Solutions to Exercises in Chapter 9

9.1 The set of primary colors, the set of colors of the rainbow, and the set of colors in the national flag (assuming, for the sake of illustration, that the flag's colors are red, white, and blue) are:

primary	=	{red, green, blue}
rainbow	=	{red, orange, yellow, green, blue, indigo, violet}
flag	=	{red, white, blue}

Their cardinalities are:

#primary	=	3
#rainbow	=	7
#flag	=	3

The set of rainbow colors in the national flag is *rainbow* ∩ *flag*, which evaluates to {red, blue}.

The set of rainbow colors not in the national flag is *rainbow* – *flag*, which evaluates to {orange, yellow, green, indigo, violet}.

The assertion that all colors in the national flag occur in the rainbow is *rainbow* ⊇ *flag*, which evaluates to false.

The assertion that the national flag contains exactly the primary colors is *primary* = *flag*, which evaluates to false.

9.3 Removing step 2.1 of Eratosthenes' sieve algorithm would cause it to remove multiples of an integer, *i*, that is not a member of the set *sieve*. This could happen only if *i* was previously removed as a multiple of some smaller integer, say *j*. In this case, all multiples of *j* will have already been removed, since any multiple of *i* is also a multiple of *j*. So removing step 2.1 will only cause the algorithm to remove integers from *sieve* that have already been removed (which is a harmless but time-consuming operation).

9.4 To compute the set of non-primes less than *m* (where *m* > 0):

```
1. Set non-primes = {1}.
2. For $i = 2, 3, \ldots$, while $i^2 < m$, repeat:
   2.1. If i is not a member of non-primes:
        2.1.1. Add all multiples of i to non-primes.
3. Terminate with answer non-primes.
```

9.5 The following `score` method returns the *percentage* of the key words that actually occur in the document:

```
public static int score (String docname,
                WordSet keywords) {
  if (keywords.size() == 0)  return 0;
  BufferedReader doc =
      new BufferedReader(
        new FileReader(docname));
  WordSet docwords = readAllWords(doc);
  doc.close();
  int matches = 0;
  Iterator iter = keywords.iterator();
  while (iter.hasNext()) {
    String kw = (String) iter.next();
    if (docwords.contains(kw))  matches++;
  }
  return matches * 100 / keywords.size();
}
```

An alternative (but less efficient) approach would be to calculate the intersection of `docwords` and `keywords` (using the `retainAll` operation), and then use the cardinality of the resulting set to calculate the percentage of matching keywords.

Sometimes we wish to compute not the actual intersection, union, or difference of two sets, but only the cardinality of the resulting set. This cardinality can be computed efficiently using an iterator, as above. However, it would be more convenient if new operations were added to the `Set` interface:

```
public int unionSize (Set that);
// Return the cardinality of the union of this set and that.

public int differenceSize (Set that);
// Return the cardinality of the difference of this set and that.

public int intersectionSize (Set that);
// Return the cardinality of the intersection of this set and that.
```

9.6 Alternative `Set` interface in which the mutative transformers are replaced by applicative transformers:

```
public interface Set {

  // Each Set object is a set whose members are objects.

  //////////// Accessors ////////////

  …

  //////////// Transformers ////////////

  public Set empty ();
  // Return an empty set.

  public Set union1 (Object obj);
  // Return the new set obtained by adding obj to this set. (Throw a
  // ClassCastException if this set cannot contain an object
  // with the class of obj.)

  public Set difference1 (Object obj);
  // Return the new set obtained by removing obj from this set.

  public Set union (Set that);
  // Return the union of this set and that.

  public Set difference (Set that);
  // Return the difference between this set and that.

  public Set intersection (Set that);
  // Return the intersection of this set and that.
```

```
        //////////// Iterator ////////////
        …
    }
```

Implement the `union1` operation in the `ArraySet` class as follows:

```
        public Set union1 (Object obj) {
            ArraySet that =
                    new ArraySet(members.length);
            for (int i = 0; i < this.card; i++)
                that.members[i] = this.members[i];
            that.card = this.card;
            Comparable it = (Comparable) obj;
            int pos = that.search(it);
            if (! it.equals(that.members[pos])) {
                // it is not already a member.
                if (that.card == that.members.length)
                    that.expand();
                for (int i = that.card; i > pos; i--)
                    that.members[i] = that.members[i-1];
                that.members[pos] = it;
                that.card++;
            }
            return that;
        }
```

Implement the other operations similarly by first creating a copy of this set and then modifying the copy accordingly.

If the `ArraySet` class were made to implement the `Cloneable` interface, a copy of this set could be created by a single statement:

```
    ArraySet that = this.clone();
```

9.7 The `Set` operations could be specified formally as follows (where b', i', or s' is the resulting boolean, integer, or set, respectively):

s.`isEmpty()`	$b' = (s = \{\ \})$
s.`size()`	$i' = \#s$
s.`contains(`x`)`	$b' = x \in s$
s.`equals(`s_2`)`	$b' = (s = s_2)$
s.`containsAll(`s_2`)`	$b' = (s \supseteq s_2)$
s.`clear()`	$s' = \{\ \}$
s.`add(`x`)`	$s' = s \cup \{x\}$
s.`remove(`x`)`	$s' = s - \{x\}$
s.`addAll(`s_2`)`	$s' = s \cup s_2$
s.`removeAll(`s_2`)`	$s' = s - s_2$
s.`retainAll(`s_2`)`	$s' = s \cap s_2$

9.8 Assuming the array representation of bounded sets, Figure S9.1 shows how the set of words in each of the sentences would be represented, assuming that no distinction is made between upper and lower case.

