Chapter 6
Maps

Advanced Topics in Java

Reference:

A Programmer's Guide to Java SE 8 Oracle Certified Professional (OCP),
Khalid A. Mughal,
Addison-Wesley Professional, *To be published in 2019*.

Khalid Azim Mughal
Associate Professor Emeritus
Department of Informatics
University of Bergen, Norway.
khalid.mughal@uib.no
http://www.ii.uib.no/~khalid

Version date: 2018-11-06

Overview

•	• Maps • Map Implementations • The Collections Class • The Arrays Class

Maps

- A *map* defines *mappings* from keys to values. The <*key*, *value*> pair is called a *mapping*, also referred to as an *entry*. A map does not allow duplicate keys, in other words, the keys are unique. Each key maps to one value at the most, implementing what is called a *single-valued map*. Thus, there is a *many-to-one* relation between keys and values. For example, in a student-grade map, a grade (value) can be awarded to many students (keys), but each student has only one grade. Changing the value that is associated with a key, results in the old entry being removed and a new entry being inserted.
- Both the keys and the values must be objects, with primitive values being wrapped in their respective primitive wrapper objects when they are put in a map.

The `Map<K,V>` Interface

- A map is not a collection and the `Map` interface does not extend the `Collection` interface. However, the mappings can be viewed as a collection in various ways: a key set, a value collection, or an entry set. A key set view or an entry set view can be iterated over to retrieve the corresponding values from the underlying map.
- The `Map` interface specifies some optional methods. Implementations should throw an `UnsupportedOperationException` if they do not support such an operation. The implementations of maps from the `java.util` package support all the optional operations of the `Map` interface (see Table 6.2 and Figure 6.3).
- The `Map<K,V>` interface and its subinterfaces `SortedMap<K,V>` and `NavigableMap<K,V>` provide a versatile set of methods to implementation a wide range of operations on maps. Several examples in this section and subsequent sections illustrate many of the methods specified in these interfaces.

Basic Key-Based Operations

- These operations constitute the basic functionality provided by a map.

`Object put(K key, V value)` Optional

`default V putIfAbsent(K key, V value)`

The `put()` method inserts the <`key`, `value`> entry into the map. It returns the old *value* previously associated with the specified `key`, if any. Otherwise, it returns the `null` value.

The default method `putIfAbsent()` associates the `key` with the given `value` and returns `null` if the specified `key` is *not* already associated with a non-`null` value, otherwise it returns the currently associated value.

`Object get(Object key)`

`default V getOrDefault(Object key, V defaultValue)`

The `get()` method returns the value to which the specified `key` is mapped, or `null` if no entry is found.

The default method `getOrDefault()` returns the value to which the specified `key` is mapped, or the specified `defaultValue` if this map contains no entry for the `key`.

`Object remove(Object key)` Optional

`default boolean remove(Object key, Object value)`

The `remove()` method deletes the entry for the specified `key`. It returns the *value* previously associated with the specified `key`, if any. Otherwise, it returns the `null` value.

The default method `remove()` removes the entry for the specified `key` and returns `true` only if the specified `key` is currently mapped to the specified `value`.

`default V replace(K key, V value)`

`default boolean replace(K key, V oldValue, V newValue)`

In the first `replace()` method, the value associated with the `key` is replaced with the specified `value` only if the `key` is already mapped to some value. It returns the *previous* value associated with the specified `key`, or `null` if there was no entry found for the `key`.

In the second `replace()` method, the value associated with the `key` is replaced with the specified `newValue` only if the `key` is currently mapped to the specified `oldValue`. It returns `true` if the value in the entry for the `key` was replaced with the `newValue`.

`boolean containsKey(Object key)`

`boolean containsValue(Object value)`

The `containsKey()` method returns `true` if the specified `key` is mapped to a value in the map.

The `containsValue()` method returns `true` if there exists one or more keys that are mapped to the specified `value`.

Bulk Operations

- Bulk operations can be performed on an entire map.

`int size()`
`boolean isEmpty()`
Return the number of entries (i.e., number of unique keys in the map) and whether the map is empty or not, respectively.

`void clear()` Optional
`void putAll(Map<? extends K, ? extends V> map)` Optional
The first method deletes all entries from the current map.

The second method copies all entries from the specified `map` to the current map. If a key from the specified map is already in the current map, its associated value in the current map is replaced with the associated value from the specified map.

`default void forEach(BiConsumer<? super K,? super V> action)`
The specified `action` is performed for each entry in this map until all entries have been processed or the `action` throws an exception. Iteration is usually performed in entry set iteration order. The functional interface `BiConsumer<T,U>` is covered in §5.7, p. 304. This method is used in Example 6.1, p. 19.

`default void replaceAll(BiFunction<? super K,? super V,? extends V> func)`
The value of each entry is replaced with the result of invoking the given binary function on the entry until all entries have been processed or the function throws an exception. The functional interface `BiFunction<T,U,R>` is covered in §5.9, p. 311, where this method is illustrated.

