Collections: Part II 15

Chapter Topics

- An overview of the Java Collections Framework in the `java.util` package: core interfaces and their implementations
- Understanding the functionality provided by the `Collection<E>` interface, and its role in the Java Collections Framework
- How to create and use lists, and how their functionality is defined by the `List<E>` interface and implemented by the classes `Array-List<E>`, `Vector<E>`, and `LinkedList<E>`
- How to create and use sets, and how their functionality is defined by the `Set<E>` interface and implemented by the classes `HashSet<E>` and `LinkedHashSet<E>`
- How to create and use sorted and navigable sets, and how their functionality is defined by the `SortedSet<E>` and `NavigableSet<E>` interfaces, respectively, and implemented by the class `TreeSet<E>`
- How to create and use queues and deques, and how their functionality is defined by the `Queue<E>` and `Deque<E>` interfaces, and implemented by the classes `PriorityQueue<E>` and `ArrayDeque<E>`, respectively
- How to create and use maps, and how their functionality is defined by the `Map<K, V>` interface and implemented by the classes `HashMap<K, V>`, `LinkedHashMap<K, V>`, and `Hashtable<K, V>`
- How to create and use sorted and navigable maps, and how their functionality is defined by the `SortedMap<K, V>` and `NavigableMap<K, V>` interfaces, respectively, and implemented by the class `TreeMap<K, V>`
- Using the utility methods found in the `Collections` and `Arrays` classes, with emphasis on sorting and searching in lists and arrays

Java SE 17 Developer Exam Objectives

[5.1] Create Java arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements	§15.1, p. 783 *to* §15.12, p. 865
❍ *List, set, map, deque, and sorted collections and arrays are covered in this chapter.*	
❍ *For arrays, see* §3.9, p. 117.	
❍ *For* `ArrayList`, *see* Chapter 12, p. 643.	
❍ *For comparing elements, see* §14.4, p. 761, *and* §14.5, p. 769, *respectively.*	

Java SE 11 Developer Exam Objectives	
[5.2] Use a Java array and List, Set, Map and Deque collections, including convenience methods ❍ *List, set, map, and deque collections are covered in this chapter.* ❍ *For arrays, see* §3.9, p. 117. ❍ *For* `ArrayList`, *see* Chapter 12, p. 643.	§15.1, p. 783 *to* §15.9, p. 840
[5.3] Sort collections and arrays using Comparator and Comparable interfaces ❍ *Sorting collections and arrays is covered in this chapter.* ❍ *For* `Comparable` *and* `Comparator` *interfaces, see* §14.4, p. 761, *and* §14.5, p. 769, *respectively.*	§15.5, p. 810 §15.10, p. 845 §15.11, p. 858 §15.12, p. 865

The topics covered in this chapter can be divided into two main parts:

- In-depth coverage of collections, deques, and maps (§15.1 to §15.10)
- Sorting and searching in collections and arrays using the utility methods from the `Collections` and the `Arrays` classes (§15.11 and §15.12)

Some topics related to collections and maps are covered elsewhere in the book:

- The `ArrayList<E>` class is covered in Chapter 12, p. 643.
- Using streams with collections is covered in Chapter 16, p. 879.
- Thread-safe or concurrent collections and maps are covered in Chapter 22, p. 1365.
- Static methods from the utility class `Arrays` are used in many examples throughout the book.

15.1 The Java Collections Framework

A *collection* allows a group of objects to be treated as a single entity. Objects can be stored, retrieved, and manipulated as *elements* of a collection. Arrays are an example of one kind of collection.

Program design often requires the handling of collections of objects. The Java Collections Framework provides a set of standard utility classes for managing various kinds of collections. The core framework is provided in the `java.util` package and comprises three main parts:

- The core *interfaces* that allow collections to be manipulated independently of their implementation (see **Figure 15.1** and **Table 15.1**). These *generic* interfaces define the common functionality exhibited by collections, and facilitate data exchange between collections.

