# Data Structures for Java

William H. Ford
William R. Topp

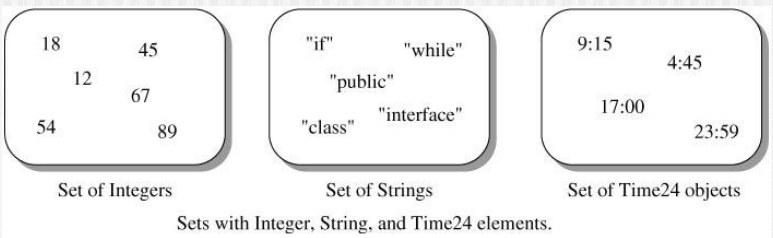
# Chapter 19
# Sets and Maps

Bret Ford



## Set Collection

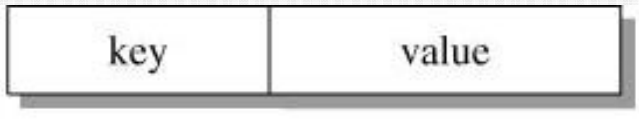
- A Set implements the Collection interface and requires that each element be unique.




Sets with Integer, String, and Time24 elements.

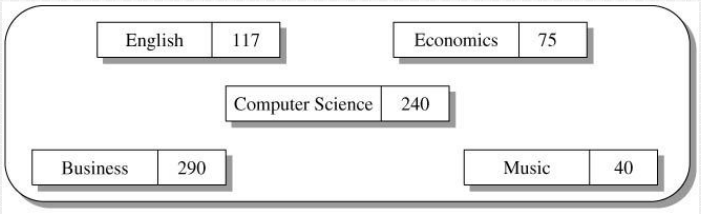
## Map

- A map stores an element as a *key-value pair*. In a pair, the first field is the key which is an attribute that uniquely identifies the element. The second field is the value which is an object associated with the key.




## Map (continued)

The figure illustrates a map that might be used for university administration. The map is a collection of String-Integer pairs to denote the number of students in each of its degree programs. The name of the degree program is the key and the number of students is the value



## Map (continued)

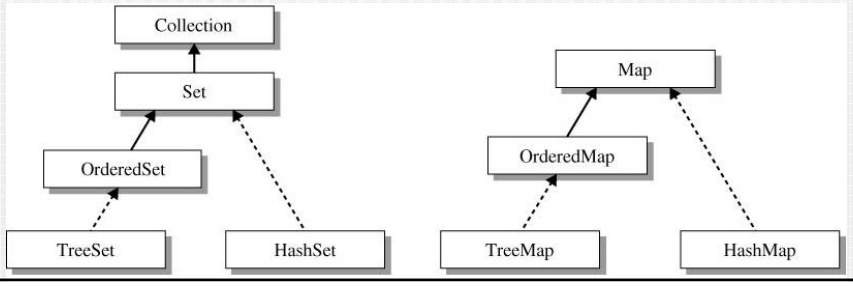
- A map collection uses the key to access the value field of the entry. For instance, assume degreeMap is the map collection for departments and student counts. The operation degreeMap.get("English") returns the Integer component with value 117, which identifies the number of English students.

## Map (concluded)

- The Set interface extends the Collection interface, but the Map interface is separate, since if defines methods not relevant to general collections.

## Set and Map Interfaces

- The interfaces and collection classes that define sets and maps. The dashed lines indicate that the class implements the interface.




## TreeSet Collection

- TreeSet is implemented by a binary search tree. As a result, it implements the OrderedSet interface that extends the Set interface and defines the methods first() and last().

# TreeSet Collection (concluded)

| class TreeSet<T> implements OrderedSet<T> | ds.util |
|---|---|
| | **Constructor** |
| | **TreeSet**()<br>Creates an empty ordered set whose elements have a specified type that must implement the Comparable interface |
| | **Methods** |
| T | **first**()<br>Returns the minimum value of the elements in the set |
| T | **last**()<br>Returns the maximum value of the elements in the set |
| String | **toString**()<br>Returns a string containing a comma separated ordered list of elements enclosed in brackets. |

# Spell Checker

- A set is an ideal structure for the implementation of a simple spelling checker.
  - The file "dict.dat" is a set of strings containing approximately 25,000 words in lower case.
  - For each word, call contains() to determine whether the word is in the dictionary set. If not, assume the word is misspelled and interact with the user for instructions on how to proceed.

