Java Collections Framework

WEEK 3 PRESENTATION 3 (COLLECTIONS II)

An Overview of the Collections Framework

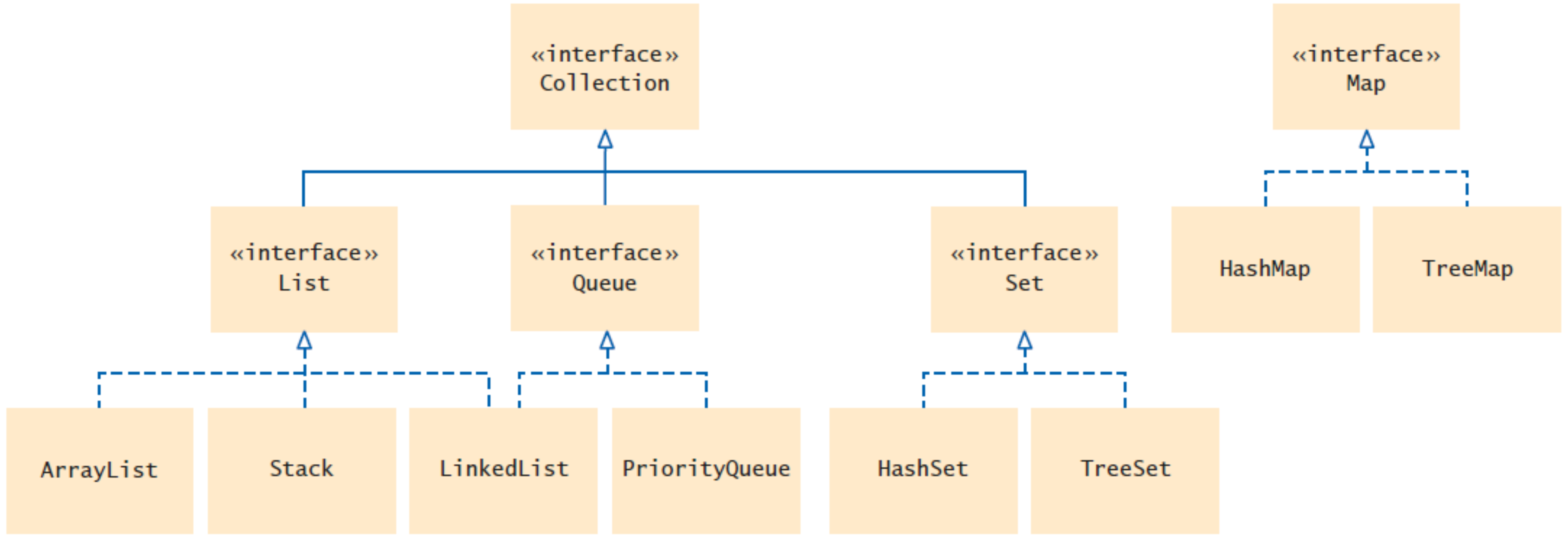


Figure 1 Interfaces and Classes in the Java Collections Framework

Figure 2 A List of Books

Figure 3 A Set of Books

Figure 4 A Stack of Books

A list is a collection that remembers the order of its elements.

A set is an unordered collection of unique elements.

A stack is a collection of elements with “last-in, first-out” retrieval.