Collection Views

- Views allow information in a map to be represented as collections. Elements can be removed from a map via a view, but cannot be added. An iterator over a view will throw an exception if the underlying map is modified concurrently. Example 6.1, p. 19, illustrates collection views.

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

These methods provide different views of a map. Changes in the map are reflected in the view, and vice versa. These methods return a set view of keys, a collection view of values, and a set view of *<key, value>* entries, respectively. Note that the `Collection` returned by the `values()` method is not a `Set`, as several keys can map to the same value, that is, duplicate values can be included in the returned collection. Each *<key, value>* in the entry set view is represented by an object implementing the nested `Map.Entry` interface. An entry in the entry set view can be manipulated by methods defined in this interface, which are self-explanatory:

```
interface Entry<K, V> {  // Nested interface in the Map interface.
    K getKey();
    V getValue();
    V setValue(V value);
}
```

Map Implementations

The `HashMap<K,V>`, `LinkedHashMap<K,V>`, and `Hashtable<K,V>` Classes

- Figure 6.3 shows four implementations of the `Map` interface in the `java.util` package: `HashMap`, `LinkedHashMap`, `TreeMap`, and `Hashtable`.
- The classes `HashMap` and `Hashtable` implement unordered maps. The class `LinkedHashMap` implements ordered maps, which are discussed below. The class `TreeMap` implements sorted maps (§, p. 125).

Table 6.1 Allowing `null` key and `null` values in Maps

Map	`null` as key	`null` as value
`HashMap`	*Allowed*	*Allowed*
`LinkedHashMap`	*Allowed*	*Allowed*
`Hashtable`	*Not Allowed*	*Not Allowed*
`TreeMap`	*Not Allowed*	*Allowed*

- While the `HashMap` class is not thread-safe and permits a `null` key and `null` values (analogous to the `LinkedHashMap` class), the `Hashtable` class is thread-safe and permits only non-`null` keys and values (see Table 6.1). The thread-safety provided by the `Hashtable` class comes with a performance penalty. Thread-safe use of maps is also provided by the methods in the `Collections` class (§, p. 21). Like the `Vector` class, the `Hashtable` class is also a legacy class that has been retrofitted to implement the `Map` interface.
- These map implementations are based on a hashing algorithm. Operations on a map thus rely on the `hashCode()` and `equals()` methods of the key objects (§6.1, p. 355).
- The `LinkedHashMap` implementation is a subclass of the `HashMap` class. The relationship between the map classes `LinkedHashMap` and `HashMap` is analogous to the relationship between their counterpart set classes `LinkedHashSet` and `HashSet`. Elements of a `HashMap` (and a `HashSet`) are unordered. The elements of a `LinkedHashMap` (and a

`LinkedHashSet`) are ordered. By default, the entries of a `LinkedHashMap` are in *key insertion order*, that is, the order in which the keys are inserted in the map. This order does not change if a key is re-inserted, because no new entry is created if the key's entry already exists. The elements in a `LinkedHashSet` are in (element) insertion order. However, a `LinkedHashMap` can also maintain its elements in (element) *access order*, that is, the order in which its entries are accessed, from least-recently accessed to most-recently accessed entries. This *ordering mode* can be specified in one of the constructors of the `LinkedHashMap` class.

- Both the `HashMap` and the `LinkedHashMap` classes provide comparable performance, but the `HashMap` class is the natural choice if ordering is not an issue. Operations such as adding, removing, or finding an entry based on a key are in constant time, as these hash the key. Operations such as finding the entry with a specific *value* are in linear time, as these involve searching through the entries.
- Adding, removing, and finding entries in a `LinkedHashMap` can be slightly slower than in a `HashMap`, as an ordered doubly-linked list has to be maintained. Iteration over a map is through one of its collection-views. For an underlying `LinkedHashMap`, the iteration time is proportional to the size of the map—regardless of its capacity. However, for an underlying `HashMap`, it is proportional to the capacity of the map.
- The concrete map implementations override the `toString()` method. The standard textual representation generated by the `toString()` method for a map is

  $\{key_1=value_1,\ key_2=value_2,\ \ldots,\ key_n=value_n\}$

- where each key_i and each $value_i$ is the textual representation generated by the `toString()` method of the individual key and value objects in the map, respectively.
- As was the case with collections, implementation classes provide a standard constructor that creates a new empty map, and a constructor that creates a new map based on an existing one. Additional constructors create empty maps with given initial capacities and load factors. The `HashMap` class provides the following constructors:

```
HashMap()
HashMap(int initialCapacity)
HashMap(int initialCapacity, float loadFactor)
```
Constructs an empty `HashMap`, using either specified or default initial capacity and load factor.

```
HashMap(Map<? extends K,? extends V> otherMap)
```
Constructs a new map containing the elements in the specified map.

