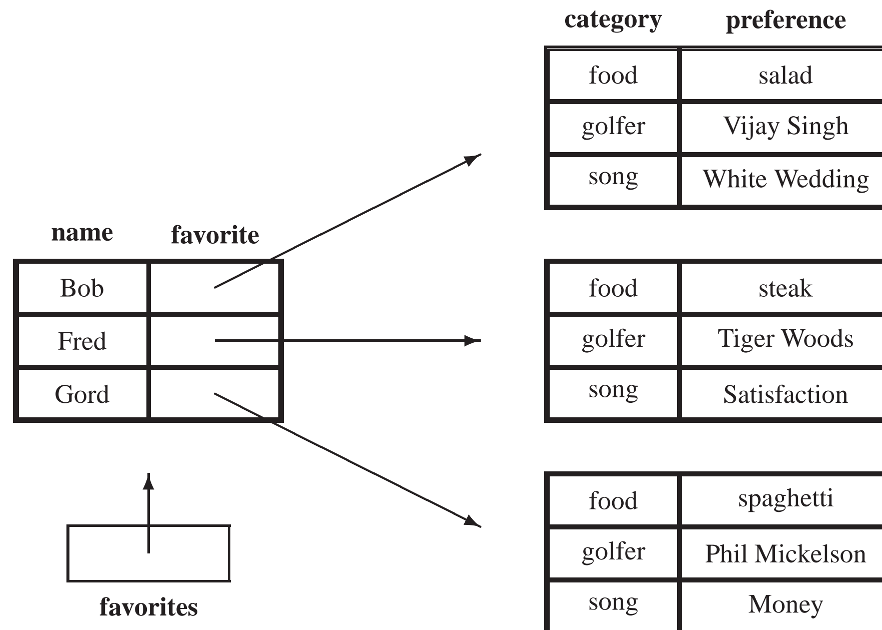

Figure 13.20: A map of maps. The keys of the first map are names. The values are favorite maps whose keys are the categories and values are the preferences.

```
f2.put("golfer", "Tiger Woods");
f2.put("song", "Satisfaction");
f2.put("food", "steak");

Map<String,String> f3 = new HashMap<String,String>();
f3.put("golfer", "Phil Mickelson");
f3.put("song", "Money");
f3.put("food", "spaghetti");
```

Finally we associate these maps as values of the favorites map:

```
favorites.put("Bob", f1);
favorites.put("Fred", f2);
favorites.put("Gord", f3);
```

It is easy to perform operations on this map. For example, to display Fred's favorite map use

```
System.out.println(favorites.get("Fred"));
```

To display Bob's favorite golfer use

```
System.out.println(favorites.get("Fred").get("golfer"));
```

To change Fred's favorite food to chicken use

```
favorites.get("Fred").put("food", "chicken");
```

■ EXAMPLE 13.30 **(For-each loop for favorites map)** The statements

```
for (String name : favorites.keySet())
{
   System.out.println(name);
   System.out.println(favorites.get(name));
}
```

produce the output

```
Bob
{golfer=Vijay Singh, food=salad, song=White Wedding}
Fred
{golfer=Tiger Woods, food=steak, song=Satisfaction}
Gord
{golfer=Phil Mickelson, food=spaghetti, song=Money}
```

which show the favorite maps one per line. To obtain an alphabetical order replace the `HashMap` implementation by `TreeMap`. ■

■ EXAMPLE 13.31 **(Nested for-each loops for favorites map)** The statements

```
for (String name : favorites.keySet())
{
   System.out.println("favorites for " + name + ":");
   Map<String,String> favorite = favorites.get(name);
   for (String  category : favorite.keySet())
   {
      String preference = favorite.get(category);
      System.out.println("   " + category + ": " + preference);
   }
}
```

produce the display

```
favorites for Bob:
   food: salad
   golfer: Vijay Singh
   song: White Wedding
favorites for Fred:
   food: steak
   golfer: Tiger Woods
   song: Satisfaction
favorites for Gord:
   food: spaghetti
   golfer: Phil Mickelson
   song: Money
```

using nested for-each loops to iterate over the maps. The outer loop iterates over each person and the inner loop iterates over all categories in each favorite map. ■