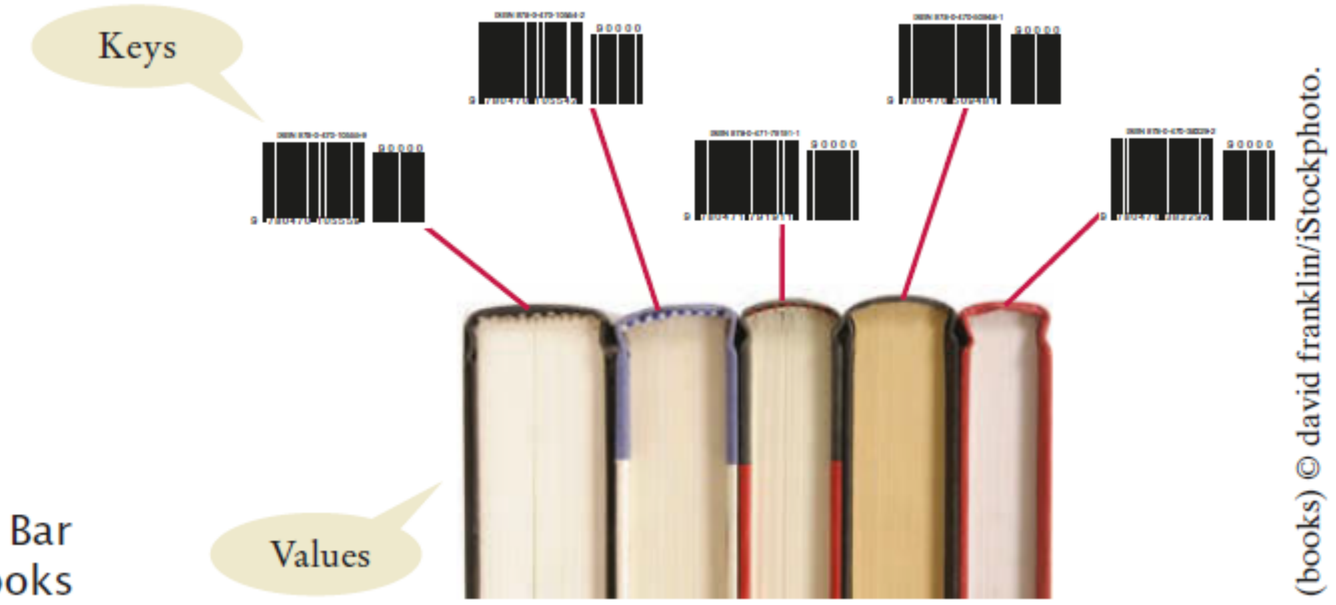


Figure 5
A Map from Bar Codes to Books

A map keeps associations between key and value objects.

