Chapter 15 -The Java Collections Framework

ListDemo.java

```
LinkedList<String> staff = new LinkedList<>();
staff.addLast("Diana");
staff.addLast("Harry");
staff.addLast("Romeo");
staff.addLast("Tom");


// | in the comments indicates the iterator position


ListIterator<String> iterator = staff.listIterator(); //|DHRT
iterator.next(); //D|HRT
iterator.next(); // DH|RT
// Add more elements after second element
iterator.add("Juliet"); // DHJ|RT
iterator.add("Nina"); // DHJN|RT


iterator.next(); // DHJNR|T


// Remove last traversed element
iterator.remove(); // DHJN|T


// Print all elements
System.out.println(staff);
System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
```

Self Check 15.5

Do linked lists take more storage space than arrays of the same size?

Answer: Yes, for two reasons. A linked list needs to store the neighboring node references, which are not needed in an arry. Moreover, there is some overhead for storing an object. In a linked list, each node is a separate object that incurs this overhead, whereas an array is a single object.

Self Check 15.6

Why don't we need iterators with arrays?

Answer: We can simply access each array element with an integer index.

Self Check 15.7

Suppose the `list` letters contains elements `"A"`, `"B"`, `"C"`, and `"D"`. Draw the contents of the list and the iterator position for the following operations:

```
ListIterator<String> iter = letters.iterator();
iter.next();
iter.next();
iter.remove();
iter.next();
iter.add("E");
iter.next();
iter.add("F");
```

Answer:

```
|ABCD
A|BCD
AB|CD
A|CD
AC|D
ACE|D
ACED|
ACEDF|
```

Self Check 15.8

Write a loop that removes all strings with length less than four from a linked list of strings called `words`.

Answer:

```
ListIterator<String> iter = words.iterator();
while (iter.hasNext())
{
   String str = iter.next();
   if (str.length() < 4) { iter.remove(); }
}
```

Self Check 15.9

Write a loop that prints every second element of a linked list of strings called `words`.

Answer:

```
ListIterator<String> iter = words.iterator();
while (iter.hasNext())
{
   System.out.println(iter.next());
   if (iter.hasNext())
   {
      iter.next(); // Skip the next element
   }
}
```

Sets

- A set organizes its values in an order that is optimized for efficiency.
- May not be the order in which you add elements.
- Inserting and removing elements is more efficient with a set than with a list.

Sets

- The `Set` interface has the same methods as the `Collection` interface.
- A set does not admit duplicates.
- Two implementing classes

  `HashSet` based on hash table

  `TreeSet` based on binary search tree

- A Set implementation arranges the elements so that it can locate them quickly.

Sets

- In a hash table

  Set elements are grouped into smaller collections of elements that share the same characteristic.

  Grouped by an integer hash code that is computed from the element.

- Elements in a hash table must implement the method `hashCode`.
- Must have a properly defined `equals` method.
- You can form hash sets holding objects of type `String`, `Integer`, `Double`, `Point`, `Rectangle`, or `Color`.

  ```
  HashSet<String>, HashSet<Rectangle>
  ```

  On this shelf, books of the same color are grouped together.
- Similarly, in a hash table, objects with the same hash code are placed in the same group.

Sets: hashCode and equals method

- Elements in a hash table must implement the method `hashCode` and Must have a properly defined `equals` method.

```
public class Person {
   String firstname;
   String lastname;
   public Person(String firstname, String lastname) {
      this.firstname = firstname;
      this.lastname = lastname;
   }

   @Override
   public int hashCode() {
      return firstname.hashCode()+lastname.hashCode();
   }

   @Override
   public boolean equals(Object obj) {
   if (!(obj instanceof Person))
         return false;
      if (obj == this)
         return true;
      Person u=(Person) obj;
      return u.firstname.equals(firstname) && u.lastname.equals(lastname);
   }
}
```