## spellChecker()

```
public static void spellChecker(
String filename)
{
   // sets storing the dictionary and the
   // misspelled words
   TreeSet<String> dictionary = new TreeSet<String>(),
         misspelledWords = new TreeSet<String>();

   Scanner dictFile = null, docFile = null;

   // create Scanner objects to input dictionary
   // and document
   try
   {
      // dictionary and document streams
      dictFile = new Scanner(new FileReader("dict.dat"));
      docFile = new Scanner(new FileReader(filename));
   }
```

## spellChecker() (continued)

```
   catch(FileNotFoundException fnfe)
   {
      System.err.println("Cannot open a file");
      System.exit(1);
   }
   // string containing each word from the
   // dictionary and from the document
   String word;
   // user response when a misspelled word is noted
   String response;

   // insert each word from file "dict.dat" into a set
   while(dictFile.hasNext())
   {
      // input next word and add to dictionary
      word = dictFile.next();
      dictionary.add(word);
   }
```

## spellChecker() (continued)

```
// read the document word by word and check spelling
while(docFile.hasNext())
{
   // get the next word from the document
   word = docFile.next();

   // look word up in the dictionary; if not
   // present assume word is misspelled; prompt
   // user to add word to the dictionary, ignore
   // it, or flag as misspelled
   if (!dictionary.contains(word))
   {
      System.out.println(word);
      System.out.print(
            "   'a'(add)  'i'(ignore)  " +
            "'m'(misspelled) ");
      response = keyIn.next();
```

## spellChecker() (concluded)

```
         // if response is 'a' add to dictionary;
         // if not ignored, add to set of
         // misspelled words
         if (response.charAt(0) == 'a')
            dictionary.add(word);
         else if (response.charAt(0) == 'm')
            misspelledWords.add(word);
      }
   }

   // display the set of misspelled words
   System.out.println("\nMisspelled words: " +
                      misspelledWords);
}
```

## Program 19.1

```
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.util.Scanner;
import ds.util.TreeSet;

public class Program19_1
{
   // keyboard input stream used by main() and spellChecker()
   static Scanner keyIn = new Scanner(System.in);

   public static void main(String[] args)
   {
      String fileName;
      // enter the file name for the document
      System.out.print("Enter the document to " +
                       "spell check: ");
      fileName = keyIn.next();
```

## Program 19.1 (concluded)

```
         // check the spelling
         spellChecker(fileName);
      }

      < method spellchecker() listed in the
        program discussion >
   }
```

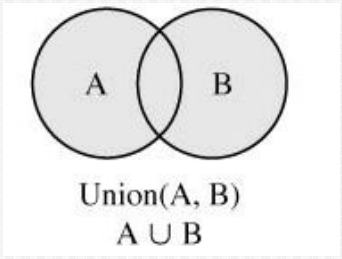
## Program 19.1 (File “spell.txt”)

```
teh message contians the url for the web-page
and a misspeled url for the email adress
```

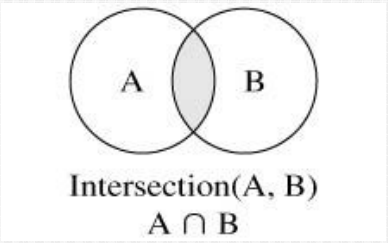
## Program 19.1 (Run)

```
Enter the document to spell check: spell.txt
teh
   'a'(add)  'i'(ignore)  'm'(misspelled) m
contians
   'a'(add)  'i'(ignore)  'm'(misspelled) m
url
   'a'(add)  'i'(ignore)  'm'(misspelled) a
web-page
   'a'(add)  'i'(ignore)  'm'(misspelled) i
misspeled
   'a'(add)  'i'(ignore)  'm'(misspelled) m
email
   'a'(add)  'i'(ignore)  'm'(misspelled) i
adress
   'a'(add)  'i'(ignore)  'm'(misspelled) m

Misspelled words: [adress, contians, misspeled, teh]
```

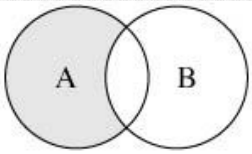
# Set Operators



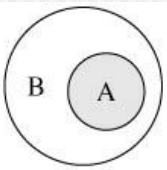

Union(A, B)
$A \cup B$




Intersection(A, B)
$A \cap B$




Difference(A, B)
$A - B$




Subset (A, B)
$A \subseteq B$

# Design for Implementation of Set Operations

```
public static <T> Set<T> setOp(Set<T> setA, Set<T> setB)
{
   Set<T> returnSet;

   // returnSet is a TreeSet or HashSet object depending on
   // argument type
   if (setA instanceof OrderedSet)
      returnSet = new TreeSet<T>();
   else
      returnSet = new HashSet<T>();
   . . .
}
```

## union(setA, setB)

```
public static <T> Set<T> union (Set<T> setA, Set<T> setB)
{
   Set<T> setUnion;

   // allocate concrete collection object for setUnion
   . . .

   // use iterator to add elements from setA
   Iterator<T> iterA = setA.iterator();
   while (iterA.hasNext()) setUnion.add(iterA.next());

   // use iterator to add non-duplicate
   // elements from setB
   Iterator<T> iterB = setB.iterator();
   while (iterB.hasNext()) setUnion.add(iterB.next());

   return setUnion;
}
```