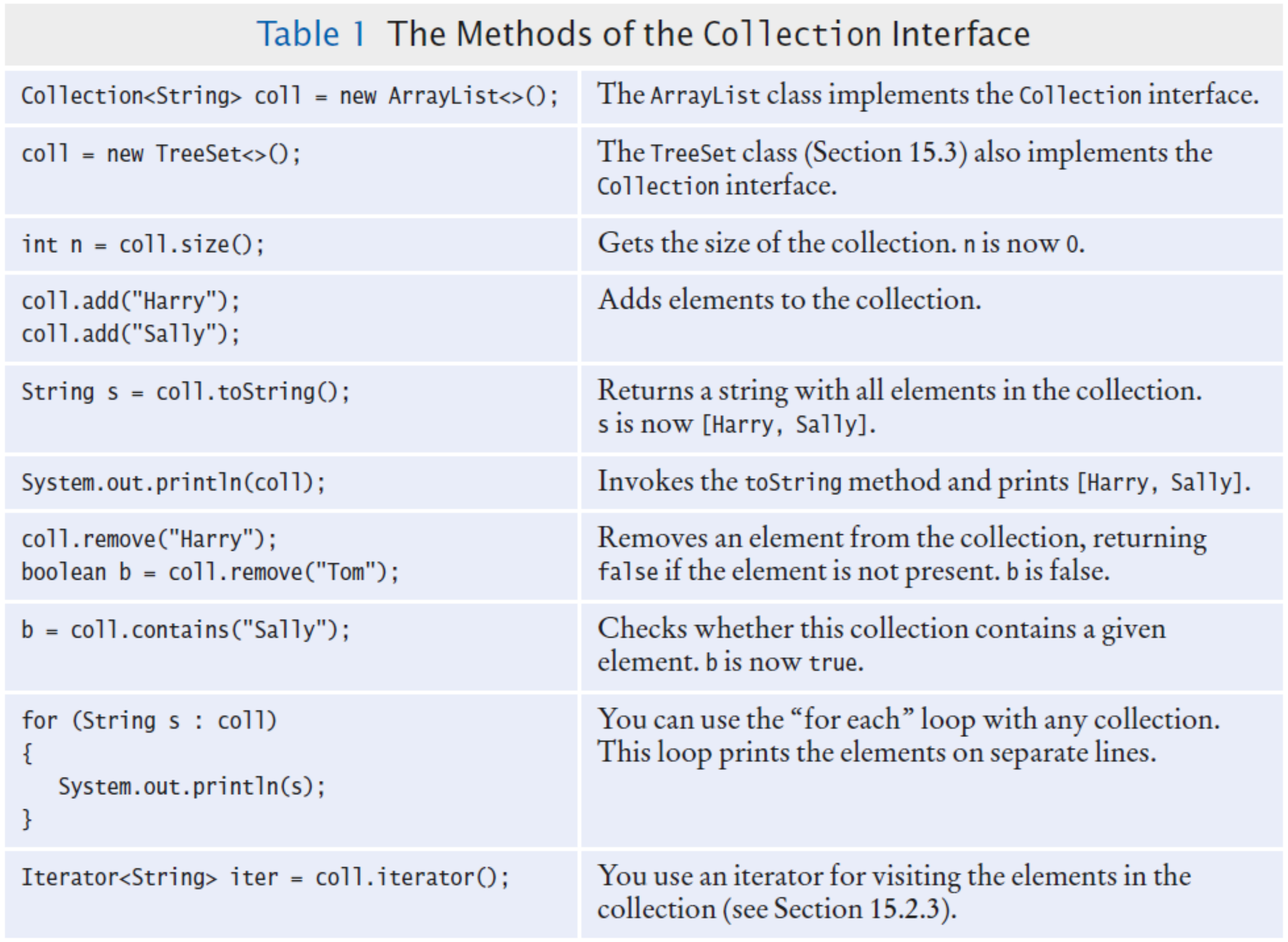


Table 1 The Methods of the Collection Interface

`Collection<String> coll = new ArrayList<>();`	The `ArrayList` class implements the `Collection` interface.
`coll = new TreeSet<>();`	The `TreeSet` class (Section 15.3) also implements the `Collection` interface.
`int n = coll.size();`	Gets the size of the collection. `n` is now `0`.

`coll.add("Harry");`
`coll.add("Sally");`
	Adds elements to the collection.

`String s = coll.toString();`	Returns a string with all elements in the collection. `s` is now `[Harry, Sally]`.
`System.out.println(coll);`	Invokes the `toString` method and prints `[Harry, Sally]`.

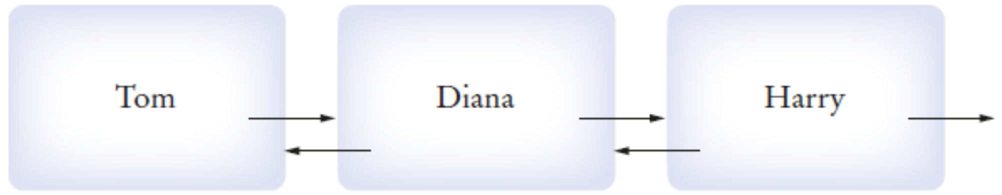
`coll.remove("Harry");`
`boolean b = coll.remove("Tom");`
	Removes an element from the collection, returning `false` if the element is not present. `b` is false.

`b = coll.contains("Sally");`	Checks whether this collection contains a given element. `b` is now `true`.

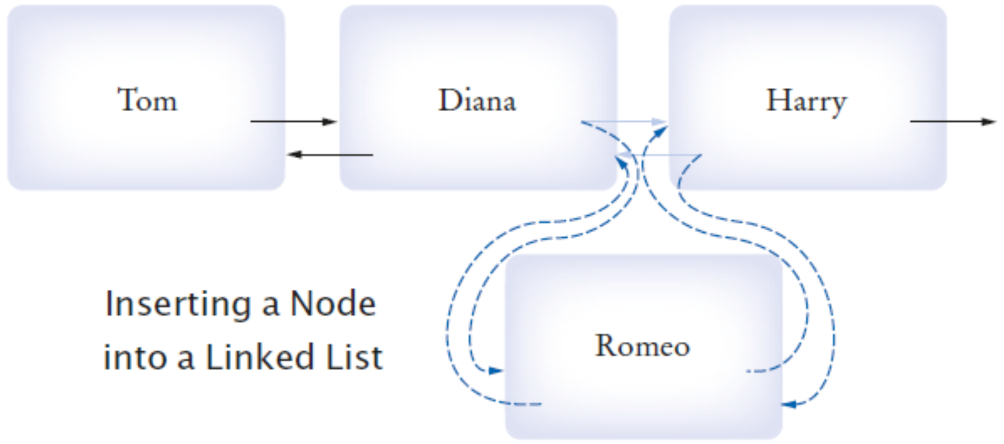
`for (String s : coll)`
`{`
   `System.out.println(s);`
`}`
	You can use the "for each" loop with any collection. This loop prints the elements on separate lines.

`Iterator<String> iter = coll.iterator();`	You use an iterator for visiting the elements in the collection (see Section 15.2.3).

A linked list consists of a number of nodes, each of which has a reference to the next node.