13.12 Recursion examples using maps

Consider a sequence $[s_m, s_{m+1}, s_{m+2}, \ldots, s_n, s_{n+1}, \ldots]$ with starting index m which is often taken to be 0. Such sequences are often defined by recurrence relations of the form $s_n = f(s_{n-1})$, which is a first order recurrence relation since the calculation of s_n depends on the previous term in the sequence, or of the form $s_n = f(s_{n-1}, s_{n-2})$, which is a second-order recurrence relation since the calculation of s_n depends on the previous two terms of the sequence. As a simple example, the recurrence relation $s_n = ns_{n-1}$ with $s_0 = 1$ can be solved to get $s_n = n!$.

Here we consider two recurrence relations, the Fibonacci sequence and the Q-sequence.

13.12.1 The Fibonacci sequence

An important second-order sequence is the Fibonacci sequence defined recursively by

$$F_n = F_{n-1} + F_{n-2}, \text{ where } F_0 = F_1 = 1.$$

There is a closed form expression for the general term F_n but it is not useful for the calculation of terms in the sequence. An efficient non-recursive method is easily written to calculate the terms in the sequence and the following recursive method can also be used

```
public long fib(int n)
{
   if (n == 0 || n == 1)
      return 1L;
   else
      return fib(n-1) + fib(n-2);
}
```

This method is very inefficient because each term is calculated many times. For example, in the calculation of f_{30} the term f_{10} is calculated recursively $10,946$ times.

We can avoid this duplication by a technique called memoization. In our case this means that we can use a map to remember the terms as they are calculated. When we calculate a term for the first time we store it in a map of type `Map<Integer,Long>`. Then whenever this term is needed again we simply look up its value in the map. Here is a class that calculates Fibonacci numbers using a map:

Class `Fibonacci`

book-project/chapter13/maps

```
package chapter13.maps;
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;

public class Fibonacci
{
   Map<Integer,Long> m;
```

```
   public void calculate()
   {
      // Create map and initialize it
      // for fib(0)= 1 and fib(1)= 1

      m = new HashMap<Integer,Long>();
      m.put(0,1L);
      m.put(1,1L);

      Scanner input = new Scanner(System.in);
      System.out.println("Enter n");
      int n = input.nextInt();

      long startTime = System.nanoTime();
      System.out.println(fib(n));
      long time = System.nanoTime() - startTime;

      double seconds = (double) time * 1e-9;
      System.out.println(seconds);
   }

   public long fib(int n)
   {
      if (! m.containsKey(n))
         m.put(n, fib(n-1) + fib(n-2));
      return m.get(n);
   }

   public static void main(String[] args)
   {
      new Fibonacci().calculate();
   }
}
```

Note that before calling `fib` we construct the map and initialize it by putting the entries for $F_0 = 1$ and $F_1 = 1$ into it.

The `fib` method first checks to see if the term F_n is in the map. If it isn't the recursive formula is used to calculate it and put it in the map, otherwise it is looked up in the map and returned.

We have included statements that determine the time in seconds taken to compute a Fibonacci number. A similar class could be written for the recursive version without using a map. Of course the results depend on the particular computer. In one test the calculation of F_{46} took 53.8 seconds without using a map and 3.43×10^{-4} seconds using a map.

13.12.2 The Q-sequence

As another more complicated example which doesn't have a simple non-recursive algorithm consider the sequence

$$Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), \text{ where } Q(1) = 1, Q(2) = 1$$

where we use the more readable function notation $Q(n) = Q_n$. The following recursive method can be used to compute the terms in the sequence.

```
public int q(int n)
{
   if (n <= 2)
      return 1;
   else
      return q(n - q(n-1)) + q(n - q(n-2));
   }
```

The following class uses a map to calculate the terms:

