Chapter 7

Many data structures have been developed so programmers can store and retrieve values efficiently. The Java API provides implementations of common data structures and algorithms, as well as a framework to organize them. In this chapter, you will learn how to work with lists, sets, maps, and other collections.

The key points of this chapter are:

1. The `Collection` interface provides common methods for all collections, except for maps which are described by the `Map` interface.
2. A list is a sequential collection in which each element has an integer index.
3. A set is optimized for efficient containment testing. Java provides `HashSet` and `TreeSet` implementations.
4. For maps, you have the choice between `HashMap` and `TreeMap` implementations. A `LinkedHashMap` retains insertion order.
5. The `Collection` interface and `Collections` class provide many useful algorithms: set operations, searching, sorting, shuffling, and more.
6. Views provide access to data stored elsewhere using the standard collection interfaces.

7.1 An Overview of the Collections Framework

The Java collections framework provides implementations of common data structures. To make it easy to write code that is independent of the choice of data structures, the collections framework provides a number of common interfaces, shown in Figure 7-1. The fundamental interface is `Collection` whose methods are shown in Table 7-1.

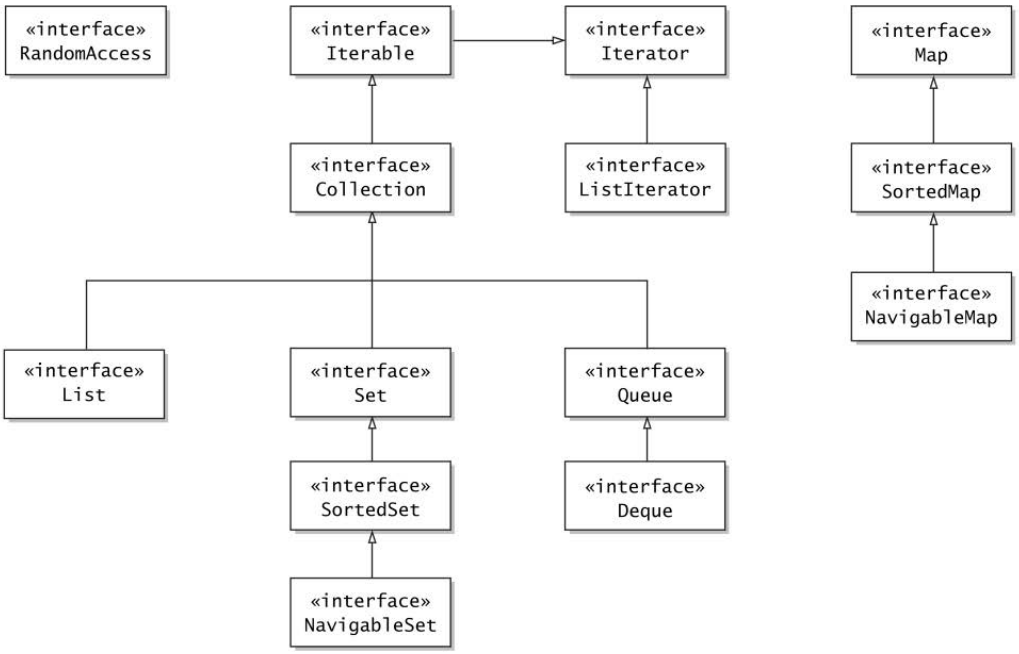

Figure 7-1 Interfaces in the Java collections framework

Table 7-1 The Methods of the `Collection<E>` Interface

Method	Description
`boolean add(E e)` `boolean addAll(Collection<? extends E> c)`	Adds `e`, or the elements in `c`. Returns `true` if the collection changed.

`boolean remove(Object o)`
`boolean removeAll(Collection<?> c)`
`boolean retainAll(Collection<?> c)`
`boolean removeIf(Predicate<? super E> filter)`
`void clear()`
	Removes `o`, or the elements in `c`, or the elements not in `c`, or matching elements, or all elements. The first four methods return `true` if the collection changed.

(Continues)

Table 7–1 The Methods of the `Collection<E>` Interface *(Continued)*

Method	Description
`int size()`	Returns the number of elements in this collection.

`boolean isEmpty()`
`boolean contains(Object o)`
`boolean containsAll(Collection<?> c)`
	Returns `true` if this collection is empty, or contains `o`, or contains all elements in `c`.

`Iterator<E> iterator()`
`Stream<E> stream()`
`Stream<E> parallelStream()`
`Spliterator<E> spliterator()`
	Yields an iterator, or a stream, or a possibly parallel stream, or a spliterator for visiting the elements of this collection. See Section 7.2 for iterators and Chapter 8 for streams. Spliterators are only of interest to implementors of streams.

`Object[] toArray()` `T[] toArray(T[] a)`	Returns an array with the elements of this collection. The second method returns `a` if it has sufficient length.