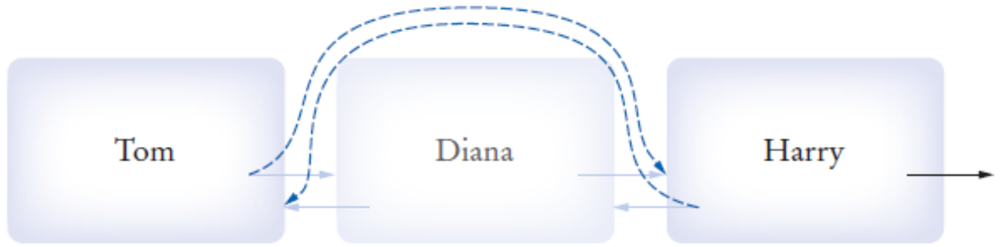


Adding and removing elements at a given location in a linked list is efficient.



Inserting a Node into a Linked List

Visiting the elements of a linked list in sequential order is efficient, but random access is not.



Removing a Node from a Linked List

Table 2 Working with Linked Lists

`LinkedList<String> list = new LinkedList<>();`	An empty list.
`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements (see Table 3 on page 698).

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator();
```

Table 3 Methods of the Iterator and ListIterator Interfaces

`String s = iter.next();`	Assume that iter points to the beginning of the list [Sally] before calling next. After the call, s is "Sally" and the iterator points to the end.

`iter.previous();`
`iter.set("Juliet");`
	The set method updates the last element returned by next or previous. The list is now [Juliet].

`iter.hasNext()`	Returns false because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`   s = iter.previous();`
`}`
	hasPrevious returns true because the iterator is not at the beginning of the list. previous and hasPrevious are ListIterator methods.

`iter.add("Diana");`	Adds an element before the iterator position (ListIterator only). The list is now [Diana, Juliet].

`iter.next();`
`iter.remove();`
	remove removes the last element returned by next or previous. The list is now [Diana].

The `HashSet` and `TreeSet` classes both implement the `Set` interface.

You can form hash sets holding objects of type `String`, `Integer`, `Double`, `Point`, `Rectangle`, or `Color`.

Set implementations arrange the elements so that they can locate them quickly.

You can form tree sets for any class that implements the `Comparable` interface, such as `String` or `Integer`.

When you construct a HashSet or TreeSet, store the reference in a Set variable.

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

or

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

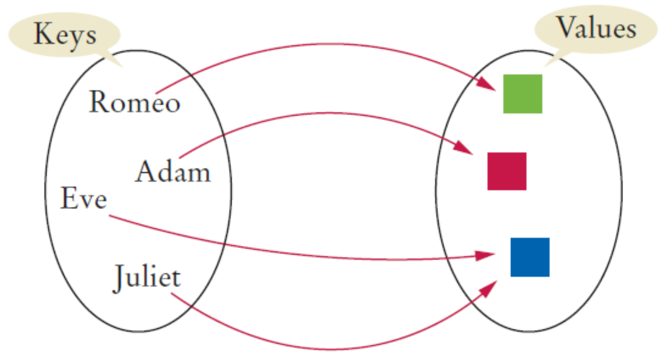
Table 4 Working with Sets

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

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

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

The HashMap and TreeMap classes both implement the Map interface.



A Map

In a *TreeMap*, the key/value associations are stored in a sorted tree, in which they are sorted according to their **keys**. For this to work, it must be possible to compare the keys to one another.

This means either that the keys must implement the interface *Comparable<K>*, or that a *Comparator* must be provided for comparing keys.
(The *Comparator* can be provided as a parameter to the *TreeMap* constructor.)

Note that in a *TreeMap*, as in a *TreeSet*, the `compareTo()` (or `compare()`) method is used to decide whether two keys are to be considered the same.

After constructing a HashMap or TreeMap, you can store the reference to the map object in a Map reference:

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

Table 5 Working with Maps

`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<>();`	Use a `HashMap` if you don’t need to visit the keys in sorted order.

`scores.put("Harry", 90);`
`scores.put("Sally", 95);`
	Adds keys and values to the map.

`scores.put("Sally", 100);`	Modifies the value of an existing key.

`int n = scores.get("Sally");`
`Integer n2 = scores.get("Diana");`
	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.

`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

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

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

Suppose that `map` is a variable of type *Map<K,V>* for some specific types *K* and *V*.

Then the following are some of the methods that are defined for `map`:

- `map.get(key)` — returns the object of type *V* that is associated by the map to the `key`. If the map does not associate any value with `key`, then the return value is `null`. Note that it's also possible for the return value to be `null` when the map explicitly associates the value `null` with the key. Referring to "`map.get(key)`" is similar to referring to "`A[key]`" for an array `A`. (But note that there is nothing like an *IndexOutOfBoundsException* for maps.)
- `map.put(key,value)` — Associates the specified `value` with the specified `key`, where `key` must be of type *K* and `value` must be of type *V*. If the map already associated some other value with the key, then the new value replaces the old one. This is similar to the command "`A[key] = value`" for an array.
- `map.putAll(map2)` — if `map2` is another map of type *Map<K,V>*, this copies all the associations from `map2` into map.
- `map.remove(key)` — if `map` associates a value to the specified `key`, that association is removed from the map.
- `map.containsKey(key)` — returns a boolean value that is `true` if the map associates some value to the specified `key`.
- `map.containsValue(value)` — returns a boolean value that is `true` if the map associates the specified `value` to some key.

If `map` is a variable of type *Map<K,V>*, then

The value returned by `map.keySet()` is a "view" of keys in the map implements the *Set<K>* interface

`map.values()` returns an object of type *Collection<V>* that contains all the values from the associations that are stored in the map. The return value is a *Collection* rather than a *Set* because it can contain duplicate elements.

One of the things that you can do with a *Set* is get an *Iterator* for it and use the iterator to visit each of the elements of the set in turn.