Class `QSequence`

book-project/chapter13/maps

```
package chapter13.maps;
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;

public class QSequence
{
   Map<Integer,Integer> m;

   public void calculate()
   {
      // Create map and initialize it
      // for q(1) = 1 and q(2) = 1

      m = new HashMap<Integer,Integer>();
      m.put(1,1);
      m.put(2,1);

      Scanner input = new Scanner(System.in);
      System.out.println("Enter n");
      int n = input.nextInt();
      long startTime = System.nanoTime();
      System.out.println(q(n));
      long time = System.nanoTime() - startTime;
      double seconds = (double) time * 1e-9;
      System.out.println(seconds);
   }

   public int q(int n)
   {
      if (! m.containsKey(n))
         m.put(n, q(n - q(n-1)) + q(n - q(n-2)));
      return m.get(n);
   }
```

```
    public static void main(String[] args)
    {
        new QSequence().calculate();
    }
}
```

In one test the calculation of $Q(45)$ took 75.8 seconds without using a map and 4.35×10^{-4} seconds using a map.

13.13 Collections utility class

The Collections class is like the Math class: it is a set of useful static methods such as sorting and searching for operating on sets, lists, and maps in the JCF. There are 50 methods in this class and we summarize only a few. For a complete description see the Java API documentation.

- **static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key))**

  Search **list** of type **T** for the given **key**. The list must be in the order specified by the Comparable interface implemented by the list. Returns the zero-based index where **key** was found or (-index - 1) where index is the location where **key** could be inserted.

- **static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)**

  Like the above version of **binarySearch** except using the specified implementation of **Comparator** to define the order. (**list** does not need to implement Comparable in this version).

- **static <T extends Comparable<? super T>> void sort(List<T> list)**

  Sort the given **list** into increasing order using the implementation of the Comparable interface provided by **list**.

- **static <T> void sort(List<T> list, Comparator<? super T> c)**

  Like the above version of **sort** except using the specified implementation of **Comparator** to define the order. (**list** does not need to implement Comparable in this version).

There is also an Arrays class in java.util that provides a similar set of static methods that operate on arrays instead of collections.

13.13.1 Book list sorting example

In this example we consider two ways to use the sort method in the Collections class to sort a list of Book objects (see page 760).

The Book class implements Comparable<Book> which defines the natural order to be increasing alphabetical order by book title. This means that we can sort a book list in this order simply by using

```
Collections.sort(list);
```

where list is a list of books.

If we want to use an order other than the natural order it is necessary to write a class that implements the Comparator<Book> interface. For example, if we want to sort in increasing alphabetic order by author then following class can be used

Class BookComparator

book-project/chapter13/lists

```
package chapter13.lists;
import java.util.Comparator;

public class BookComparator implements Comparator<Book>
{
   /**
    * Compare this book to another book using the author.
    * @param b1 the first book
    * @param b2 the second book
    * @return negative, zero, positive results
    */
   public int compare(Book b1, Book b2)
   {
      return b1.getAuthor().compareTo(b2.getAuthor());
   }
}
```

Now we can use the statement

```
Collections.sort(list, new BookComparator());
```

to sort by author. Here is a class that illustrates these two sorting methods:

Class SortBookList

book-project/chapter13/lists

```
package chapter13.lists;
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;

public class SortBookList
{
   public void processBookList()
   {
```

```
      // A simple list of books

      List<Book> list = new ArrayList<Book>();
      list.add(new Book("Dead Souls", "Ian Rankin", 25.95 ,10));
      list.add(new Book("Stranger House", "Reginald Hill", 29.50 ,0));
      list.add(new Book("Not Safe After Dark", "Peter Robinson", 32.99 ,10));
      list.add(new Book("Original Sin", "P. D. James", 39.95 ,0));
      list.add(new Book("Fleshmarket Close", "Ian Rankin", 25.00 ,0));

      // Sort using the sort method in the Collections class
      // The order uses titles (Book implements Comparable)

      Collections.sort(list);
      System.out.println("List sorted by title:");
      displayList(list);

      // Now use a Comparator the sorts using the author

      Collections.sort(list, new BookComparator());
      System.out.println("List sorted by author:");
      displayList(list);
   }

   public static <E> void displayList(List<E> list)
   {
      for (E element : list)
         System.out.println(element);
   }

   public static void main(String[] args)
   {
      SortBookList books = new SortBookList();
      books.processBookList();
   }
}
```