A `List` is a sequential collection: Elements have position 0, 1, 2, and so on. Table 7-2 shows the methods of that interface.

The `List` interface is implemented both by the `ArrayList` class, which you have seen throughout this book, and the `LinkedList` class. If you took a course on data structures, you probably remember a linked list—a sequence of linked nodes, each carrying an element. Insertion in the middle of a linked list is speedy—you just splice in a node. But to get to the middle, you have to follow all the links from the beginning, which is slow. There are applications for linked lists, but most application programmers will probably stick with array lists when they need a sequential collection. Still, the `List` interface is useful. For example, the method `Collections.nCopies(n, o)` returns a `List` object with `n` copies of the object `o`. That object "cheats" in that it doesn't actually store `n` copies but, when you ask about any one of them, returns `o`.

CAUTION: The `List` interface provides methods to access the *n*th element of a list, even though such an access may not be efficient. To indicate that it is, a collection class should implement the `RandomAccess` interface. This is a tagging interface without methods. For example, `ArrayList` implements `List` and `RandomAccess`, but `LinkedList` implements only the `List` interface.

Table 7–2 The `List` Interface

Method	Description

`boolean add(int index, E e)`
`boolean addAll(int index,`
`  Collection<? extends E> c)`
`boolean add(E e)`
`boolean addAll(Collection<? extends E> c)`
	Adds `e`, or the elements in `c`, before `index` or to the end. Returns `true` if the list changed.

`E get(int index)`
`E set(int index, E element)`
`E remove(int index)`
	Gets, sets, or removes the element at the given index. The last two methods return the element at the index before the call.

`int indexOf(Object o)` `int lastIndexOf(Object o)`	Returns the index of the first or last element equal to `o`, or `-1` if there is no match.
`ListIterator<E> listIterator()` `ListIterator<E> listIterator(int index)`	Yields a list iterator for all elements or the elements starting at `index`.
`void replaceAll(UnaryOperator<E> operator)`	Replaces each element with the result of applying the operator to it.
`void sort(Comparator<? super E> c)`	Sorts this list, using the ordering given by `c`.
`static List<E> of(E... elements)`	Yields an unmodifiable list containing the given elements.
`List<E> subList(int fromIndex, int toIndex)`	Yields a view (Section 7.6) of the sublist starting at `fromIndex` and ending before `toIndex`.

In a `Set`, elements are not inserted at a particular position, and duplicate elements are not allowed. A `SortedSet` allows iteration in sort order, and a `NavigableSet` has methods for finding neighbors of elements. You will learn more about sets in Section 7.3, "Sets" (page 242).

A `Queue` retains insertion order, but you can only insert elements at the tail and remove them from the head (just like a queue of people). A `Deque` is a double-ended queue with insertion and removal at both ends.

All collection interfaces are generic, with a type parameter for the element type (`Collection<E>`, `List<E>`, and so on). The `Map<K, V>` interface has a type parameter `K` for the key type and `V` for the value type.

You are encouraged to use the interfaces as much as possible in your code. For example, after constructing an `ArrayList`, store the reference in a variable of type `List`:

```
List<String> words = new ArrayList<>();
```

Whenever you implement a method that processes a collection, use the least restrictive interface as parameter type. Usually, a `Collection`, `List`, or `Map` will suffice.

One advantage of a collections framework is that you don't have to reinvent the wheel when it comes to common algorithms. Some basic algorithms (such as `addAll` and `removeIf`) are methods of the `Collection` interface. The `Collections` utility class contains many additional algorithms that operate on various kinds of collections. You can sort, shuffle, rotate, and reverse lists, find the maximum or minimum, or the position of an arbitrary element in a collection, and generate collections with no elements, one element, or `n` copies of the same element. Table 7-3 provides a summary.

Table 7-3 Useful Methods of the `Collections` Class

Method (all are static)	Description
`boolean disjoint(Collection<?> c1, Collection<?> c2)`	Returns `true` if the collections have no elements in common.
`boolean addAll(Collection<? super T> c, T... elements)`	Adds all elements to `c`.
`void copy(List<? super T> dest, List<? extends T> src)`	Copies all elements from `src` to the same indexes in `dest` (which must be at least as long as `src`).
`boolean replaceAll(List<T> list, T oldVal, T newVal)`	Replaces all `oldVal` elements with `newVal`, either of which may be `null`. Returns `true` if at least one match was found.
`void fill(List<? super T> list, T obj)`	Sets all elements of the list to `obj`.
`List<T> nCopies(int n, T o)`	Yields an immutable list with `n` copies of `o`.
`int frequency(Collection<?> c, Object o)`	Returns the number of elements in `c` equal to `o`.