```
Set<Map.Entry<String,Double>> entries = map.entrySet();
Iterator<Map.Entry<String,Double>> entryIter = entries.iterator();
System.out.println("The map contains the following associations:");
while (entryIter.hasNext()) {
   Map.Entry<String,Double> entry = entryIter.next();
   String key = entry.getKey();  // Get the key from the entry.
   Double value = entry.getValue();  // Get the value.
   System.out.println( "   (" + key + "," + value + ")" );
}
```

or, using a for-each loop to avoid some of the ugly type names:

```
for (Map.Entry<String,Double> e : map.entrySet()) {
   System.out.println( "   (" + e.getKey() + "," + e.getValue() + ")" );
}
```

`map.entrySet()` returns a set that contains all the associations from the map.

The elements in the set are objects of type *Map.Entry<K,V>*.

The return type is written as *Set<Map.Entry<K,V>>*.

Each `Map.Entry` object contains one key/value pair, and defines methods `getKey()` and `getValue()` for retrieving the key and the value.

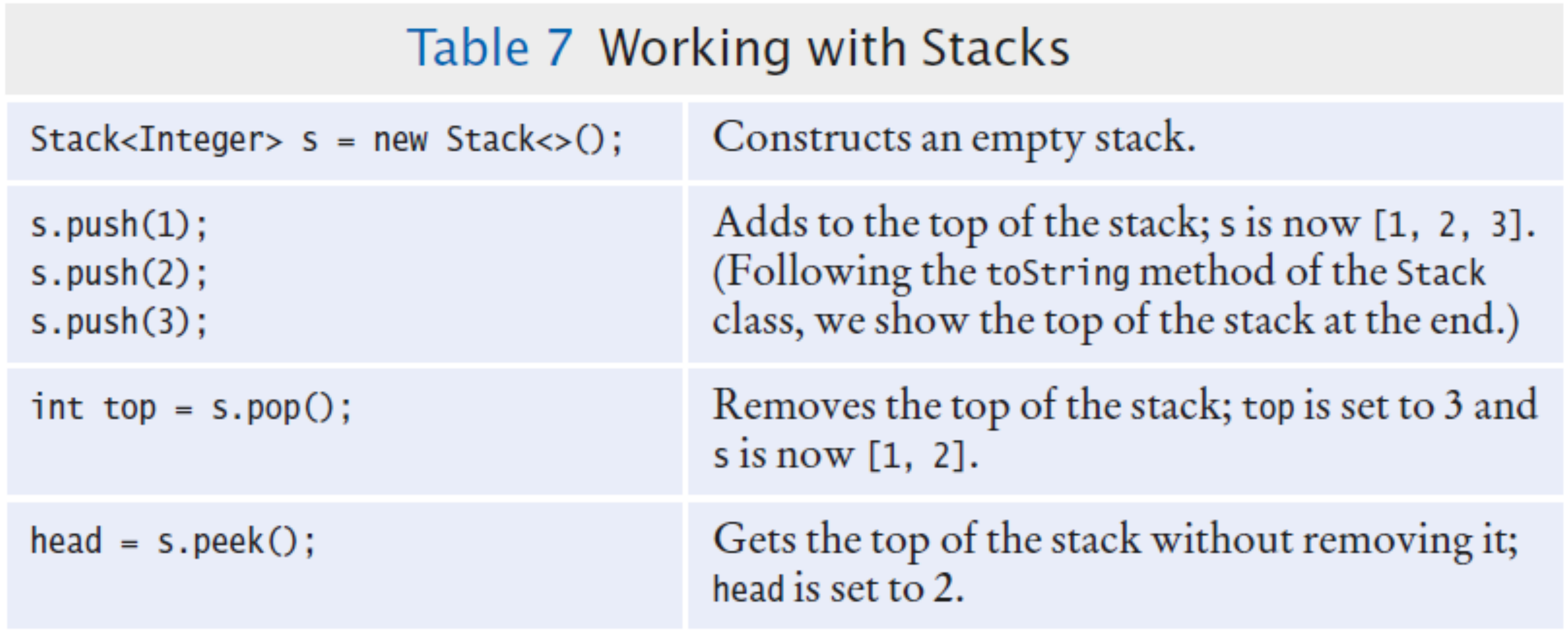


Table 7 Working with Stacks

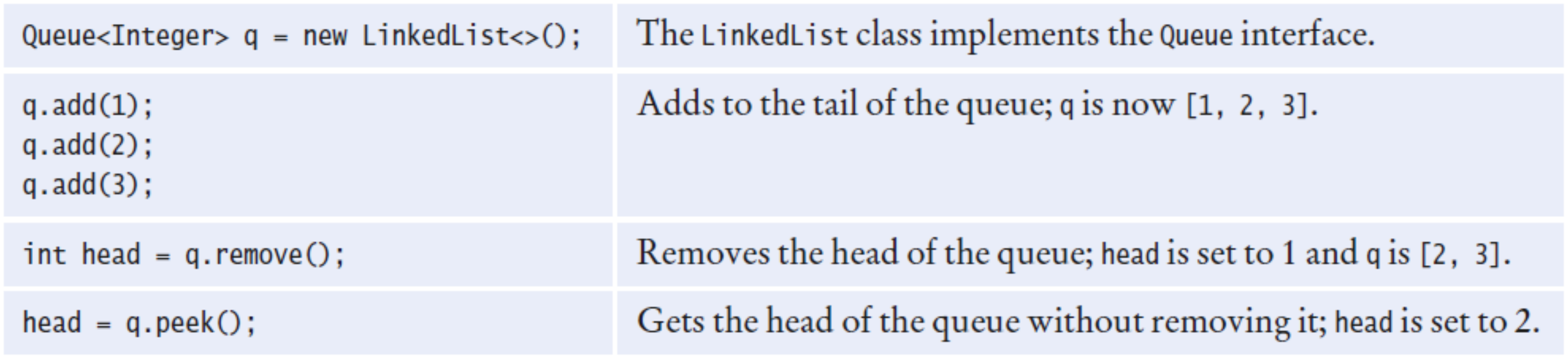
`Stack<Integer> s = new Stack<>();`	Constructs an empty stack.

`s.push(1);`
`s.push(2);`
`s.push(3);`
	Adds to the top of the stack; s is now `[1, 2, 3]`. (Following the `toString` method of the `Stack` class, we show the top of the stack at the end.)

`int top = s.pop();`	Removes the top of the stack; `top` is set to 3 and s is now `[1, 2]`.
`head = s.peek();`	Gets the top of the stack without removing it; `head` is set to 2.

```
if (!s.empty()) …
if (s.size() > 0) …
```

Table 8 Working with Queues



`Queue<Integer> q = new LinkedList<>();`	The `LinkedList` class implements the `Queue` interface.

`q.add(1);`
`q.add(2);`
`q.add(3);`
	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.

`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.

Table 9 Working with Priority Queues

`PriorityQueue<Integer> q = new PriorityQueue<>();`	This priority queue holds `Integer` objects. In practice, you would use objects that describe tasks.
`q.add(3); q.add(1); q.add(2);`	Adds values to the priority queue.
`int first = q.remove();` `int second = q.remove();`	Each call to `remove` removes the most urgent item: `first` is set to 1, `second` to 2.
`int next = q.peek();`	Gets the smallest value in the priority queue without removing it.

Because the priority queue needs to be able to tell which element is the smallest, the added elements should belong to a class that implements the `Comparable` interface. Thus, each removal operation extracts the *minimum* element from the queue.