The output is

```
List sorted by title:
Book[Dead Souls,Ian Rankin,25.95,10]
Book[Fleshmarket Close,Ian Rankin,25.0,0]
Book[Not Safe After Dark,Peter Robinson,32.99,10]
Book[Original Sin,P. D. James,39.95,0]
Book[Stranger House,Reginald Hill,29.5,0]
List sorted by author:
Book[Dead Souls,Ian Rankin,25.95,10]
Book[Fleshmarket Close,Ian Rankin,25.0,0]
Book[Original Sin,P. D. James,39.95,0]
Book[Not Safe After Dark,Peter Robinson,32.99,10]
Book[Stranger House,Reginald Hill,29.5,0]
```

13.14 Programming exercises

▶ **Exercise 13.1 (A random `remove` method)**
Modify the `Bag<E>` interface on page 729 by adding a random `remove` method with prototype

```
E remove();
```

that removes a random element from this bag and returns it. If the bag is empty then `null` is returned. Write the method implementation (it will be the same for both `FixedBag<E>` and `DynamicBag<E>`). You can use the `Random` class in `java.util` that has a `nextInt` method.

▶ **Exercise 13.2 (An indexed `add` method)**
For the `Array<E>` interface add a method with prototype

```
void add(int k, E element);
```

that adds the given `element` at index `k`. The method should throw `IndexOutOfBoundsException` if `k < 0` or `k > size()`.

The element originally at position `k` and all following elements need to be moved up one place to create a place for the new element. The special case when `k` has the value `size()` corresponds to adding the element at the end of the array.

Write the implementation of this method for the `DynamicArray<E>` implementation of the `Array<E>` interface.

▶ **Exercise 13.3 (An indexed `remove` method)**
For the `Array<E>` interface add a method with prototype

```
E remove(int k);
```

that removes the element at index `k` by shifting all following elements down one place. The element removed is returned.

Write the implementation of this method for the `DynamicArray<E>` implementation of the `Array<E>` interface.

▶ **Exercise 13.4 (An `indexOf` method)**
Modify the `Array<E>` interface on page 738 to include an `indexOf` method with prototype

```
int indexOf(E element);
```

that returns the index of the first occurrence of the given element `E` or -1 if the element is not found. Write the method implementation for `DynamicArray<E>`.

▶ **Exercise 13.5 (Generating random sets of numbers)**
Using the idea in Example 13.12, write a method with prototype

```
Set<Integer> randomSet(int n, int a, int b, Random random);
```

that returns a set of `n` integers randomly chosen in the range `a` to `b` inclusive using the `Random` class in `java.util`.

► Exercise 13.6 (Generating Lotto 649 numbers using sets)
Using Exercise 13.5 write a class called `Lotto649` that generates *n* sets of 6 numbers in the range 1 to 49 and displays them.

► Exercise 13.7 (Generating Lotto 649 numbers without using sets)
Without using the JCF write a class called `Lotto649NoSets` that generates n sets of 6 numbers in the range 1 to 49 and displays them.

► Exercise 13.8 (Password generator using sets)
We want to generate a set of unique passwords. Each password is made from the lower and upper case letters and the digits and has a specified length. Write a class to do this. The input is the number of passwords in the set and the number of characters in each password (same for all passwords in the set). Some sample output is

```
d6rIH hX9Av Ki4SK wDAWx olWhW TVU7Y hGDSw VZecI ga7Sy 0DEij
7aDws T0urW MMjk9 JDAHZ vRb1x lGz3q ibiuE H7nbF CB6zY EGzuX
Dhiou mLtkI Eud22 wNbVo iIhLZ Zc73V taPFL wPJGZ nOy9x DPx9F
leJv3 KhmqQ y23g0 ey3Kr VQvq1
```

corresponding to a set size of 35 with 5 characters in each password. Display 10 passwords per line except possibly for the last line.

