Collections Framework – Part 3

Chapter 9, Core Java Volume I

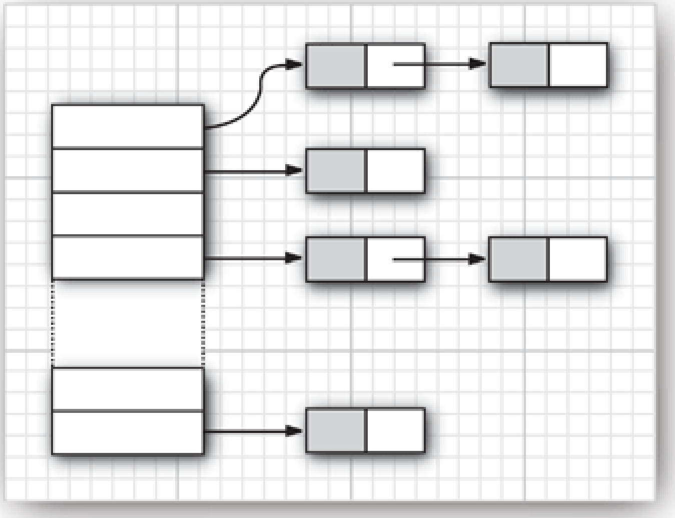
Chapter 16, Java: How to Program, 10th Ed. (Deitels)

Contents

- Hash Sets
- Example: HashSet
- TreeSet
- Maps
- Example: Map
- Updating Map Entries
- Example: Memoization
- Map Views
- Algorithms
- Sorting
- Shuffling

Hash Sets

- If you don't care about element ordering, you can use more efficient collections.
- Sets can add, remove, and find elements quickly.
- Hash set uses hash codes to group elements into buckets:
- Elements are visited in seemingly random order.



Example: HashSet (Listing 9.2)

```
public class SetTest
{
   public static void main(String[] args)
   {
      Set<String> words = new HashSet<>();
      long totalTime = 0;
      long callTime;
      try (Scanner in = new Scanner(System.in))
      {
         while (in.hasNext())
         {
            String word = in.next();
            callTime = System.currentTimeMillis();
            words.add(word);
            callTime = System.currentTimeMillis() - callTime;
            totalTime += callTime;
         }
      }
```

```
      Iterator<String> iter = words.iterator();
      for (int i = 1; i <= 20 && iter.hasNext(); i++)
         System.out.println(iter.next());
      System.out.println(". . .");
      System.out.println(words.size() + " distinct words. " +
            totalTime + " milliseconds.");
   }
}
```

TreeSet

- Tree sets visit elements in sorted order.
- In practice, a bit slower than hash sets.

Document	Total Words	Distinct Words	HashSet	TreeSet
Alice in the Wonder Land	28195	5909	5 sec	7 sec
The Count of Monte Cristo	466300	37545	74 sec	98 sec

- Tree set needs total ordering—not always easy to find.
  - In a total ordering, two elements compare identically only when they are equal.
  - What would be a total ordering for Rectangle?
- Use tree sets when your elements are comparable and you need traversal in sorted order.

Maps

- A map stores key/value associations.
- HashMap hashes the keys, TreeMap organizes them in sorted order.
- Add an association to a map:

```
Map<String, Employee> staff = new HashMap<>(); // HashMap implements Map
Employee harry = new Employee(. . .);
staff.put("987-98-9996", harry);
```

- Retrieve a value with a given key:

```
String id = "987-98-9996";
Employee e = staff.get(id); // gets harry
```

- The get method returns null if the key is absent. Better approach:

```
Map<String, Integer> scores = . . .;
int score = scores.getOrDefault(id, 0); // Gets 0 if the id is not present
```

- map.remove(key) removes a key.
  - returns its value if it contains the key, otherwise returns null
- Easiest way to iterate over a map:

```
scores.forEach((k, v) -> System.out.println("key=" + k + ", value=" + v));
```

Example: Map(Listing 9.6)

```
public class MapTest
{
   public static void main(String[] args)
   {
      Map<String, Employee> staff = new HashMap<>();
      staff.put("144-25-5464", new Employee("Amy Lee"));
      staff.put("567-24-2546", new Employee("Harry Hacker"));
      staff.put("157-62-7935", new Employee("Gary Cooper"));
      staff.put("456-62-5527", new Employee("Francesca Cruz"));
      System.out.println(staff);                          // print all entries
      staff.remove("567-24-2546");                        // remove an entry
      staff.put("456-62-5527", new Employee("Francesca Miller")); // replace an entry
      System.out.println(staff.get("157-62-7935"));        // look up a value
      staff.forEach((k, v) ->                             // iterate through all entries
         System.out.println("key=" + k + ", value=" + v));
   }
}
```