- The `LinkedHashMap` and `Hashtable` classes have constructors analogous to the four constructors for the `HashMap` class. In addition, the `LinkedHashMap` class provides a constructor where the ordering mode can also be specified:

```
LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)
```
Constructs a new, empty `LinkedHashMap` with the specified initial capacity, the specified load factor, and the specified ordering mode. The ordering mode is `true` for *access order* and `false` for *key insertion order*.

- Example 6.1 prints a textual histogram for the frequency of weight measurements in a weight group, where a weight group is defined as an interval of five units. The weight measurements are supplied as program arguments. A `HashMap<Integer, Integer>` is used, where the key is the weight group and the value is the frequency. The example illustrates many key-based operations on maps, the creation of key views, and iteration over a map.
- We have intentionally used a `HashMap<K,V>`. We leave it as an exercise for the reader to use a `TreeMap<K,V>` which simplifies the solution further, when the entries are maintained in key-sorted order.
- The program proceeds as follows:
  - An empty `HashMap<Integer, Integer>` is created at (1), where the key is the weight group and the associated value is the frequency.
  - A `for(:)` loop is used at (2) to read the weights specified as program arguments, converting each weight to its corresponding weight group and updating the frequency of the weight group.
  - Each program argument is parsed to a `double` value at (3), which is then used to determine the correct weight group at (4). The call to the `merge()` method at (5) updates the frequency map:

```
// With method reference:
groupFreqMap.merge(weightGroup, 1, Integer::sum);                    // (5)

// With lambda expression:
```

```
groupFreqMap.merge(weightGroup, 1,
   (oldVal, givenVal) -> Integer.sum(oldVal, givenVal));
```

- If the weight group is not in the map, its frequency is set to `1`, otherwise the method reference `Integer::sum` increments the frequency by the given value `1`. In both cases an appropriate entry for the weight group is put in the frequency map. Note that the arguments passed to the method reference and the lambda expression: `oldVal` is the current value associated with the key and `givenVal` is the specified value, `1`, in the argument to the `merge()` method.
- Generic types guarantee that the keys and the values in the map are of the correct type, and autoboxing/unboxing of primitive values guarantees the correct type of an operand in an expression.
- Some other strategies to update the frequency map are outlined here, but none is more elegant than the one with the `merge()` method.
- The straight forward solution below requires an explicit `null` check to determine whether the value returned by the `get()` method is `null`, or risk a `NullPointerException` at runtime when incrementing the `null` value at (7).

```
Integer frequency = groupFreqMap.get(weightGroup);                  // (5)
if (frequency == null) frequency = 0;                               // (6)
groupFreqMap.put(weightGroup, ++frequency);                         // (7)
```

- Below, the `getOrDefault()` method never returns a `null` value, since it returns the associated frequency value or the specified default value `0` depending on whether the weight group is found or not found is in the map, respectively. No explicit `null`

check is necessary.

```
Integer frequency = groupFreqMap.getOrDefault(weightGroup, 0);       // (5)
groupFreqMap.put(weightGroup, ++frequency);                          // (6)
```

- The `putIfAbsent()` method puts an entry with frequency `0` for the weight group if it is not found, so that the `get()` method will always return a non-`null` value.

```
groupFreqMap.putIfAbsent(weightGroup, 0);                            // (5)
Integer frequency = groupFreqMap.get(weightGroup);                   // (6)
groupFreqMap.put(weightGroup, ++frequency);                          // (7)
```

- The `compute()` method always updates the value associated with the key if the binary function returns a non-`null` value, which is always the case below. The `null` check is now done in the lambda expression.

```
groupFreqMap.compute(weightGroup,                                    // (5)
      (k, v) -> v == null ? 1 : v + 1);
```

- The program creates a sorted set of keys (which are weight groups) from the `groupFreqMap` at (6). The `keySet()` method returns a set view of keys, which is passed as argument to a `TreeSet` to create a sorted set—in this case, the natural ordering of `Integer`s is used.

```
TreeSet<Integer> sortedKeySet = new TreeSet<>(groupFreqMap.keySet()); // (6)
```

- The histogram is printed by implicitly iterating over the sorted key set at (7). Since a sorted set is `Iterable`, we can use the `forEach()` method that takes a consumer to print the histogram.

```
sortedKeySet.forEach(key ->                                          // (7)
    System.out.printf("%5s: %s%n", key,
        String.join("", Collections.nCopies(groupFreqMap.get(key), "*")))
```

- For each key, the corresponding value (i.e., the frequency) is retrieved and converted to a string with corresponding number of "*". The method `Collections.nCopies()` creates a list with equal number of elements as its first argument and where each element is the same as the second argument. The elements of this list are joined to create a string by the `String.join()` method with the first argument specifying the delimiter to use between the elements, which in this case is the empty string.
- Alternately, a `for(:)` loop can be used to explicitly iterate over the sorted set view of the frequency map.