► Exercise 13.9 (Printing a collection one element per line)
The standard `toString` method creates a string which, when displayed, is all on one line. Write a static polymorphic method with prototype

```
public static <E> void printCollection(Collection<E> c)
```

that prints the elements one per line.

► Exercise 13.10 (Removing duplicate words)
Write a class similar to `RemoveDuplicateWords` on Page 753 and called `UniqueWords` that creates two sets. The first is a set of unique words as defined in the `RemoveDuplicateWords` class, and the second is a set of duplicate words (words that appeared more than once in the input file). From these sets create a set of words that did not have any duplicates in the input. For example for the input

```
a b c d a b e
```

the output should be

```
3 unique words found:
[c,d,e]
2 duplicate words found:
[a,b]
```

► Exercise 13.11 (Adapter class version of the Bag ADT)
Write an adapter class implementation `ArrayBag<E>` of the `Bag<E>` interface on page 729 that adapts an `ArrayList<E>` object. The adapter class has the following structure

```
import java.util.ArrayList;

/**
 * An adapter class implementation of Bag<E>
 */
public class Bag<E>
{
   // This is an adapter class version of the
   // bag ADT that uses an ArrayList

   private ArrayList<E> bag;

   public Bag() {...}
   public Bag(int initialCapacity) {...}

   /**
    * Copy constructor.
    * @param b the bag to copy
    */
   public Bag(Bag<E> b) {...}

   public int size() {...}
   public boolean isEmpty() {...}

   public boolean add(E element) {...}
   public boolean remove(E element) {...}
   public boolean contains(E element) {...}

   public String toString()
   {
      return "Bag" + bag.toString();
   }
}
```

Here all methods are implemented using the `bag` object instance data field of type `ArrayList<E>`.

► **Exercise 13.12 (Memory tester game)**
Write a class called `MemoryTester` that uses the `DynamicBag<Integer>` class and the algorithm shown in Figure 13.21.

Here is some typical output assuming that there are 5 numbers to guess and the numbers are in the range 1 to 10

```
Bag[9,9,8,5,4]
Enter guesses for the 5 numbers in range 1 to 10
9 9 7 5 3
You have 3 guesses correct
Enter guesses for the 5 numbers in range 1 to 10
```

```
ALGORITHM MemoryGame()
Make a bag that can hold 5 integers.
Generate 5 random integers in the range 1 to 10
    and add them to the bag.
LOOP
    Make a copy of the original bag
    Ask user for 5 guesses of numbers in the bag
        and remove the guesses from the bag copy if possible.
    IF bag copy is now empty THEN
        EXIT LOOP
    END IF
    Determine how many guesses are correct.
    Tell user how many guesses are correct.
END LOOP
Congratulate user on winning the game.
```

Figure 13.21: Memory game algorithm

```
9 9 8 5 5
You have 4 guesses correct
Enter guesses for the 5 numbers in range 1 to 10
8 8 7 4 3
You have 2 guesses correct
Enter guesses for the 5 numbers in range 1 to 10
9 9 8 5 4
Congratulations all guesses are correct
```

Here the first line actually shows the answer so that you can check your class. When it is working you can remove this display.

► Exercise 13.13 (Cities and Countries map)
We start with the following text file `cities.txt`

```
Toronto:Canada
Chicago:USA
Frankfort:Germany
Sudbury:Canada
Venice:Italy
Acapulco:Mexico
Berlin:Germany
Barcelona:Spain
Los Angeles:USA
Vancouver:Canada
Rome:Italy
```

```
Miami:USA
London:UK
Mexico City:Mexico
Madrid:Spain
Florence:Italy
```

that is a list of cities and their countries. In general each country can appear several times.

We want to read this file a line at a time and produce a file `countries.txt` having the form

```
Canada -> [Sudbury, Toronto, Vancouver]
Germany -> [Berlin, Frankfort]
Italy -> [Florence, Rome, Venice]
Mexico -> [Acapulco, Mexico City]
Spain -> [Barcelona, Madrid]
UK -> [London]
USA -> [Chicago, Los Angeles, Miami]
```

The output has the form of a map from `String` to `List<String>`:

```
Map<String,List<String>> map = new TreeMap<String,List<String>>();
```

which will arrange the countries in sorted order.