## intersection(setA, setB)

```
public static <T> Set<T> intersection (
Set<T> setA, Set<T> setB)
{
   Set<T> setIntersection;
   T item;

   // allocate concrete collection object
   // for setIntersection
   . . .
```

## intersection(setA, setB) (2)

```
        // scan elements in setA and check whether
        // they are also elements in setB
        Iterator<T> iterA = setA.iterator();
        while (iterA.hasNext())
        {
           item = iterA.next();
           if (setB.contains(item))
              setIntersection.add(item);
        }

        return setIntersection;
     }
```

## difference(setA, setB)

```
public static <T> Set<T> difference (
Set<T> setA, Set<T> setB)
{
   Set<T> setDifference;
   T item;

   // allocate concrete collection object for
   // setDifference; then scan elements in setA and
   //check whether they are not in setB
   Iterator<T> iterA = setA.iterator();
   while (iterA.hasNext())
   {
      item = iterA.next();
      if (!setB.contains(item))
         setDifference.add(item);
   }
   return setDifference;
}
```

## subset(setA, setB)

```
public static <T> boolean subset(
Set<T> setA, Set<T> setB)
{
   return intersection(setA, setB).size() ==
         setA.size();
}
```

## Program 19.2

```
import java.io.*;
import java.util.Scanner;
import ds.util.Sets;
import ds.util.TreeSet;
import ds.util.Set;

public class Program19_2
{
   public static void main(String[] args)
   {
      // declare sets for current and new
      // computer accounts
      Set<String> oldAcct = new TreeSet<String>(),
         currAcct = new TreeSet<String>(), processAcct,
         newAcct, carryOverAcct, obsoleteAcct;
```

## Program 19.2 (continued)

```
// input names from file into the set
try
{
   readAccounts("oldAcct.dat", oldAcct);
   readAccounts("currAcct.dat", currAcct);
}
catch(IOException ioe)
{
   System.err.println("Cannot open account file");
   System.exit(1);
}

// use set union to determine all
// accounts to update
processAcct =
   Sets.union(currAcct, oldAcct);
```

## Program 19.2 (continued)

```
// use set intersection to determine
// carryover accounts
carryOverAcct =
   Sets.intersection(currAcct, oldAcct);

// use set difference to determine new
// and obsolete accounts
newAcct = Sets.difference(currAcct, oldAcct);
obsoleteAcct = Sets.difference(oldAcct, currAcct);
```

## Program 19.2 (continued)

```
        // output statements provide a set
        // description and a list of elements
        // in the set
        System.out.println("Old Accounts:       " +
              oldAcct);
        System.out.println("Current Accounts:   " +
              currAcct);
        System.out.println("Process Accounts:   " +
              processAcct);
        System.out.println("New Accounts:       " +
              newAcct);
        System.out.println("Carryover Accounts: " +
              carryOverAcct);
        System.out.println("Obsolete Accounts:  " +
              obsoleteAcct);
    }
```

## Program 19.2 (concluded)

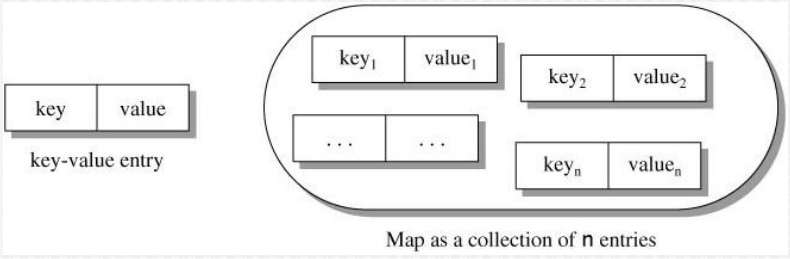
```
    public static void readAccounts(
    String filename, Set<String> t) throws IOException
    {
       Scanner sc = new Scanner(
             new FileReader(filename));
       String acctName;

       // input the set of current accounts
       while(sc.hasNext())
       {
          acctName = sc.next();
          t.add(acctName);
       }
    }
}
```

## Program 19.2 (Run)

```
Old Accounts:       [fbrue, gharris, lhung, tmiller]
Current Accounts:   [ascott, fbrue, wtubbs]
Process Accounts:   [ascott, fbrue, gharris, lhung,
                    tmiller, wtubbs]
New Accounts:       [ascott, wtubbs]
Carryover Accounts: [fbrue]
Obsolete Accounts:  [gharris, lhung, tmiller]
```

## Maps

- A map stores data in entries, which are key-value pairs. A key serves like an array index to locate the corresponding value in the map. As a result, we call a map an associative array.