(Continues)

Table 7–3 Useful Methods of the `Collections` Class *(Continued)*

Method (all are static)	Description
`int indexOfSubList(List<?> source, List<?> target)` `int lastIndexOfSubList(List<?> source, List<?> target)`	Returns the start of the first or last occurrence of the target list within the source list, or `-1` if there is none.
`int binarySearch(List<? extends Comparable<? super T>> list, T key)` `int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`	Returns the position of the key, assuming that the list is sorted by the natural element order or `c`. If the key is not present, returns `-i - 1` where `i` is the location at which the key should be inserted.
`sort(List<T> list)` `sort(List<T> list, Comparator<? super T> c)`	Sorts the list, using the natural element order or `c`.
`void swap(List<?> list, int i, int j)`	Swaps the elements at the given position.
`void rotate(List<?> list, int distance)`	Rotates the list, moving the element with index `i` to `(i + distance) % list.size()`.

`void reverse(List<?> list)`
`void shuffle(List<?> list)`
`void shuffle(List<?> list, Random rnd)`
	Reverses or randomly shuffles the list.

`synchronized(Collection|List|Set|SortedSet|NavigableSet|Map|SortedMap|NavigableMap)()`	Yields a synchronized view (see Section 7.6).
`unmodifiable(Collection|List|Set|SortedSet|NavigableSet|Map|SortedMap|NavigableMap)()`	Yields an unmodifiable view (see Section 7.6).
`checked(Collection|List|Set|SortedSet|NavigableSet|Map|SortedMap|NavigableMap|Queue)()`	Yields a checked view (see Section 7.6).

7.2 Iterators

Each collection provides a way to iterate through its elements in some order. The `Iterable<T>` superinterface of `Collection` defines a method

```
Iterator<T> iterator()
```

It yields an iterator that you can use to visit all elements.

```
Collection<String> coll = ...;
Iterator<String> iter = coll.iterator();
while (iter.hasNext()) {
    String element = iter.next();
    Process element
}
```

In this case, you can simply use the enhanced `for` loop:

```
for (String element : coll)  {
    Process element
}
```

NOTE: For *any* object `c` of a class that implements the `Iterable<E>` interface, the enhanced `for` loop is translated to the preceding form.

The `Iterator` interface also has a `remove` method which removes the previously visited element. This loop removes all elements that fulfill a condition:

```
while (iter.hasNext()) {
    String element = iter.next();
    if (element fulfills the condition)
        iter.remove();
}
```

However, it is easier to use the `removeIf` method:

```
coll.removeIf(e -> e fulfills the condition);
```

CAUTION: The `remove` method removes the last element that the iterator has returned, not the element to which the iterator points. You can't call `remove` twice without an intervening call to `next` or `previous`.

The `ListIterator` interface is a subinterface of `Iterator` with methods for adding an element before the iterator, setting the visited element to a different value, and for navigating backwards. It is mainly useful for working with linked lists.

```
List<String> friends = new LinkedList<>();
ListIterator<String> iter = friends.listIterator();
iter.add("Fred"); // Fred |
iter.add("Wilma"); // Fred Wilma |
iter.previous(); // Fred | Wilma
iter.set("Barney"); // Fred | Barney
```

CAUTION: If you have multiple iterators visiting a data structure and one of them mutates it, the other ones can become invalid. An invalid iterator may throw a `ConcurrentModificationException` if you continue using it.

7.3 Sets

A set can efficiently test whether a value is an element, but it gives up something in return: It doesn't remember in which order elements were added. Sets are useful whenever the order doesn't matter. For example, if you want to disallow a set of bad words as usernames, their order doesn't matter. You just want to know whether a proposed username is in the set or not.

The `Set` interface is implemented by the `HashSet` and `TreeSet` classes. Internally, these classes use very different implementations. If you have taken a course in data structures, you may know how to implement hash tables and binary trees—but you can use these classes without knowing their internals.

Generally, hash sets are a bit more efficient, provided you have a good *hash function* for your elements. Library classes such as `String` or `Path` have good hash functions. You learned how to write hash function for your own classes in Chapter 4.

For example, that set of bad words can be implemented simply as

```
Set<String> badWords = new HashSet<>();
badWords.add("sex");
badWords.add("drugs");
badWords.add("c++");
if (badWords.contains(username.toLowerCase()))
    System.out.println("Please choose a different user name");
```

You use a `TreeSet` if you want to traverse the set in sorted order. One reason you might want to do this is to present users a sorted list of choices.

The element type of the set must implement the `Comparable` interface, or you need to supply a `Comparator` in the constructor.

```
TreeSet<String> countries = new TreeSet<>(); // Visits added countries in sorted order
countries = new TreeSet<>((u, v) ->
    u.equals(v) ? 0
    : u.equals("USA") ? -1
    : v.equals("USA") ? 1
    : u.compareTo(v));
    // USA always comes first
```

The `TreeSet` class implements the `SortedSet` and `NavigableSet` interfaces, whose methods are shown in Tables 7-4 and 7-5.