The complete code for the `ArraySet` class can be found on the companion Web site.

(i) {‘to’, ‘be’, ‘or’, ‘not’}

0	1	2	3	*card* = 4	5	6	7	8	9	10	11
be	not	or	to								

(ii) {‘the’, ‘moon’, ‘is’, ‘a’, ‘balloon’}

0	1	2	3	4	*card* = 5	6	7	8	9	10	11
a	balloon	is	moon	the							

(iii) {‘the’, ‘rain’, ‘in’, ‘spain’, ‘falls’, ‘mainly’, ‘on’,

0	1	2	3	4	5	6	7	*card* = 8	9	10	11
falls	in	mainly	on	plain	rain	spain					

Figure S9.1 Representation of sets of words using a sorted array (with *maxcard* = 12).

9.9 Figure S9.2 shows three possible representations of the set of words {‘be’, ‘not’, ‘or’, ‘to’} when the set is represented by an unsorted array.

The modifications to the code should be straightforward.

The algorithms and their complexities are summarized in Table S9.3

	0	1	2	3	*card* = 4	5
(i)	to	be	or	not		

	0	1	2	3	*card* = 4	5
(ii)	be	not	or	to		

	0	1	2	3	*card* = 4	5
(iii)	not	or	be	to		

Figure S9.2 Three possible representations of a set of words using an unsorted array (with *maxcard* = 6).

Table S9.3 Implementation of bounded sets using unsorted arrays: summary of algorithms (where n, n_1, and n_2 are the cardinalities of the sets involved).

Operation	Algorithm	Time complexity
`contains`	linear search	$O(n)$
`equals`	repeated linear search	$O(n_1 n_2)$
`containsAll`	variant of repeated linear search	$O(n_1 n_2)$
`add`	linear search combined with insertion	$O(n)$
`remove`	linear search combined with deletion	$O(n)$
`addAll`	repeated insertions	$O(n_1 n_2)$
`removeAll`	repeated deletions	$O(n_1 n_2)$
`retainAll`	linear search combined with removal	$O(n_1 n_2)$

9.10 The modifications to remove the instance variable `card` from the array implementation of bounded sets are straightforward. One consequence is that that the `size` operation now has $O(n)$ time complexity. The other consequence is that each operation must either check for null elements in the array, or first determine the set’s cardinality and then proceed as before.

9.11 A suitable representation for sets of colors is affected by the fact that `java.awt.Color` objects are not comparable. This forces an unsorted implementation. There is little to choose between an unsorted array implementation and an unsorted SLL implementation.

9.12 A suitable representation for sets of ISO-LATIN-1 (8-bit) characters is `IntSet`. Each set has at most 256 members, and an array of 256 booleans would not waste too much space. Most sets of characters, however, are likely to be quite small (less than about 50 elements), so `ArraySet` might also be suitable.

A suitable representation for sets of Unicode (16-bit) characters is `ArraySet`. Since there are 65536 possible characters, `IntSet` would be too wasteful of space. (*Note:* A hash table (Chapter 12) or a balanced search tree (Chapters 10 and 16) would however be more suitable.)

Given that `digits` and `letters` represent sets of ISO-LATIN-1 characters, we can write the following expressions:

`digits.contains(ch)`	(tests whether `ch` is a digit)
`letters.contains(ch)`	(tests whether `ch` is a letter)
`letters.contains(ch) || digits.contains(ch)`	(tests whether `ch` is a letter or digit)

9.13 A suitable representation for sets of countries would depend on the anticipated cardinalities. If the cardinalities are expected to be small, `ArraySet` would be reasonably efficient whilst not wasting too much space. If the cardinalities are large or unknown, neither `ArraySet` nor `LinkedSet` is really suitable. (*Note:* A hash table (Chapter 12) or a balanced search tree (Chapters 10 and 16) would be suitable.)

9.14 In the `IntSet` implementation of small-integer sets, the `size` operation has time complexity $O(m)$, since counting the number of true elements requires the entire contents of the array to be inspected.

Modifications to include the set's cardinality as an instance variable are straightforward. When adding (or deleting) an element, the cardinality should be increased (or decreased) only if the array component previously contained false (or true). The `size` operation's time complexity becomes $O(1)$.

9.16 The UNIX `spell` command uses a stored vocabulary that omits suffices (e.g., it contains 'abandon' but not 'abandons', 'abandoned', etc.) To test whether a given word is recognized, the `spell` command first strips any recognized suffices (such as '-s' and 'ed') using certain rules, and then tests whether the resulting word stem is in the vocabulary. It omits to ensure that the word stem and suffix actually go together. In consequence, many non-words are incorrectly recognized (e.g., 'abandonization' is recognized because '-ization' is a recognized suffix and the word stem 'abandon' is in the vocabulary).