Updating Map Entries

- Updating a map entry is tricky because the first time is special.
- Consider updating a word count:

```
counts.put(word, counts.get(word) + 1);
```

- What if word wasn't present?

```
counts.put(word, counts.getOrDefault(word, 0) + 1);
```

- Another approach:

```
counts.putIfAbsent(word, 0);
counts.put(word, counts.get(word) + 1); // Now we know that get will succeed
```

- Even better:

```
counts.merge(word, 1, Integer::sum);
```

  - If word wasn't present, put 1. Otherwise, put the sum of 1 and the previous value.

Example: Memoization

```
import java.math.BigInteger;
import java.util.HashMap;
import java.util.Map;
public class Fibonacci
{
    private Map<Integer, BigInteger>
            memoizeHashMap = new HashMap<>();
    { // initialization block
        memoizeHashMap.put(0, BigInteger.ZERO);
        memoizeHashMap.put(1, BigInteger.ONE);
        memoizeHashMap.put(2, BigInteger.ONE);
    }
```

```
    private BigInteger fibonacci(int n)
    {
        if (memoizeHashMap.containsKey(n)) {
            return memoizeHashMap.get(n);
        } else {
            BigInteger result = fibonacci(n - 1).add(fibonacci(n - 2));
            memoizeHashMap.put(n, result);
            return result;
        }
    }
    public static void main(String[] args)
    {
        Fibonacci fibonacci = new Fibonacci();
        for (int i = 0; i < 100; i++) {
            System.out.println(fibonacci.fibonacci(i));
        }
    }
}
```

Map Views

- In the Java collections framework, a map isn't a collection.
- Can get collections of keys, values, and key/value pairs:

```
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K, V>> entrySet()
```

- To visit all keys and values, can use:

```
Set<String> keys = staff.keySet();
for (String k : keys)
     do something with k and  staff.get(k)
```

- More efficiently:

```
for (Map.Entry<String, Employee> entry : staff.entrySet())
{
   String k = entry.getKey();
   Employee v = entry.getValue();
   do something with k, v
}
```