Table 7-4 `SortedSet<E>` Methods

Method	Description
`E first()` `E last()`	The first and last element in this set.

`SortedSet<E> headSet(E toElement)`
`SortedSet<E> subSet(E fromElement, E toElement)`
`SortedSet<E> tailSet(E fromElement)`
	Returns a view of the elements starting at `fromElement` and ending before `toElement`.

Table 7-5 `NavigableSet<E>` Methods

Method	Description

`E higher(E e)`
`E ceiling(E e)`
`E floor(E e)`
`E lower(E e)`
	Returns the closest element $>$ | $\geq$ | $\leq$ | $<$ `e`.

`E pollFirst()` `E pollLast()`	Removes and returns the first or last element, or returns `null` if the set is empty.

`NavigableSet<E> headSet(E toElement, boolean inclusive)`
`NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toExclusive)`
`NavigableSet<E> tailSet(E fromElement, boolean inclusive)`
	Returns a view of the elements from `fromElement` to `toElement` (inclusive or exclusive).

7.4 Maps

Maps store associations between keys and values. Call `put` to add a new association, or change the value of an existing key:

```
Map<String, Integer> counts = new HashMap<>();
counts.put("Alice", 1); // Adds the key/value pair to the map
counts.put("Alice", 2); // Updates the value for the key
```

This example uses a hash map which, as for sets, is usually the better choice if you don't need to visit the keys in sorted order. If you do, use a `TreeMap` instead.

Here is how you can get the value associated with a key:

```
int count = counts.get("Alice");
```

If the key isn't present, the `get` method returns `null`. In this example, that would cause a `NullPointerException` when the value is unboxed. A better alternative is

```
int count = counts.getOrDefault("Alice", 0);
```

Then a count of `0` is returned if the key isn't present.

When you update a counter in a map, you first need to check whether the counter is present, and if so, add 1 to the existing value. The `merge` method simplifies that common operation. The call

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

associates `word` with 1 if the key wasn't previously present, and otherwise combines the previous value and 1, using the `Integer::sum` function.

Table 7-6 summarizes the map operations.

You can get *views* of the keys, values, and entries of a map by calling these methods:

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

The collections that are returned are not copies of the map data, but they are connected to the map. If you remove a key or entry from the view, then the entry is also removed from the underlying map.

To iterate through all keys and values of a map, you can iterate over the set returned by the `entrySet` method:

```
for (Map.Entry<String, Integer> entry : counts.entrySet()) {
    String k = entry.getKey();
    Integer v = entry.getValue();
    Process k, v
}
```

Or simply use the `forEach` method:

```
counts.forEach((k, v) -> {
    Process k, v
});
```

CAUTION: Some map implementations (for example, `ConcurrentHashMap`) disallow `null` for keys or values. And with those that allow it (such as `HashMap`), you need to be very careful if you do use `null` values. A number of map methods interpret a `null` value as an indication that an entry is absent, or should be removed.

Table 7-6 `Map<K, V>` Methods

Method	Description
`V get(Object key)` `V getOrDefault(Object key, V defaultValue)`	If `key` is associated with a non-`null` value `v`, returns `v`. Otherwise, returns `null` or `defaultValue`.
`V put(K key, V value)`	If `key` is associated with a non-`null` value `v`, associates `key` with `value` and returns `v`. Otherwise, adds entry and returns `null`.
`V putIfAbsent(K key, V value)`	If `key` is associated with a non-`null` value `v`, ignores `value` and returns `v`. Otherwise, adds entry and returns `null`.
`V merge(K key, V value, BiFunction< ? super V,? super V,? extends V> remappingFunction)`	If `key` is associated with a non-`null` value `v`, applies the function to `v` and `value` and either associates `key` with the result or, if the result is `null`, removes the key. Otherwise, associates `key` with `value`. Returns `get(key)`.
`V compute(K key, BiFunction< ? super K,? super V,? extends V> remappingFunction)`	Applies the function to `key` and `get(key)`. Either associates `key` with the result or, if the result is `null`, removes the key. Returns `get(key)`.
`V computeIfPresent(K key, BiFunction< ? super K,? super V,? extends V> remappingFunction)`	If `key` is associated with a non-`null` value `v`, applies the function to `key` and `v` and either associates `key` with the result or, if the result is `null`, removes the key. Returns `get(key)`.
`V computeIfAbsent(K key, Function< ? super K,? extends V> mappingFunction)`	Applies the function to `key` unless `key` is associated with a non-`null` value. Either associates `key` with the result or, if the result is `null`, removes the key. Returns `get(key)`.
`void putAll(Map<? extends K, ? extends V> m)`	Adds all entries from `m`.
`V remove(Object key)` `V replace(K key, V newValue)`	Removes the key and its associated value, or replaces the old value. Returns the old value, or `null` if none existed.