Example 6.1 Using Maps

```
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.TreeSet;

public class Histogram {
  public static void main(String[] args) {
    System.out.println("Data: " + Arrays.toString(args));

    // Create a map to store the frequency for each group.
    Map<Integer, Integer> groupFreqMap = new HashMap<>();                    // (1)

    // Determine the frequencies:
    for (String argument : args) {                                           // (2)
      double weight = Double.parseDouble(argument);                          // (3)
      int weightGroup = (int) Math.round(weight/5.0)*5;                      // (4)
      groupFreqMap.merge(weightGroup, 1, Integer::sum);                      // (5)
    }
    System.out.println("Frequencies: " + groupFreqMap);

    // Create sorted key set.
    TreeSet<Integer> sortedKeySet = new TreeSet<>(groupFreqMap.keySet());  // (6)
```

```
        System.out.println("Histogram:");
        // Implicit iteration over the key set.
        sortedKeySet.forEach(key ->                                          // (7)
            System.out.printf("%5s: %s%n", key,
                String.join("", Collections.nCopies(groupFreqMap.get(key), "*")))
        );
    }
}
```

- Running the program with the following arguments:

```
>java Histogram 74 75 93 75 93 82 61 92 10 185
```

- gives the following output:

```
Data: [74, 75, 93, 75, 93, 82, 61, 92, 10, 185]
Frequencies: {80=1, 185=1, 10=1, 90=1, 75=3, 60=1, 95=2}
Histogram:
   10: *
   60: *
   75: ***
   80: *
   90: *
   95: **
  185: *
```

The `Collections` Class

- The Java Collections Framework also contains two classes, `Collections` and `Arrays`, that provide various operations on collections and arrays, such as algorithms for sorting and searching, or creating customized collections. Practically any operation on a collection can be done using the methods provided by this framework.
- The methods provided are all `public` and `static`, therefore these two keywords will be omitted in their method header declarations in this section.
- The methods also throw a `NullPointerException` if the specified collection or array references passed to them are `null`.

Ordering Elements in Lists

- In order to sort the elements of a collection by their *natural ordering* or by a *total ordering*, the elements must implement the `Comparable<E>` (p. 375) or a `Comparator<E>` (p. 382) must be provided, respectively.
- The `Collections` class provides two generic static methods for sorting lists.

```
<E extends Comparable<? super E>> void sort(List<E> list)
<E> void sort(List<E> list, Comparator<? super E> comparator)
```

The first method sorts the elements in the list according to their natural ordering. The second method does the sorting according to the total ordering defined by the comparator.

In addition, all elements in the list must be *mutually comparable*: the method call `e1.compareTo(e2)` (or `e1.compare(e2)` in case of the comparator) must not throw a `ClassCastException` for any elements `e1` and `e2` in the list. In other words, it should be possible to compare any two elements in the list. Note that the second method does not require that the type parameter `E` is `Comparable`.

If the specified comparator is `null` then all elements in this list must implement the `Comparable<E>` interface and the natural ordering for the elements is used.

```
<E> Comparator<E> reverseOrder()
<E> Comparator<E> reverseOrder(Comparator<E> comparator)
```

The first method returns a comparator that enforces the reverse of the natural ordering. The second one reverses the total ordering defined by the comparator. Both are useful for maintaining objects in reverse-natural or reverse-total ordering in sorted collections and arrays.

- The `List<E>` interface also defines an abstract method for sorting lists, with semantics equal to its namesake in the `Collections` class.

```
// Defined in the List <E> interface.
void sort(Comparator<? super E> comparator)
```

This list is sorted according to the total ordering defined by the specified comparator. If the specified comparator is `null` then all elements in this list must implement the `Comparable<E>` interface and the natural ordering for the elements is used.

- This code shows how a list of strings is sorted according to different criteria.We have used the `sort()` method from the `List<E>` interface and from the `Collections` class.

```
List<String> strList = new ArrayList<>();
Collections.addAll(strList, "biggest", "big", "bigger", "Bigfoot");

strList.sort(null);                                    // Natural order
strList.sort(Comparator.comparing(String::length));    // length order
strList.sort(Collections.reverseOrder());             // Reverse natural order
```

```
Collections.sort(strList, String.CASE_INSENSITIVE_ORDER);// Case insensitive
order
Collections.sort(strList,                          // Reverse case insensitive order
                 Collections.reverseOrder(String.CASE_INSENSITIVE_ORDER));
```

- The output below shows the list before sorting, followed by the results from the calls to the `sort()` methods above, respectively:

```
Before sorting:                          [biggest, big, bigger, Bigfoot]
After sorting in natural order:          [Bigfoot, big, bigger, biggest]
After sorting in length order:           [big, bigger, Bigfoot, biggest]
After sorting in reverse natural order:  [biggest, bigger, big, Bigfoot]
After sorting in insensitive order:      [big, Bigfoot, bigger, biggest]
After sorting in reverse insensitive order: [biggest, bigger, Bigfoot, big]
```