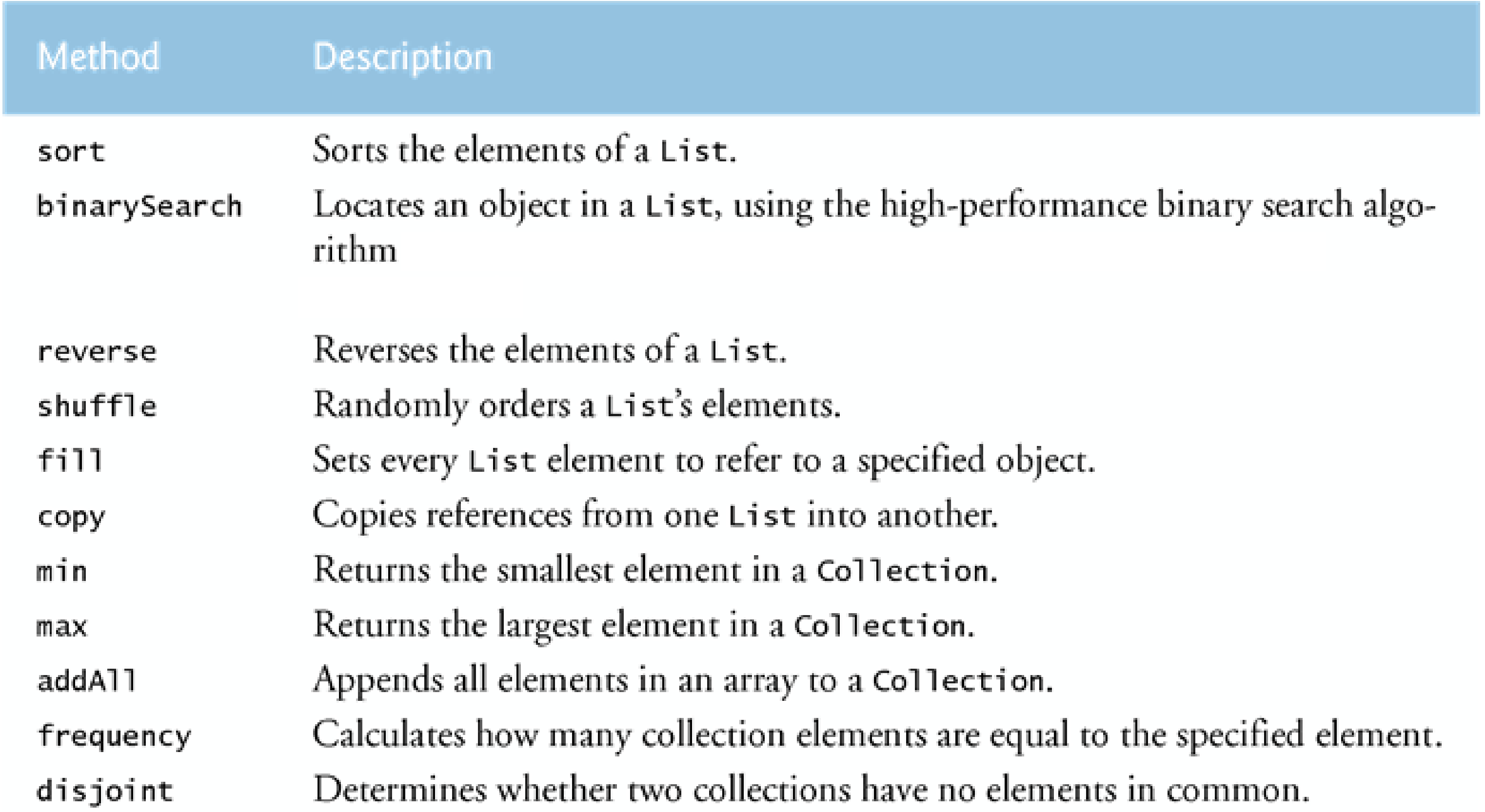
- Efficient and elegant:

```
staff.forEach((k, v) ->  do something with k, v);
```

- Calling remove on the key set removes the key and associated value from the map.

Algorithms

- Class `Collections` provides several high-performance algorithms for manipulating collection elements.
- The algorithms are implemented as static methods.



Method	Description
`sort`	Sorts the elements of a `List`.
`binarySearch`	Locates an object in a `List`, using the high-performance binary search algorithm
`reverse`	Reverses the elements of a `List`.
`shuffle`	Randomly orders a `List`'s elements.
`fill`	Sets every `List` element to refer to a specified object.
`copy`	Copies references from one `List` into another.
`min`	Returns the smallest element in a `Collection`.
`max`	Returns the largest element in a `Collection`.
`addAll`	Appends all elements in an array to a `Collection`.
`frequency`	Calculates how many collection elements are equal to the specified element.
`disjoint`	Determines whether two collections have no elements in common.

Algorithms

- Generic Collection interface have a great advantage – you only need to implement your algorithms once.
- Example: finding the maximum element

for arrays

```
T largest = a[0];
for(int i=1; i < a.length; i++)
   if(largest.compareTo(a[i]) < 0)
       largest = a[i];
```

for array lists

```
T largest = v.get(0);
for(int i=1; i < v.size(); i++)
   if(largest.compareTo(v.get(i) < 0)
       largest = v.get(i);
```