9.17 The `recognizes` method of Program 9.16 uses vocabularies represented by SLLs. Suppose that the main, user, and ignored vocabularies contain m, u, and i members, respectively.

(a) If the word is in the main vocabulary only, the method will require 1 (best-case), $m/2$ (average-case), or m (worst-case) comparisons.

(b) If the word is in the user vocabulary only, the method will require $m + 1$ (best-case), $m + u/2$ (average-case), or $m + u$ (worst-case) comparisons.

(c) If the word is in the ignored vocabulary only, the method will require $m + u + 1$ (best-case), $m + u + i/2$ (average-case), or $m + u + i$ (worst-case) comparisons.

9.20 A possible contract for a bag ADT is shown in Program S9.4.

We can represent a bag in the same way as a set, except that each member is paired with a positive integer (which is the number of instances of that member). Figures S9.5 and S9.6 illustrate sorted array and sorted SLL representations of bags.

Programs S9.5 outlines an array implementation of the bag ADT. An SLL implementation would be written along similar lines.

```
public interface Bag {

    // Each Bag object is a bag whose members are Comparable objects.

    //////////// Accessors ////////////

    public boolean isEmpty ();
    // Return true if and only if this bag is empty.

    public int size ();
    // Return the cardinality of this bag, i.e., the total number of instances.

    public boolean contains (Object obj);
    // Return true if and only if obj is a member of this bag.

    public int instances (Object obj);
    // Return the number of instances of obj in this bag (0 if obj is not a
    // member).

    public boolean equals (Bag that);
    // Return true if and only if this bag is equal to that.

    public boolean containsAll (Bag that);
    // Return true if and only if this bag subsumes that (i.e., every member of
    // that has at least as many instances in this bag).

    //////////// Transformers ////////////

    public void clear ();
    // Make this bag empty.

    public void add (Object obj);
    // Add an instance of obj to this bag. (Throw a ClassCastException
    // if this bag cannot contain an object with the class of obj.)

    public void remove (Object obj);
    // Remove an instance of obj from this bag.

    public void addAll (Bag that);
    // Make this bag the union of itself and that.

    //////////// Iterator ////////////

    public Iterator iterator();
    // Return an iterator that will visit all members of this bag, in no particular
    // order.

}
```

Program S9.4 Contract for a bag ADT.

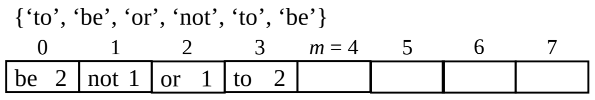

Figure S9.5 Representation of a bounded bag using an array (with *maxm* = 8).

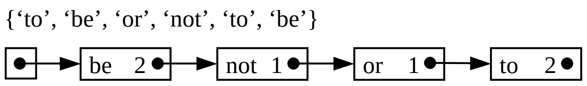

Figure S9.6 Representation of a bag using an SLL.

```
public interface ArrayBag {

   // Each ArrayBag object is a bag whose members are Comparable
   // objects.

   // This bag is represented as follows. The number of distinct members is
   // held in m. The members are held in the sorted subarray
   // entries[0…m–1], each member paired with its number of instances.
   private ArrayBag.Entry[] entries;
   private int m;

   //////////// Constructor ////////////

   public ArrayBag (int maxm) {
      entries = new ArrayBag.Entry[maxm];
      m = 0;
   }

   //////////// Accessors ////////////

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

   public int size () {
      int s = 0;
      for (int i = 0; i < m; i++)
         s += entries[i].count;
      return s;
   }

   public int instances (Object obj) {
      if (obj instanceof Comparable) {
         Comparable it = (Comparable) obj;
         int pos = search(it);
         if (it.equals(entries[pos].member))
            return entries[pos].count;
      }
      return 0;
   }

   …

   //////////// Transformers ////////////

   public void add (Object obj) {
      Comparable it = (Comparable) obj;
      int pos = search(it);
      if (it.equals(entries[pos].member))
         entries[pos].count++;
      else {
         if (m == entries.length)  expand();
         for (int i = m; i > pos; i--)
            entries[i] = entries[i-1];
         entries[pos] = new ArrayBag.Entry(it, 1);
         m++;
      }
   }

   …

   //////////// Auxiliary methods ////////////

   private void expand () { … }

   private int search (Comparable target) { … }
```

Program S9.7 Outline implementation of bounded bags using arrays *(continued on next page)*.

```
   ////////////  Auxiliary inner class  ////////////

   private class Entry {

      // Each ArrayBag.Entry object is a pair consisting of a bag
      // member and the number of instances of that member.

      private Comparable member;
      private int count;

      private Entry (Comparable mem, int n) {
         member = mem;  count = n;
      }

   }
}
```

Program S9.7 Outline implementation of bounded bags using arrays *(continued)*.