- It is important to note that the element type of the list must implement the `Comparable<E>` interface, otherwise the compiler will report an error. The following code shows that a list of `StringBuilders` cannot be sorted, because the class `StringBuilder` does not implement the `Comparable<E>` interface.

```
List<StringBuilder> sbList = new ArrayList<>();
Collections.addAll(sbList, new StringBuilder("smallest"),
                   new StringBuilder("small"), new StringBuilder("smaller"));
Collections.sort(sbList);                              // Compile-time error!
```

- Below is an example of a list whose elements are not mutually comparable. Raw

types are used intentionally to create such a list. Predictably, the `sort()` method throws an exception, because the primitive wrapper classes do not permit interclass comparison.

```
List freakList = new ArrayList();                    // Raw types.
Collections.addAll(freakList, 23, 3.14, 10L);
freakList.sort(null);                                // ClassCastException
```

- The comparator returned by the `reverseOrder()` method can be used with *sorted* collections. The elements in the following sorted set would be maintained in descending order:

```
Set<Integer> intSet = new TreeSet<>(Collections.reverseOrder());
Collections.addAll(intSet, 9, 11, -4, 1);
System.out.println(intSet);          // [11, 9, 1, -4]
```

- The following utility methods apply to *any* list, regardless of whether the elements are `Comparable` or not:

`void reverse(List<?> list)`
Reverses the order of the elements in the list.

`void rotate(List<?> list, int distance)`
Rotates the elements towards the end of the list by the specified distance. A negative value for the `distance` will rotate toward the start of the list.

`void shuffle(List<?> list)`
`void shuffle(List<?> list, Random rnd)`
Randomly permutes the list, that is, *shuffles* the elements.

`void swap(List<?> list, int i, int j)`
Swaps the elements at indices `i` and `j`.

- The effect of these utility methods can be limited to a sublist, that is, a segment of the list. The following code illustrates rotation of elements in a list. Note how the rotation in the sublist view is reflected in the original list.

```
// intList refers to the following list:                    [9, 11, -4, 1, 7]
Collections.rotate(intList, 2);       // Two to the right.    [1, 7, 9, 11, -4]
Collections.rotate(intList, -2);      // Two to the left.     [9, 11, -4, 1, 7]
List intSublist = intList.subList(1,4);// Sublist:               [11, -4, 1]
Collections.rotate(intSublist, -1);    // One to the left.         [-4, 1, 11]
                                      // intList is now:      [9, -4, 1, 11, 7]
```

Searching in Collections

- The `Collections` class provides two static methods for finding elements in *sorted* lists.

```
<E> int binarySearch(List<? extends Comparable<? super E>> list, E key)
<E> int binarySearch(List<? extends E> list, E key,
                     Comparator<? super E> cmp))
```

The methods use binary search to find the index of the `key` element in the specified sorted list. The first method requires that the list is sorted according to natural ordering, whereas the second method requires that it is sorted according to the total ordering dictated by the comparator. The elements in the list and the key must also be *mutually comparable*.

- Successful searches return the index of the key in the list. A non-negative value indicates a successful search. Unsuccessful searches return a negative value given by the formula `-(`*insertion point* `+ 1)`, where *insertion point* is the index where the key would have been, had it been in the list. In the code below, the return value -3 indicates that the key would have been at index 2, had it been in the list.

```
Collections.sort(strList);
// Sorted in natural order: [Bigfoot, big, bigger, biggest]
// Search in natural order:
out.println(Collections.binarySearch(strList, "bigger"));   // Successful:    2
out.println(Collections.binarySearch(strList, "bigfeet"));  // Unsuccessful: -3
out.println(Collections.binarySearch(strList, "bigmouth")); // Unsuccessful: -5
```

- Proper use of the search methods requires that the list is sorted, and the search is performed according to the same sort ordering. Otherwise, the search results are *unpredictable*. The example below shows the results of the search when the list `strList` above was sorted in reverse natural ordering, but was searched assuming natural ordering. Most importantly, the return values reported for unsuccessful searches for the respective keys are incorrect in the list that was sorted in reverse natural ordering.

```
Collections.sort(strList, Collections.reverseOrder());
// Sorted in reverse natural order: [biggest, bigger, big, Bigfoot]
// Searching in natural order:
out.println(Collections.binarySearch(strList, "bigger"));   //  1
out.println(Collections.binarySearch(strList, "bigfeet"));  // -1 (INCORRECT)
out.println(Collections.binarySearch(strList, "bigmouth")); // -5 (INCORRECT)
```