for hash sets

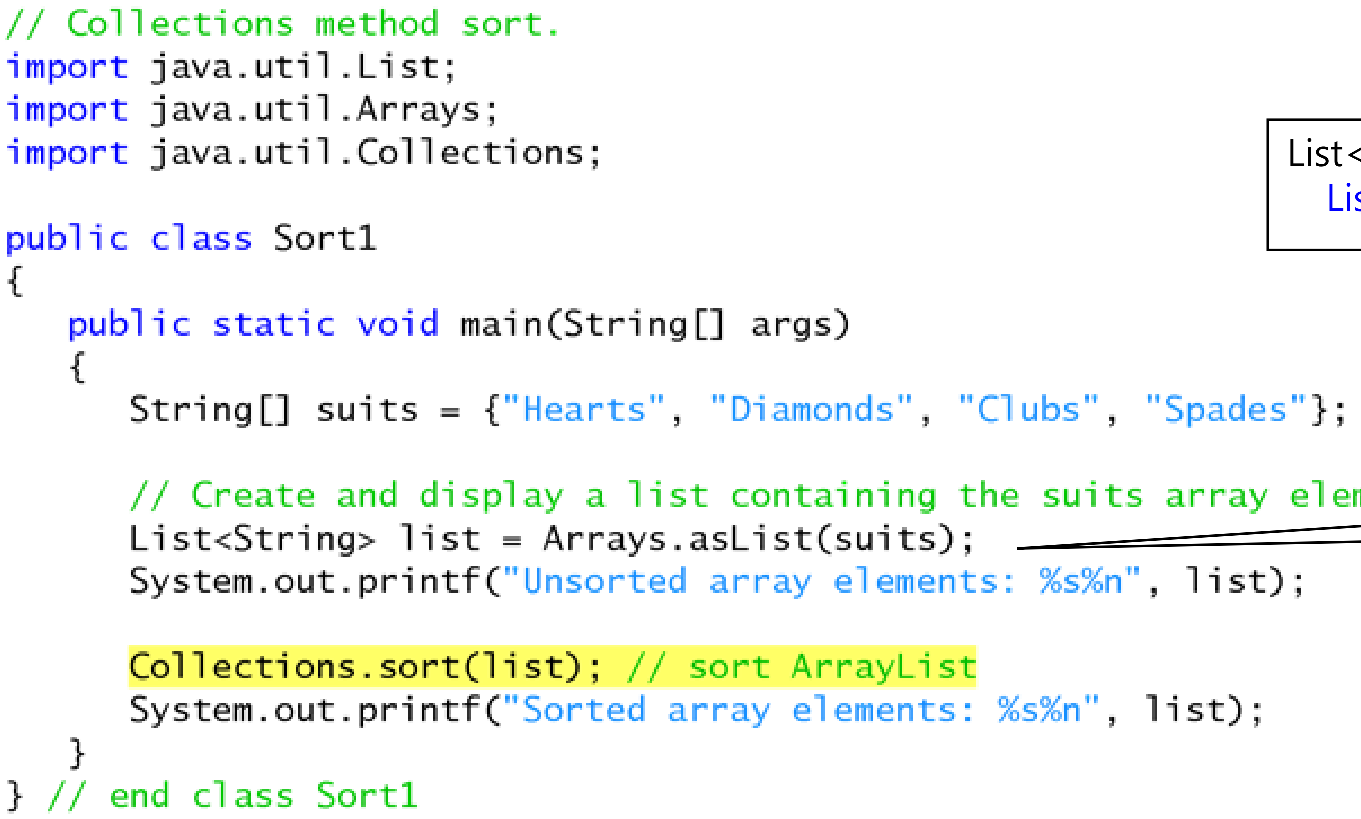
```
???
```

for all classes that implements Collection Interface

```
public static <T extends Comparable> T max(Collections<T> c)
{
  Iterator<T> iter = c.iterator();
  T largest = iter.next();
  while(iter.hasNext())
  {
      T next = iter.next();
       if(largest.compareTo(next) < 0)
          largest = next;
  }
  return largest;
}
```

Sorting

- *Collections.sort* sorts the elements of a List
- The elements must implement the *Comparable* interface.



```
// Collections method sort.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort1
{
   public static void main(String[] args)
   {
      String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};

      // Create and display a list containing the suits array elements
      List<String> list = Arrays.asList(suits);
      System.out.printf("Unsorted array elements: %s%n", list);

      Collections.sort(list); // sort ArrayList
      System.out.printf("Sorted array elements: %s%n", list);
   }
} // end class Sort1
```

```
List<String> list =
   List.of("Heart", "Diamond", "Clubs", "Spades");
```

unmodifiable

mutable, but not resizable

Sorting

- Using sort method in List interface and comparators

```
List<Employee> staff = LinkedList<>();
//filling collection
staff.sort(Comparator.reverseOrder);


stsff.sort(Comparator.comparingDouble(Employee::getSalary));
```

Shuffling

```
public class ShuffleTest
{
   public static void main(String[] args)
   {
      List<Integer> numbers = new ArrayList<>();
      for (int i = 1; i <= 49; i++)
         numbers.add(i);   // autoboxing
      Collections.shuffle(numbers);
      List<Integer> winningCombination = numbers.subList(0, 6);
      Collections.sort(winningCombination);
      System.out.println(winningCombination);
   }
}
```