Sets

```
import java.util.HashSet;
public class ExMain {
public static void main(String[] args) {
        // Create a HashSet object called numbers
        HashSet<Integer> numbers = new HashSet<Integer>();
        // Add values to the set
        numbers.add(4);
        numbers.add(7);
        numbers.add(8);
        // Show which numbers between 1 and 10 are in the set
        for(int i = 1; i <= 10; i++) {
                if(numbers.contains(i)) {
                System.out.println(i + " was found!");
                } else {
                System.out.println(i + " was not found!");
                }
        }
}
}
```

Sets: addAll method

```
import java.util.HashSet;
public class HashSetExample {
  public static void main(String[] args) {
    HashSet<String> vegetableSet = new HashSet<>();

    //Adding elements to the HashSet.
    vegetableSet.add("Potato");
    vegetableSet.add("Onion");
    vegetableSet.add("Broccoli");

    //Printing the elements in the HashSet.
    System.out.println("Values in the set are:= "+ vegetableSet);

    //Creating a sample HashSet.
    HashSet<String> sampleSet = new HashSet<>();

    //Adding elements to the new set.
    sampleSet.add("Tomato");
    sampleSet.addAll(vegetableSet);

    //Printing the elements in the HashSet.
    System.out.println("Values in the sample HashSet are:= "+sampleSet);
  }
}
```

Sets

- In a `TreeSet`

  Elements are kept in sorted order

- Elements are stored in nodes.
- The nodes are arranged in a tree shape,

  Not in a linear sequence

- You can form tree sets for any class that implements the `Comparable` interface:

  Example: `String` or `Integer`.

Sets

- Use a `TreeSet` if you want to visit the set's elements in sorted order.

  Otherwise choose a `HashSet`.

  It is a bit more efficient — if the hash function is well chosen.

Sets

- Store the reference to a `TreeSet` or `HashSet` in a `Set<String>` variable:

```
   Set<String> names = new HashSet<>();
or
   Set<String> names = new TreeSet<>();
```

- After constructing the collection object:

  The implementation no longer matters

  Only the interface is important

Working with Sets

- Adding and removing elements:

```
names.add("Romeo");
names.remove("Juliet");
```

- Sets don't have duplicates.

  Adding a duplicate is ignored.

- Attempting to remove an element that isn't in the set is ignored.
- The `contains` method tests whether an element is contained in the set:

```
if (names.contains("Juliet")) . . .
```

  The `contains` method uses the `equals` method of the element type

Working with Sets

- To process all elements in the set, get an iterator.
- A set iterator visits the elements in the order in which the set implementation keeps them.

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   Do something with name
}
```

- You can also use the “for each” loop

```
for (String name : names)
{
   Do something with name
}
```

- You cannot add an element to a set at an iterator position - A set is unordered.
- You can remove an element at an iterator position.
- The iterator interface has no `previous` method.

Working with Sets

Table 4 Working with Sets	
`Set<String> names;`	Use the interface type for variable declarations.
`names = new HashSet<>();`	Use a `TreeSet` if you need to visit the elements in sorted order.
`names.add("Romeo");`	Now `names.size()` is 1.
`names.add("Fred");`	Now `names.size()` is 2.
`names.add("Romeo");`	`names.size()` is still 2. You can't add duplicates.
`if (names.contains("Fred"))`	The `contains` method checks whether a value is contained in the set. In this case, the method returns `true`.
`System.out.println(names);`	Prints the set in the format `[Fred, Romeo]`. The elements need not be shown in the order in which they were inserted.

`for (String name : names)`
`{`
`  . . .`
`}`
	Use this loop to visit all elements of a set.

`names.remove("Romeo");`	Now `names.size()` is 1.
`names.remove("Juliet");`	It is not an error to remove an element that is not present. The method call has no effect.

Sets: setup

```
HashSet<String> vowelSet = new HashSet<>();

    //Adding elements to the HashSet.
    vowelSet.add("A");
    vowelSet.add("E");
    vowelSet.add("I");
    vowelSet.add("O");
    vowelSet.add("U");

    //Printing the elements.
    System.out.println("Values in the vowel set are:="+vowelSet);