Write a class called `Cities` to solve this problem. You can use an `ArrayList<String>` for each list. See Section 13.11.5 for a simpler example, To sort the lists for each country, before using `PrintWriter` to write the results to a file, use the static `sort` method in the `Collections` class.

► Exercise 13.14 (Another version of the cities and countries map)
Write a version of the `Cities` class from the previous exercise called `Cities2` that produces the same output but expects its input in the compact form

```
Toronto:Canada, Chicago:USA, Frankfort:Germany, Sudbury:Canada
Venice:Italy, Acapulco:Mexico, Berlin:Germany
Barcelona:Spain, Los Angeles:USA, Vancouver:Canada
Rome:Italy, Miami:USA, London:UK
Mexico City:Mexico, Madrid:Spain, Florence:Italy
```

that permits multiple entries per line in the input file separated by commas (use a nested loop with the outer loop using `split(",")` in the outer loop and `split(":")` in the inner loop.

► Exercise 13.15 (Favorites map using data file)
We want to read a text file `favorites.txt` such as

```
Fred:golfer:Tiger Woods
Bob:food:salad
Fred:food:steak
Bob:song:White Wedding
Gord:golfer:Phil Mickelson
Fred:song:Satisfaction
Bob:golfer:Vijay Singh
Gord:song:Money
Gord:food:spaghetti
```

and produce the display shown in Example 13.31 which also corresponds to the favorites map given in Figure 13.20.

Write a class called `Favorites` that does the processing using the map structure of Section 13.11.6. Also see Example 13.30 and Example 13.31.

► Exercise 13.16 (ArrayList version of an address book)

The purpose of this exercise is to write a GUI version of an address book program that uses an `ArrayList` to hold the address book entries. Each entry is an object from an inner class called `AddressBookEntry`. An entry is really two strings, one called the `key` for the name of the person and another called the `value` representing the address information.