- Searching the list in reverse natural ordering requires that an appropriate comparator is supplied during the search (as during the sorting), resulting in correct results:

```
Collections.sort(strList, Collections.reverseOrder());
// Sorted in reverse natural order: [biggest, bigger, big, Bigfoot]
// Searching in reverse natural order:
out.println(Collections.binarySearch(strList, "bigger",
                                     Collections.reverseOrder())); //  1
out.println(Collections.binarySearch(strList, "bigfeet",
                                     Collections.reverseOrder())); // -3
out.println(Collections.binarySearch(strList, "bigmouth",
```

```
                                    Collections.reverseOrder())); // -1
```

- The following methods search for *sublists*:

```
int indexOfSubList(List<?> source, List<?> target)
int lastIndexOfSubList(List<?> source, List<?> target)
```

These two methods find the first or the last occurrence of the `target` list in the `source` list, respectively. They return the starting position of the `target` list in the `source` list. The methods are applicable to lists of *any* type.

- The following methods find the maximum and minimum elements in a collection:

```
<E extends Object & Comparable<? super E>>
    E max(Collection<? extends E> c)
<E> E max(Collection<? extends E> c, Comparator<? super E> cmp)
<E extends Object & Comparable<? super E>>
    E min(Collection<? extends E> c)
<E> E min(Collection<? extends E> cl, Comparator<? super E> cmp)
```

Find the maximum and minimum elements in a collection. The one-argument methods require that the elements have a natural ordering, i.e., are `Comparable`. The other methods require that the elements have a total ordering enforced by the comparator. Calling any of the methods with an empty collection as parameter results in an `NoSuchElementException`.

The time for the search is proportional to the size of the collection.

These methods are analogous to the methods `first()` and `last()` in the `SortedSet` class, and the methods `firstKey()` and `lastKey()` in the `SortedMap` class.

The `Arrays` Class

- In this section we look at some selected utility methods from the `Arrays` class, in particular, for sorting, searching, and creating list views of arrays.
- Using the overloaded static method `Arrays.stream()` to create streams with arrays as the data source is covered in §7.3, p. 502.

Sorting Arrays

- The `Arrays` class provides enough overloaded versions of the `sort()` method to sort practically any type of array. The discussion on sorting lists (page 22) is also applicable to sorting arrays.

```
void sort(type[] array)
void sort(type[] array, int fromIndex, int toIndex)
```

These methods sort the elements in the array according to their natural ordering. Permitted *type* for elements include `byte`, `char`, `double`, `float`, `int`, `long`, `short` and `Object`. In the case of an array of objects being passed as argument, the *objects* must be mutually comparable according to the natural ordering defined by the `Comparable<E>` interface.

```
<E> void sort(E[] array, Comparator<? super E> cmp)
<E> void sort(E[] array, int fromIndex, int toIndex,
              Comparator<? super E> cmp)
```

The two generic methods above sort the array according to the total ordering dictated by the comparator. In particular, the methods require that the elements are mutually comparable according to this comparator.

The bounds, if specified in the methods above, define a half-open interval. Only elements in this interval are then sorted.

The `Arrays` class also defines analogous methods with the name `parallelSort` for sorting in parallel.

- The experiment from §, p. 22, with a list of strings is now repeated with an array of strings, giving identical results. An array of strings is sorted according to different criteria.

```
String[] strArray =  {"biggest", "big", "bigger", "Bigfoot"};
Arrays.sort(strArray);                                          // Natural order
Arrays.sort(strArray, Comparator.comparing(String::length));   // Length order
Arrays.sort(strArray, Collections.reverseOrder());   // Reverse natural order
Arrays.sort(strArray, String.CASE_INSENSITIVE_ORDER);// Case insensitive order
Arrays.sort(strArray,                          // Reverse case insensitive order
            Collections.reverseOrder(String.CASE_INSENSITIVE_ORDER));
```

- The output below shows the array before sorting, followed by the results from the calls to the `sort()` methods above, respectively:

```
Before sorting:                                 [biggest, big, bigger, Bigfoot]
After sorting in natural order:                 [Bigfoot, big, bigger, biggest]
After sorting in length order:                  [big, bigger, Bigfoot, biggest]
After sorting in reverse natural order:         [biggest, bigger, big, Bigfoot]
After sorting in case insensitive order:        [big, Bigfoot, bigger, biggest]
After sorting in reverse case insensitive order: [biggest, bigger, Bigfoot, big]
```

- The examples below illustrate sorting an array of primitive values (`int`) at (1), an array of type `Object` containing mutually comparable elements (`String`) at (2), and a half-open interval in reverse natural ordering at (3). A `ClassCastException` is thrown

when the elements are not mutually comparable, at (4) and (5).

```
int[] intArray = {5, 3, 7, 1};                  // int
Arrays.sort(intArray);                          // (1) Natural order: [1, 3, 5, 7]

Object[] objArray1 = {"I", "am", "OK"};         // String
Arrays.sort(objArray1);                         // (2) Natural order: [I, OK, am]

Comparable<Integer>[] comps = new Integer[] {5, 3, 7, 1}; // Integer
Arrays.sort(comps, 1, 4, Collections.reverseOrder());// (3) Reverse natural
order:
                                                        //     [5, 7, 3, 1]


Object[] objArray2 = {23, 3.14, "ten"};         // Not mutually comparable
//  Arrays.sort(objArray2);                     // (4) ClassCastException

Number[] numbers = {23, 3.14, 10L};             // Not mutually comparable
//  Arrays.sort(numbers);                       // (5) ClassCastException
```