## The Map Interface

- `Map` method `size` returns the number of key/value pairs in the `Map`.
- `Map` method `isEmpty` returns a `boolean` indicating whether the `Map` is empty.
- `Map` method `clear()` is identical to the one of the Collection interface.
- `Map` method `put` creates a new entry in the map or replaces an existing entry's value.
  - Method `put` returns the key's prior associated value, or `null` if the key was not in the map.
- `Map` method `get` returns the value associated to the specified key in the map.

## The Map Interface (2)

- `Map` method `containsKey` determines whether a key is in a map.
- `Map` method `remove()` uses only the key as an argument.
- A map does not have an iterator to scan its elements.
  - `HashMap` method `keySet` and `HashMap` method `entrySet` return the keys and the entries in a map as a set.

# The Map Interface (concluded)

| interface MAP<K,V> (partial) | ds.util |
|---|---|
| void | **clear**()<br>Removes all mappings from this map. |
| boolean | **containsKey**(Object key)<br>Returns `true` if this map contains a mapping for the specified key. |
| boolean | **isEmpty**()<br>Returns `true` if this map contains no key-value mappings. |
| V | **remove**(Object key)<br>Removes the mapping for this key from this map if present. Returns the previous value associated with specified key, or `null` if there was no mapping for key. |
| int | **size**()<br>Returns the number of key-value mappings in this map. |
| | **Access/Update Methods** |
| K | **get**(Object key)<br>Returns the value to which this map maps the specified key or `null` if the map contains no mapping for this key. |
| V | **put**(K key, V value)<br>Associates the specified value with the specified key in this map. Returns the previous value associated with key, or `null` if there was no mapping for key. |

# Map Collection

- Three of the several classes that implement interface `Map` are `Hashtable`, `HashMap` and `TreeMap`.
- `Hashtable`s and `HashMap`s store elements in hash tables
  - Hashing is a high-speed scheme for converting keys into unique array indices.
  - The load factor is the ratio of the number of occupied cells in the hash table to the total number of cells in the hash table.
  - A hash table's load factor affects the performance of hashing schemes. The closer this ratio gets to 1.0, the greater the chance of collisions.
- `TreeMap`s store elements in trees.

```
1  // Fig. 20.18: WordTypeCount.java
2  // Program counts the number of occurrences of each word in a String.
3  import java.util.Map;
4  import java.util.HashMap;
5  import java.util.Set;
6  import java.util.TreeSet;
7  import java.util.Scanner;
8
9  public class WordTypeCount
10 {
11    public static void main( String[] args )
12    {
13       // create HashMap to store String keys and Integer values
14       Map< String, Integer > myMap = new HashMap< String, Integer >();
15
16       createMap( myMap ); // create map based on user input
17       displayMap( myMap ); // display map content
18    } // end main
19
20    // create map from user input
21    private static void createMap( Map< String, Integer > map )
22    {
```

**Fig. 20.18** | Program counts the number of occurrences of each word in a String. (Part 1 of 4.)

```
23       Scanner scanner = new Scanner( System.in ); // create scanner
24       System.out.println( "Enter a string:" ); // prompt for user input
25       String input = scanner.nextLine();
26
27       // tokenize the input
28       String[] tokens = input.split( " " );
29
30       // processing input text
31       for ( String token : tokens )
32       {
33          String word = token.toLowerCase(); // get lowercase word
34
35          // if the map contains the word
36          if ( map.containsKey( word ) ) // is word in map
37          {
38             int count = map.get( word ); // get current count
39             map.put( word, count + 1 ); // increment count
40          } // end if
41          else
42             map.put( word, 1 ); // add new word with a count of 1 to map
43       } // end for
44    } // end method createMap
```

**Fig. 20.18** | Program counts the number of occurrences of each word in a String. (Part 2 of 4.)

```
45
46    // display map content
47    private static void displayMap( Map< String, Integer > map )
48    {
49       Set< String > keys = map.keySet(); // get keys
50
51       // sort keys
52       TreeSet< String > sortedKeys = new TreeSet< String >( keys );
53
54       System.out.println( "\nMap contains:\nKey\t\tValue" );
55
56       // generate output for each key in map
57       for ( String key : sortedKeys )
58          System.out.printf( "%-10s%10s\n", key, map.get( key ) );
59
60       System.out.printf(
61          "\nsize: %d\nisEmpty: %b\n", map.size(), map.isEmpty() );
62    } // end method displayMap
63 } // end class WordTypeCount
```

**Fig. 20.18** | Program counts the number of occurrences of each word in a String. (Part 3 of 4.)

```
Enter a string:
this is a sample sentence with several words this is another sample
sentence with several different words

Map contains:
Key             Value
a                   1
another             1
different           1
is                  2
sample              2
sentence            2
several             2
this                2
with                2
words               2

size: 10
isEmpty: false
```

**Fig. 20.18** | Program counts the number of occurrences of each word in a String. (Part 4 of 4.)