    //Creating a new Set.
    HashSet<String> consonantSet = new HashSet<>();
    consonantSet.add("A");
    consonantSet.add("K");
    consonantSet.add("P");
    consonantSet.add("U");

    //Printing the elements.
    System.out.println("Values in the consonant set are:="+consonantSet);
```

Sets: set operations

```
//Intersection of two sets.
vowelSet.retainAll(consonantSet);
System.out.println("The result of intersection of two sets is:="+vowelSet);
// A U

//Union of two sets.
consonantSet.addAll(vowelSet);
System.out.println("Values in the set after union of sets:="+consonantSet);
// P A U K

//Difference between two sets.
consonantSet.removeAll(vowelSet);
System.out.println("The result of difference between two sets is :="+consonantSet);
// P K
```

SpellCheck Example Program

- Read all the correctly spelled words from a dictionary file

  Put them in a set

- Reads all words from a document

  Put them in a second set

- Print all the words in the second set that are not in the dictionary set.
- Potential misspellings

SpellCheck.java

```
1  import java.util.HashSet;
2  import java.util.Scanner;
3  import java.util.Set;
4  import java.io.File;
5  import java.io.FileNotFoundException;
6
7  /**
8     This program checks which words in a file are not present in a dictionary.
9  */
10 public class SpellCheck
11 {
12    public static void main(String[] args)
13       throws FileNotFoundException
14    {
15       // Read the dictionary and the document
16
17       Set<String> dictionaryWords = readWords("words.txt");
18       Set<String> documentWords = readWords("alice30.txt");
19
20       // Print all words that are in the document but not the dictionary
21
22       for (String word : documentWords)
23       {
24          if (!dictionaryWords.contains(word))
25          {
26             System.out.println(word);
27          }
28       }
29    }
30
31    /**
32       Reads all words from a file.
33       @param filename the name of the file
34       @return a set with all lowercased words in the file. Here, a
35       word is a sequence of upper- and lowercase letters.
```

Program Run:

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
```

Self Check 15.10

Arrays and lists remember the order in which you added elements; sets do not. Why would you want to use a set instead of an array or list?

Answer: Adding and removing elements as well as testing for membership is more efficient with sets.

Self Check 15.11

Why are set iterators different from list iterators?

Answer: Sets do not have an ordering, so it doesn't make sense to add an element at a particular iterator position, or to traverse a set backward.

Self Check 15.12

What is wrong with the following test to check whether the `Set<String>s` contains the elements `"Tom"`, `"Diana"`, and `"Harry"`?

```
if (s.toString().equals("[Tom, Diana, Harry]")) . . .
```

Answer: You do not know in which order the set keeps the elements.

Self Check 15.13

How can you correctly implement the test of Self Check 12?

Answer: Here is one possibility:

```
if (s.size() == 3 && s.contains("Tom")
     && s.contains("Diana")
     && s.contains("Harry"))
   . . .
```

Self Check 15.14

Write a loop that prints all elements that are in both `Set<String>s` and `Set<String>t`.

Answer:

```
for (String str : s)
{
   if (t.contains(str))
   {
      System.out.println(str);
   }
}
```

Self Check 15.15

Suppose you changed line 40 of the `SpellCheck` program to use a `TreeSet` instead of a `HashSet`. How would the output change?

Answer: The words would be listed in sorted order.

Maps

- A map allows you to associate elements from a **key set** with elements from a **value collection**.
- Use a map when you want to look up objects by using a key.