Searching in Arrays

- The `Arrays` class provides enough overloaded versions of the `binarySearch()` method to search in practically any type of array that is sorted. The discussion on searching in lists (page 27) is also applicable to searching in arrays.

```
int binarySearch(type[] array, type key)
int binarySearch(type[] array, int fromIndex, int toIndex, type key)
```

These methods search for `key` in the `array` where the elements are sorted according to the natural ordering of the elements. Permitted *type* for elements include `byte`, `char`, `double`, `float`, `int`, `long`, `short`, and `Object`. In the case where an array of objects is passed as argument, the *objects* must be sorted in natural ordering, as defined by the `Comparable<E>` interface.

The methods return the index to the key in the sorted array, if the key exists. If not, a negative index is returned, corresponding to -(*insertion point + 1*), where *insertion point* is the index of the element where the key would have been found, if it had been in the array. In case there are duplicate elements equal to the key, there is no guarantee which duplicate's index will be returned. The elements and the key must be *mutually comparable*.

The bounds, if specified in the methods, define a half-open interval. The search is then confined to this interval.

```
<E> int binarySearch(E[] array, E key, Comparator<? super E> c)
<E> int binarySearch(E[] array, int fromIndex, int toIndex, E key,
                     Comparator<? super E> c)
```

The two generic methods above require that the array is sorted according to the total ordering dictated by the comparator. In particular, its elements are mutually comparable according to this comparator. The comparator must be equivalent to the one that was used for sorting the array, otherwise the results are unpredictable.

- The experiment from page 27 with a list of strings is now repeated with an array of strings, giving identical results. In the code below the return value -3 indicates that the key would have been found at index 2 had it been in the list.

```
Arrays.sort(strArray);
// Sorted according to natural order: [Bigfoot, big, bigger, biggest]
// Search in natural order:
out.println(Arrays.binarySearch(strArray, "bigger"));   // Successful:    2
out.println(Arrays.binarySearch(strArray, "bigfeet"));  // Unsuccessful: -3
out.println(Arrays.binarySearch(strArray, "bigmouth")); // Unsuccessful: -5
```

- Results are unpredictable if the array is not sorted, or the ordering used in the search is not the same as the sort ordering. Searching in the `strArray` using reverse natural ordering when the array is sorted in natural ordering, gives the wrong result:

```
out.println(Arrays.binarySearch(strArray, "bigger",
                                Collections.reverseOrder()));  //  -1 (INCORRECT)
```

- A `ClassCastException` is thrown if the key and the elements are not mutually comparable:

```
out.println(Arrays.binarySearch(strArray, 4)); // Key: 4 => ClassCastException
```

- However, this incompatibility is caught at compile time in the case of arrays with primitive values:

```
// Sorted int array (natural order): [1, 3, 5, 7]
out.println(Arrays.binarySearch(intArray, 4.5));// Key: 4.5 => Compile time
error!
```

Creating List Views of Arrays

- The `asList()` method in the `Arrays` class and the `toArray()` method in the `Collection` interface provide the bidirectional bridge between arrays and collections. The `asList()` method of the `Arrays` class creates `List` views of arrays.

> `<E> List<E> asList(E... elements)`
> Returns a *fixed-size list view* that is backed by the *array* corresponding to the variable arity parameter `elements`. The method is annotated with `@SafeVarargs` because of variable arity parameter. The annotation suppresses the heap pollution warning in its declaration and also unchecked generic array creation warning at the call sites.

- Changes to the elements of the list view reflect in the array, and vice versa. The list view is said to be *backed* by the array. The size of the list view is equal to the array length and *cannot* be changed. The iterator for a list view does not support the `remove()` method.
- The code below illustrates using the `asList()` method. The `list1` is backed by the `array1` at (1). The `list2` is backed by an implicit array of `Integers` at (2). An array of primitive type cannot be passed as argument to this method, as evident by the compile time error at (3). However, the `Collections.addAll()` method provides better performance when adding a few elements to an *existing* collection.

```
Integer[] array1 = new Integer[] {9, 1, 1};
int[] array2 = new int[] {9, 1, 1};
```

```
List<Integer> list1 = Arrays.asList(array1);          // (1) An array of Integers

List<Integer> list2 = Arrays.asList(9, 1, 1);          // (2) Varargs
// List<Integer> intList3 = Arrays.asList(array2);    // (3) Compile-time error!
```