(Continues)

Table 7–6 `Map<K, V>` Methods *(Continued)*

Method	Description
`boolean remove(Object key, Object value)` `boolean replace(K key, V value, V newValue)`	Provided that `key` was associated with `value`, removes the entry or replaces the old value and returns `true`. Otherwise, does nothing and returns `false`. These methods are mainly of interest when the map is accessed concurrently.
`int size()`	Returns the number of entries.
`boolean isEmpty()`	Checks if this map is empty.
`void clear()`	Removes all entries.
`void forEach(BiConsumer<? super K, ? super V> action)`	Applies the action to all entries.
`void replaceAll(BiFunction<? super K, ? super V,? extends V> function)`	Calls the function on all entries. Associates keys with non-`null` results and removes keys with `null` results.
`boolean containsKey(Object key)` `boolean containsValue(Object value)`	Checks whether the map contains the given key or value.

`Set<K> keySet()`
`Collection<V> values()`
`Set<Map.Entry<K, V>> entrySet()`
	Returns views of the keys, values, and entries.

`static Map<K, V> of()`
`static Map<K, V> of(K k1, V v1)`
`static Map<K, V> of(K k1, V v1, K k2, V v2)`
`...`
	Yields an unmodifiable map containing up to ten keys and values.

TIP: Sometimes, you need to present map keys in an order that is different from the sort order. For example, in the JavaServer Faces framework, you specify labels and values of a selection box with a map. Users would be surprised if the choices were sorted alphabetically (Friday, Monday, Saturday, Sunday, Thursday, Tuesday, Wednesday) or in the hash code order. In that case, use a `LinkedHashMap` that remembers the order in which entries were added and iterates through them in that order.

7.5 Other Collections

In the following sections, I briefly discuss some collection classes that you may find useful in practice.

7.5.1 Properties

The `Properties` class implements a map that can be easily saved and loaded using a plain text format. Such maps are commonly used for storing configuration options for programs. For example:

```
Properties settings = new Properties();
settings.put("width", "200");
settings.put("title", "Hello, World!");
try (OutputStream out = Files.newOutputStream(path)) {
    settings.store(out, "Program Properties");
}
```

The result is the following file:

```
#Program Properties
#Mon Nov 03 20:52:33 CET 2014
width=200
title=Hello, World\!
```

NOTE: As of Java 9, property files are encoded in UTF-8. (Previously, they were encoded in ASCII, with characters greater than `'\u007e'` written as Unicode escapes `\u`*nnnn*.) Comments start with `#` or `!`. A newline in a key or value is written as `\n`. The characters `\`, `#`, `!` are escaped as `\\`, `\#`, `\!`.

To load properties from a file, call

```
try (InputStream in = Files.newInputStream(path)) {
    settings.load(in);
}
```

Then use the `getProperty` method to get a value for a key. You can specify a default value used when the key isn't present:

```
String title = settings.getProperty("title", "New Document");
```

NOTE: For historical reasons, the `Properties` class implements `Map<Object, Object>` even though the values are always strings. Therefore, don't use the `get` method—it returns the value as an `Object`.

The `System.getProperties` method yields a `Properties` object with system properties. Table 7-7 describes the most useful ones.

Table 7–7 Useful System Properties

Property Key	Description
`user.dir`	The "current working directory" of this virtual machine
`user.home`	The user's home directory
`user.name`	The user's account name
`java.version`	The Java runtime version of this virtual machine
`java.home`	The home directory of the Java installation
`java.class.path`	The class path with which this VM was launched
`java.io.tmpdir`	A directory suitable for temporary files (such as `/tmp`)
`os.name`	The name of the operating system (such as `Linux`)
`os.arch`	The architecture of the operating system (such as `amd64`)
`os.version`	The version of the operating system (such as `3.13.0-34-generic`)
`file.separator`	The file separator (`/` on Unix, `\` on Windows)
`path.separator`	The path separator (`:` on Unix, `;` on Windows)
`line.separator`	The newline separator (`\n` on Unix, `\r\n` on Windows)

7.5.2 Bit Sets

The `BitSet` class stores a sequence of bits. A bit set packs bits into an array of `long` values, so it is more efficient to use a bit set than an array of `boolean` values. Bit sets are useful for sequences of flag bits or to represent sets of non-negative integers, where the `i`th bit is `1` to indicate that `i` is contained in the set.

The `BitSet` class gives you convenient methods for getting and setting individual bits. This is much simpler than the bit-fiddling necessary to store bits in `int` or `long` variables. There are also methods that operate on all bits together for set operations, such as union and intersection. See Table 7-8 for a complete list. Note that the `BitSet` class is not a collection class—it does not implement `Collection<Integer>`.