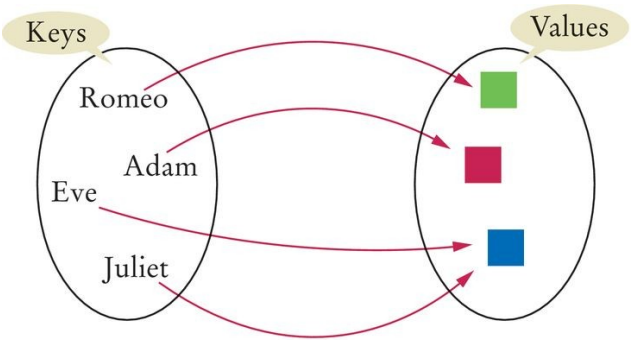

Figure 10 A Map

- Two implementations of the `Map` interface:

  `HashMap`

  `TreeMap`

- Store the reference to the map object in a `Map` reference:

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

Maps

- Use the `put` method to add an association:

```
favoriteColors.put("Juliet", Color.RED);
```

- You can change the value of an existing association by calling `put` again:

```
favoriteColors.put("Juliet", Color.BLUE);
```

- The `get` method returns the value associated with a key:

```
Color julietsFavoriteColor = favoriteColors.get("Juliet");
```

- If you ask for a key that isn't associated with any values, the `get` method returns `null`.
- To remove an association, call the `remove` method with the key:

```
favoriteColors.remove("Juliet");
```

Working with Maps

Table 5 Working with Maps

`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<>();`	Use a `HashMap` if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.
`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

`for (String key : scores.keySet())`
`{`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

Map

- Sometimes you want to enumerate all keys in a map.
- The `keySet` method yields the set of keys.
- Ask the key set for an iterator and get all keys.
- For each key, you can find the associated value with the `get` method.
- To print all key/value pairs in a map `m`:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
    Color value = m.get(key);
    System.out.println(key + "->" + value);
}
```

MapDemo.java

```
1  import java.awt.Color;
2  import java.util.HashMap;
3  import java.util.Map;
4  import java.util.Set;
5
6  /**
7     This program demonstrates a map that maps names to colors.
8  */
9  public class MapDemo
10 {
11    public static void main(String[] args)
12    {
13       Map<String, Color> favoriteColors = new HashMap<>();
14       favoriteColors.put("Juliet", Color.BLUE);
15       favoriteColors.put("Romeo", Color.GREEN);
16       favoriteColors.put("Adam", Color.RED);
17       favoriteColors.put("Eve", Color.BLUE);
18
19       // Print all keys and values in the map
20
21       Set<String> keySet = favoriteColors.keySet();
22       for (String key : keySet)
23       {
24          Color value = favoriteColors.get(key);
25          System.out.println(key + " : " + value);
26       }
27    }
28 }
```

Program Run:

```
Juliet : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```

Self Check 15.16

What is the difference between a set and a map?

Answer: A set stores elements. A map stores associations between keys and values.

Self Check 15.19

Suppose you want to track how many times each word occurs in a document. Declare a suitable map variable.

Answer: `Map<String, Integer> wordFrequency;`

Self Check 15.20

What is a `Map<String, HashSet<String>>`? Give a possible use for such a structure.

Answer: It associates strings with sets of strings. One application would be a thesaurus that lists synonyms for a given word. For example, the key `"improve"` might have as its value the set `["ameliorate", "better", "enhance", "enrich", "perfect", "refine"]`.

Java 8 Note 15.1

- Updating Map Entries used to be tedious

```
Integer count = frequencies.get(word); // Get the old frequency count
// If there was none, put 1; otherwise, increment the count
if (count == null) { count = 1; }
else { count = count + 1; }
frequencies.put(word, count);
```

- New `merge` method in `Map` interface
- Specify:

  Key

  Value to be used if key not yet present

  Function to compute the updated value if key is present

- Replace code above with single line using lambda expression:

```
frequencies.merge(word, 1, (oldValue, value) -> oldValue + value);
```

```
Map<String, Integer> frequencies = new HashMap<>();
frequencies.put("Car", 1);
frequencies.put("Cat", 1);

frequencies.merge("Cat", 1, (oldValue, value) -> oldValue + value);

count = frequencies.get("Cat");
System.out.println(count);
```

Choosing a Collection

1. Determine how you access the values.
2. Determine the element types or key/value types.
3. Determine whether element or key order matters.
4. For a collection, determine which operations must be efficient.
5. For hash sets and maps, decide whether you need to implement the hashCode and equals methods.
6. If you use a tree, decide whether to supply a comparator.