- Various operations on the `list1` show how changes are reflected in the backing `array1`. Elements cannot be added to the list view (shown at (5)), and elements cannot be removed from a list view (shown at (10)). An `UnsupportedOperationException` is thrown in both cases. An element at a given position can be changed, as shown at (6). The change is reflected in the `list1` and the `array1`, as shown at (7a) and (7b), respectively. A sublist view is created from the `list1` at (8), and sorted at (11). The changes in the `sublist1` are reflected in the `list1` and the backed `array1`.

```
System.out.println(list1);                          // (4) [9, 1, 1]
// list1.add(10);                                  // (5) UnsupportedOperationException
list1.set(0, 10);                                   // (6)
System.out.println(list1);                          // (7a) [10, 1, 1]
System.out.println(Arrays.toString(array1)); // (7b) [10, 1, 1]
List<Integer> sublist1 = list1.subList(0, 2);// (8)
System.out.println(sublist1);                       // (9) [10, 1]
// sublist1.clear();                               // (10) UnsupportedOperationException
Collections.sort(sublist1);                         // (11)
System.out.println(sublist1);                       // (12a) [1, 10]
System.out.println(list1);                          // (12b) [1, 10, 1]
```

```
System.out.println(Arrays.toString(array1)); // (12c) [1, 10, 1]
```

- The code below shows how duplicates can be eliminated from an array, using these two methods:

```
String[] jiveArray   = new String[] {"java", "jive", "java", "jive"};
Set<String> jiveSet  = new HashSet<>(Arrays.asList(jiveArray));// (1)
String[] uniqueJiveArray = jiveSet.toArray(new String[0]);     // (2)
System.out.println(Arrays.toString(uniqueJiveArray));           // (3) [java,
jive]
```

- At (1), the `jiveArray` is used to create a `List`, which, in turn, is used to create a `Set`. At (2), the argument to the `toArray()` method specifies the type of the array to be created from the set. The final array `uniqueJiveArray` does not contain duplicates, as can be seen at (3).

Miscellaneous Utility Methods in the `Arrays` Class

- The methods `toString()` (page 38) and `fill()` (Example 6.1, Example 6.10) have previously been used in this chapter.

> `String toString(`*type*`[] array)`
> `String deepToString(Object[] array)`
> Returns a text representation of the contents (or "deep contents") of the specified array. The first method calls the `toString()` method of the element if this element is not an array, but calls the `Object.toString()` method if the element is an array—creates a "one-dimension" text representation of an array. The second method "goes deeper" and creates a multidimensional textual string representation for an arrays of arrays—that is, arrays that have arrays as elements. The *type* can be any primitive type.
>
> `void fill(`*type*`[] array, `*type*` value)`
> `void fill(`*type*`[] array, int fromIndex, int toIndex, `*type*` value)`
> The *type* can be any primitive type or `Object` in these methods. Assigns the specified value to each element of the specified array or the subarray given by the half-open interval whose index bounds are specified.

- The code blow illustrates how the `Arrays.fill()` method can be used. Each element in the local array `bar` is assigned the character `'*'` using the `fill()` method and the array is printed.

```
for (int i = 0; i < 5; ++i) {
```

```
    char[] bar = new char[i];
    Arrays.fill(bar, '*');
    System.out.printf("%d %s%n", i, Arrays.toString(bar));
  }
```

- Output from the `for(;;)` loop:

```
0 []
1 [*]
2 [*, *]
3 [*, *, *]
4 [*, *, *, *]
```

- The `Arrays.toString()` method has been used in many examples to convert the contents of an array to a string representation. The `Arrays.deepToString()` method can be used to convert the contents of a *multidimensional* array to a string representation, where each element that is an array is also converted to a string representation.

```
char[][] twoDimArr = new char[2][2];
Arrays.fill(twoDimArr[0], '*');
Arrays.fill(twoDimArr[1], '*');
System.out.println(Arrays.deepToString(twoDimArr));    // [[*, *], [*, *]]
```

- In addition to initializing an array in an array initialization block, the following overloaded `setAll()` methods in the `Arrays` class can be used to initialize an existing

array.

> `<T> void setAll(T[] array, IntFunction<? extends T> generator)`
> `void setAll(int[] array, IntUnaryOperator generator)`
> `void setAll(long[] array, IntToLongFunction generator)`
> `void setAll(double[] array, IntToDoubleFunction generator)`
> Set all elements of the specified array, using the provided generator function to compute each element.

- The code below illustrates how each element of an existing array can be initialized to the result of applying a function.

```
String[] strArr = new String[5];
Arrays.setAll(strArr, i -> "Str@" + i); // [Str@0, Str@1, Str@2, Str@3, Str@4]

int[] intArr = new int[4];
Arrays.setAll(intArr, i -> i * i);      // [0, 1, 4, 9]
```