Table 7-8 Methods of the `BitSet` Class

Method	Description
`BitSet()` `BitSet(int nbits)`	Constructs a bit set that can initially hold 64, or `nbits`, bits.

`void set(int bitIndex)`
`void set(int fromIndex, int toIndex)`
`void set(int bitIndex, boolean value)`
`void set(int fromIndex, int toIndex, boolean value)`
	Sets the bit at the given index, or from `fromIndex` (inclusive) to `toIndex` (exclusive), to `1` or to the given value.

`void clear(int bitIndex)`
`void clear(int fromIndex, int toIndex)`
`void clear()`
	Sets the bit at the given index, or from `fromIndex` (inclusive) to `toIndex` (exclusive), or all bits to `0`.

`void flip(int bitIndex)` `void flip(int fromIndex, int toIndex)`	Flips the bit at the given index, or from `fromIndex` (inclusive) to `toIndex` (exclusive).
`boolean get(int bitIndex)` `BitSet get(int fromIndex, int toIndex)`	Gets the bit at the given index, or from `fromIndex` (inclusive) to `toIndex` (exclusive).

`int nextSetBit(int fromIndex)`
`int previousSetBit(int fromIndex)`
`int nextClearBit(int fromIndex)`
`int previousClearBit(int fromIndex)`
	Returns the index of the next/previous `1`/`0` bit, or `-1` if none exists.

`void and(BitSet set)`
`void andNot(BitSet set)`
`void or(BitSet set)`
`void xor(BitSet set)`
	Forms the intersection|difference|union|symmetric difference with `set`.

`int cardinality()`	Returns the number of `1` bits in this bit set. **Caution:** The `size` method returns the current size of the bit vector, not the size of the set.
`byte[] toByteArray[]` `long[] toLongArray[]`	Packs the bits of this bit set into an array.
`IntStream stream()` `String toString()`	Returns a stream or string of the integers (that is, indexes of `1` bits) in this bit set.

`static BitSet valueOf(byte[] bytes)`
`static BitSet valueOf(long[] longs)`
`static BitSet valueOf(ByteBuffer bb)`
`static BitSet valueOf(LongBuffer lb)`
	Yields a bit set containing the supplied bits.

`boolean isEmpty()` `boolean intersects(BitSet set)`	Checks whether this bit set is empty, or has an element in common with `set`.

7.5.3 Enumeration Sets and Maps

If you collect sets of enumerated values, use the `EnumSet` class instead of `BitSet`. The `EnumSet` class has no public constructors. Use a static factory method to construct the set:

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
Set<Weekday> always = EnumSet.allOf(Weekday.class);
Set<Weekday> never = EnumSet.noneOf(Weekday.class);
Set<Weekday> workday = EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
Set<Weekday> mwf = EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY, Weekday.FRIDAY);
```

You can use the methods of the `Set` interface to work with an `EnumSet`.

An `EnumMap` is a map with keys that belong to an enumerated type. It is implemented as an array of values. You specify the key type in the constructor:

```
EnumMap<Weekday, String> personInCharge = new EnumMap<>(Weekday.class);
personInCharge.put(Weekday.MONDAY, "Fred");
```

7.5.4 Stacks, Queues, Deques, and Priority Queues

A stack is a data structure for adding and removing elements at one end (the "top" of the stack). A queue lets you efficiently add elements at one end (the "tail") and remove them from the other end (the "head"). A double-ended queue, or deque, supports insertion and removal at both ends. With all these data structures, adding elements in the middle is not supported.

The `Queue` and `Deque` interfaces define the methods for these data structures. There is no `Stack` interface in the Java collections framework, just a legacy `Stack` class from the earliest days of Java that you should avoid. If you need a stack, queue, or deque and are not concerned about thread safety, use an `ArrayDeque`.

With a stack, use the `push` and `pop` methods.

```
ArrayDeque<String> stack = new ArrayDeque<>();
stack.push("Peter");
stack.push("Paul");
stack.push("Mary");
while (!stack.isEmpty())
    System.out.println(stack.pop());
```

With a queue, use `add` and `remove`.

```
Queue<String> queue = new ArrayDeque<>();
queue.add("Peter");
queue.add("Paul");
queue.add("Mary");
while (!queue.isEmpty())
    System.out.println(queue.remove());
```

Threadsafe queues are commonly used in concurrent programs. You will find more information about them in Chapter 10.

A *priority queue* retrieves elements in sorted order after they were inserted in arbitrary order. That is, whenever you call the `remove` method, you get the smallest element currently in the priority queue.

A typical use for a priority queue is job scheduling. Each job has a priority. Jobs are added in random order. Whenever a new job can be started, the highest priority job is removed from the queue. (Since it is traditional for priority 1 to be the "highest" priority, the `remove` operation yields the minimum element.)