Now we need to write a class called `AddressBook` that manages the address book. This class will be used by the GUI class `AddressBookGUI`. The `AddressBook` class has the following structure which you must complete as indicated by the `TODO` lines.

```
public class AddressBook
{
   private List<AddressBookEntry> list; // the list of address book entries
   private String fileName; // name of file containing the list
   private String fileStatus; // Status or error message or empty

   /**
    * Construct an empty address book with a given initial
    * size. No attempt is made to read an address book from
    * a binary object file so this constructor is mainly
    * used for debugging.
    * @param initialSize the initial address book size
    */
   public AddressBook(int initialSize)
   {
      list = new ArrayList<AddressBookEntry>(initialSize);
      fileName = "";
      fileStatus = "";
   }

   /**
    * Make an address book from the data in a binary object file,
    * if the file exists, else construct a new address book. Errors
    * are recorded as strings that can be retrieved using the
    * fileStatus() method.
    */
   public AddressBook(String inFileName)
   {
      fileName = inFileName;
      read();
   }

   /**
```

```
 * Read the address book from a binary object file. If
 * a binary object file does not exist then create a new
 * address book.
 * Errors are recorded as strings that can be retrieved using the
 * fileError() method.
 */
public void read()
{
   // If no error then the string is empty
   fileStatus = "";

   ObjectInputStream in = null;
   try
   {
      in = new ObjectInputStream(new FileInputStream(fileName));
      list = (List<AddressBookEntry>) in.readObject();
      fileStatus = "Address book file has been loaded";
   }
   catch (FileNotFoundException e)
   {
      // make a new address book if input file not found.
      list = new ArrayList<AddressBookEntry>();
      fileStatus = "New address book list has been created";
   }
   catch (ClassNotFoundException e)
   {
      fileStatus = "Invalid address book file";
   }
   catch (IOException e)
   {
      fileStatus = "Unknown error reading address book file";
   }
   finally // make sure the file was closed
   {
      try
      {
         if (in != null) in.close();
      }
      catch (IOException e)
      {
         fileStatus = "Unknown error closing address book file";
      }
   }
}

/**
```

```
 * Write the address book as a binary object file.
 * Errors are recorded as strings that can be retrieved using the
 * fileError() method.
 */
public void write()
{
   fileStatus = "Address book has been saved in file";
   ObjectOutputStream out = null;
   try
   {
      out = new ObjectOutputStream(new FileOutputStream(fileName));
      out.writeObject(list);
   }
   catch (FileNotFoundException e)
   {
      fileStatus = "Address book file not found";
   }
   catch (IOException e)
   {
      fileStatus = "Unknown error writing address book file";
   }
   finally
   {
      try
      {
         if (out != null) out.close();
      }
      catch (IOException e)
      {
         fileStatus = "Unknown error closing output file";
      }
   }
}

/**
 * Return file error or status string after a file operation.
 * @return the status string.
 */
public String fileStatus()
{
   return fileStatus;
}

/**
 * Return number of entries in address book.
 * @return number of entries in address book
```

```
     */
    public int size()
    {
       return list.size();
    }

    /**
     * Return the value associated with a given key.
     * @param key the key to find
     * @return value of key found else null
     */
    public String get(String key)
    {
       // TODO
    }

    /**
     * Add a new entry to the address book.
     * If the entry already exists then it is an update operation
     * so the value of the entry for this key is updated.
     * @param key the key of entry to add or update
     * @param value new value for the entry
     */
    public void add(String key, String value)
    {
       // TODO
    }

    /**
     * Delete an entry from address book given its key.
     * @param key the key of the entry
     * @return true if entry was deleted
     * else false if entry did not exist.
     */
    public boolean delete(String key)
    {
       // TODO
    }

    /**
     * Return a string representation of this list.
     * @return a string representation of this list.
     */
    public String toString()
    {
       // TODO
```

```
}

// -------- inner class for address book entries ------------

/**
 * An object of this class is an entry in an address book database.
 * Each entry is a key-value pair. The keys and values are strings.
 */
private static class AddressBookEntry implements java.io.Serializable
{
   private static final long serialVersionUID = 1L;
   private String key;
   private String value;

   /**
    * Construct an entry given a key and a value.
    * @param key the key for the entry.
    * @param value the value associated with the key.
    */
   public AddressBookEntry(String key, String value)
   {
      this.key = key;
      this.value = value;
   }

   /**
    * Return the key for this entry.
    * @return Return the key for this entry.
    */
   public String getKey()
   {
      return key;
   }

   /**
    * Return the value associated with this key
    * @return Return the value associated with this key.
    */
   public String getValue()
   {
      return value;
   }

   /**
    * Test this object for equality with obj
    * @param obj the object to test with this object
```

```
            * @return true if the two objects have the same keys else false
            */
           public boolean equals(Object obj)
           {
              if (obj == null) return false;
              if (! getClass().equals(obj.getClass())) return false;
              AddressBookEntry entry = (AddressBookEntry) obj;
              return key.equals(entry.key);
           }

           /**
            * Define a string representation of this object.
            * @return Return the string representation of this object.
            */
           public String toString()
           {
              return "AddressBookEntry[" + key + ", " + value + "]";
           }
        }
     }
```

Now write the `AddressBookGUI` class that uses the `AddressBook` class. This class can have a `JTextField` for the key, and a `JTextArea` for the value. Another `JTextArea` can be used to display output and status information and `JButton` objects can be used for ”Save”, ”Search”, ”Delete”, ”Add”, and ”Display All” operations.

► Exercise 13.17 (Map version of an address book)
Repeat the previous exercise using a map of the type `Map<String,String>` instead of a list to hold the address book entries. Now there is no need for the inner `AddressBookEntry` class. The GUI class will be the same as in the previous exercise, only the `AddressBook` class will change.

► Exercise 13.18 (Map version of a telephone directory)
Write a GUI version of telephone directory that uses a sorted map.

 Copyright 2002, 2005, 2007, Barry G. Adams