```
public class Job implements Comparable<Job> { ... }
...
PriorityQueue<Job> jobs = new PriorityQueue<>();
jobs.add(new Job(4, "Collect garbage"));
jobs.add(new Job(9, "Match braces"));
jobs.add(new Job(1, "Fix memory leak"));
...
while (jobs.size() > 0) {
    Job job = jobs.remove(); // The most urgent jobs are removed first
    execute(job);
}
```

Just like a `TreeSet`, a priority queue can hold elements of a class that implements the `Comparable` interface, or you can supply a `Comparator` in the constructor. However, unlike a `TreeSet`, iterating over the elements does not necessarily yield them in sorted order. The priority queue uses algorithms for adding and removing elements that cause the smallest element to gravitate to the root, without wasting time on sorting all elements.

7.5.5 Weak Hash Maps

The `WeakHashMap` class was designed to solve an interesting problem. What happens with a value whose key is no longer used anywhere in your program? If the last reference to a key has gone away, there is no longer any way to refer to the value object so it should be removed by the garbage collector.

It isn't quite so simple. The garbage collector traces live objects. As long as the map object is live, all entries in it are live and won't be reclaimed—and neither will be the values that are referenced by the entries.

This is the problem that the `WeakHashMap` class solves. This data structure cooperates with the garbage collector to remove key/value pairs when the only reference to the key is the one from the hash table entry.

Technically, the `WeakHashMap` uses *weak references* to hold keys. A `WeakReference` object holds a reference to another object—in our case, a hash table key. Objects of this type are treated in a special way by the garbage collector. If an object is reachable *only* by a weak reference, the garbage collector reclaims the object and places the weak reference into a queue associated with the `WeakReference` object. Whenever a method is invoked on it, a `WeakHashMap` checks its queue of weak references for new arrivals and removes the associated entries.

7.6 Views

A collection *view* is a lightweight object that implements a collection interface, but doesn't store elements. For example, the `keySet` and `values` methods of a map yield views into the map.

In the following sections, you will see some views that are provided by the Java collections framework.

7.6.1 Small Collections

The `List`, `Set`, and `Map` interfaces provide static methods yielding a set or list with given elements, and a map with given key/value pairs.

For example,

```
List<String> names = List.of("Peter", "Paul", "Mary");
Set<Integer> numbers = Set.of(2, 3, 5);
```

yield a list and a set with three elements. For a map, specify the keys and values like this:

```
Map<String, Integer> scores = Map.of("Peter", 2, "Paul", 3, "Mary", 5);
```

The elements, keys, or values may not be `null`.

The `List` and `Set` interfaces have 11 `of` methods with zero to ten arguments, and an `of` method with a variable number of arguments. The specializations are provided for efficiency.

For the `Map` interface, it is not possible to provide a version with variable arguments since the argument types alternate between the key and value types. There is a static method `ofEntries` that accepts an arbitrary number of `Map.Entry<K, V>` objects, which you can create with the static `entry` method. For example,

```
import static java.util.Map.*;
...
Map<String, Integer> scores = ofEntries(
    entry("Peter", 2),
    entry("Paul", 3),
    entry("Mary", 5));
```

The `of` and `ofEntries` methods produce objects of classes that have an instance variable for each element, or that are backed by an array.

These collection objects are *unmodifiable*. Any attempt to change their contents results in an `UnsupportedOperationException`.

If you want a mutable collection, you can pass the unmodifiable collection to the constructor:

```
List<String> names = new ArrayList<>(List.of("Peter", "Paul", "Mary"));
```

NOTE: There is also a static `Arrays.asList` method that is similar to `List.of`. It returns a mutable list that is not resizable. That is, you can call `set` but not `add` or `remove` on the list.

7.6.2 Ranges

You can form a sublist view of a list. For example,

```
List<String> sentence = ...;
List<String> nextFive = sentence.subList(5, 10);
```

This view accesses the elements with index 5 through 9. Any mutations of the sublist (such as setting, adding, or removing elements) affect the original.

For sorted sets and maps, you specify a range by the lower and upper bound:

```
TreeSet<String> words = ...;
SortedSet<String> asOnly = words.subSet("a", "b");
```

As with `subList`, the first bound is inclusive, and the second exclusive.

The `headSet` and `tailSet` methods yield a subrange with no lower or upper bound.

```
NavigableSet<String> nAndBeyond = words.tailSet("n");
```

With the `NavigableSet` interface, you can choose for each bound whether it should be inclusive or exclusive—see Table 7-5.

For a sorted map, there are equivalent methods `subMap`, `headMap`, and `tailMap`.

7.6.3 Unmodifiable Views

Sometimes, you want to share the contents of a collection but you don't want it to be modified. Of course, you could copy the values into a new collection, but that is potentially expensive. An unmodifiable view is a better choice. Here is a typical situation. A `Person` object maintains a list of friends. If the `getFriends` gave out a reference to that list, a caller could mutate it. But it is safe to provide an unmodifiable list view:

```
public class Person {
    private ArrayList<Person> friends;

    public List<Person> getFriends() {
        return Collections.unmodifiableList(friends);
    }
    ...
}
```

All mutator methods throw an exception when they are invoked on an unmodifiable view.

As you can see from Table 7-3, you can get unmodifiable views as collections, lists, sets, sorted sets, navigable sets, maps, sorted maps, and navigable maps.

NOTE: In Chapter 6, you saw how it is possible to smuggle the wrong kind of elements into a generic collection (a phenomenon called "heap pollution"), and that a runtime error is reported when the inappropriate element is retrieved, not when it is inserted. If you need to debug such a problem, use a *checked view*. Where you constructed, say, an `ArrayList<String>`, instead use

```
List<String> strings
    = Collections.checkedList(new ArrayList<>(), String.class);
```

The view monitors all insertions into the list and throws an exception when an object of the wrong type is added.

NOTE: The `Collections` class produces *synchronized* views that ensure safe concurrent access to data structures. In practice, these views are not as useful as the data structures in the `java.util.concurrent` package that were explicitly designed for concurrent access. I suggest you use those classes and stay away from synchronized views.

Exercises

1. Implement the "Sieve of Erathostenes" algorithm to determine all prime numbers $\leq n$. Add all numbers from 2 to n to a set. Then repeatedly find the smallest element s in the set, and remove s^2, $s \cdot (s + 1)$, $s \cdot (s + 2)$, and so on. You are done when $s^2 > n$. Do this with both a `HashSet<Integer>` and a `BitSet`.
2. In an array list of strings, make each string uppercase. Do this with (a) an iterator, (b) a loop over the index values, and (c) the `replaceAll` method.
3. How do you compute the union, intersection, and difference of two sets, using just the methods of the `Set` interface and without using loops?
4. Produce a situation that yields a `ConcurrentModificationException`. What can you do to avoid it?
5. Implement a method `public static void swap(List<?> list, int i, int j)` that swaps elements in the usual way when the type of `list` implements the `RandomAccess` interface, and that minimizes the cost of visiting the positions at index `i` and `j` if it is not.
6. I encouraged you to use interfaces instead of concrete data structures—for example, a `Map` instead of a `TreeMap`. Unfortunately, that advice goes only so far. Suppose you have a method parameter of type `Map<String, Set<Integer>>`, and someone calls your method with a `HashMap<String, HashSet<Integer>>`. What happens? What parameter type can you use instead?
7. Write a program that reads all words in a file and prints out how often each word occurred. Use a `TreeMap<String, Integer>`.
8. Write a program that reads all words in a file and prints out on which line(s) each of them occurred. Use a map from strings to sets.
9. You can update the counter in a map of counters as

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

   Do the same without the `merge` method, (a) by using `contains`, (b) by using `get` and a `null` check, (c) by using `getOrDefault`, (d) by using `putIfAbsent`.
10. Implement Dijkstra's algorithm to find the shortest paths in a network of cities, some of which are connected by roads. (For a description, check out your favorite book on algorithms or the Wikipedia article.) Use a helper class `Neighbor` that stores the name of a neighboring city and the distance. Represent the graph as a map from cities to sets of neighbors. Use a `PriorityQueue<Neighbor>` in the algorithm.

11. Write a program that reads a sentence into an array list. Then, using `Collections.shuffle`, shuffle all but the first and last word, without copying the words into another collection.
12. Using `Collections.shuffle`, write a program that reads a sentence, shuffles the words, and prints the result. Fix the capitalization of the initial word and the punctuation of the last word (before and after the shuffle). Hint: Don't shuffle the words.
13. The `LinkedHashMap` calls the method `removeEldestEntry` whenever a new element is inserted. Implement a subclass `Cache` that limits the map to a given size provided in the constructor.
14. Write a method that produces an immutable list view of the numbers from `0` to `n`, without actually storing the numbers.
15. Generalize the preceding exercise to an arbitrary `IntFunction`. Note that the result is an infinite collection, so certain methods (such as `size` and `toArray`) should throw an `UnsupportedOperationException`.
16. Improve the implementation of the preceding exercise by caching the last 100 computed function values.
17. Demonstrate how a checked view can give an accurate error report for a cause of heap pollution.
18. The `Collections` class has static variables `EMPTY_LIST`, `EMPTY_MAP`, and `EMPTY_SET`. Why are they not as useful as the `emptyList`, `emptyMap`, and `emptySet` methods?