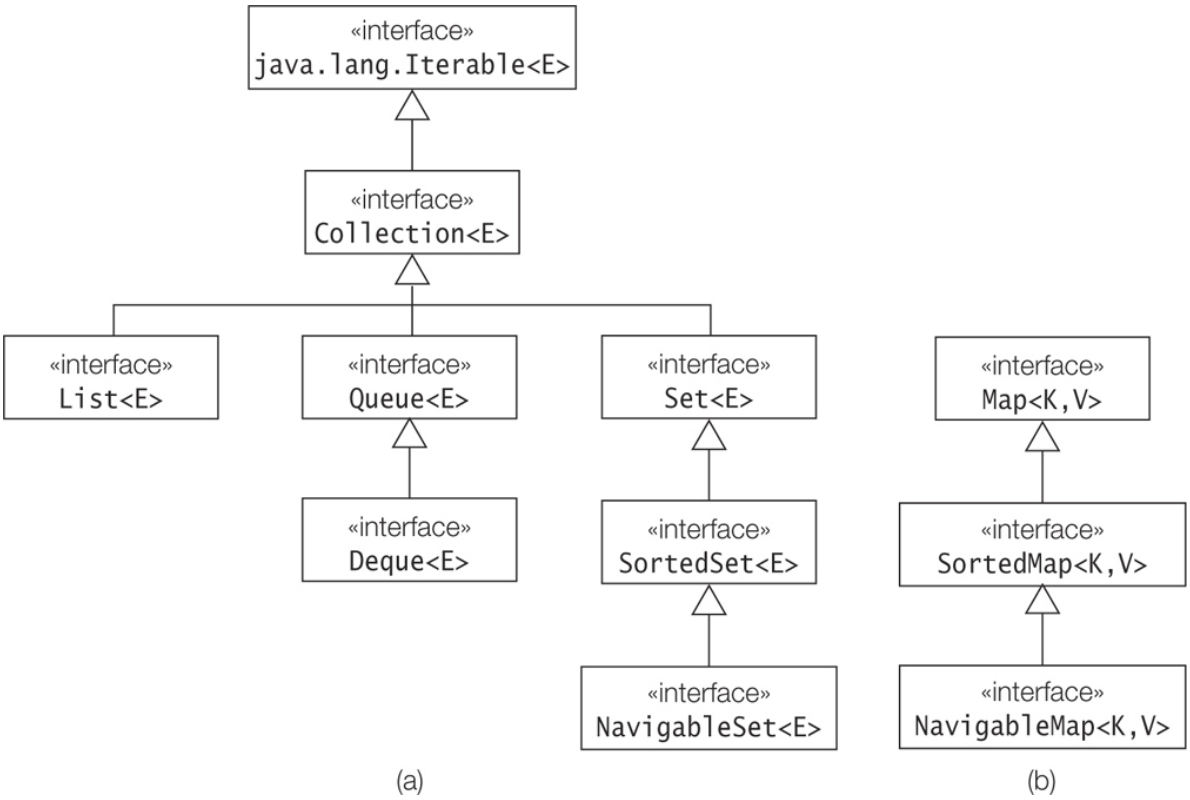

Figure 15.1 *The Core Interfaces*

Table 15.1 *Core Interfaces and Concrete Classes in the Collections Framework*

Interface	Description	Concrete classes
`Collection<E>`	A basic interface that defines the normal operations that allow a collection of objects to be maintained or handled as a single unit.	–
`List<E> extends Collection<E>`	The `List<E>` interface extends the `Collection<E>` interface to maintain a sequence of elements that can contain duplicates.	`ArrayList<E>`, `Vector<E>`, `LinkedList<E>`
`Set<E> extends Collection<E>`	The `Set<E>` interface extends the `Collection<E>` interface to represent its	`HashSet<E>`, `LinkedHashSet<E>`

Interface	Description	Concrete classes
	mathematical namesake: a *set* of unique elements.	
`SortedSet<E>` `extends Set<E>`	The `SortedSet<E>` interface extends the `Set<E>` interface to provide the required functionality for maintaining a set in which the elements are stored in some sort order.	`TreeSet<E>`
`NavigableSet<E>` `extends` `SortedSet<E>`	The `NavigableSet<E>` interface extends and replaces the `SortedSet<E>` interface to maintain a sorted set, and should be the preferred choice in new code.	`TreeSet<E>`
`Queue<E> extends` `Collection<E>`	The `Queue<E>` interface extends the `Collection<E>` need to be processed according to some policy—that is, insertion at one end and removal at the other, usually as FIFO (first in, first out).	`PriorityQueue<E>,` `LinkedList<E>`
`Deque<E> extends` `Queue<E>`	The `Deque<E>` interface extends the `Queue<E>` interface to maintain a queue whose elements can be processed at both ends: *d*ouble-*e*nded *que*ue.	`ArrayDeque<E>,` `LinkedList<E>`
`Map<K,V>`	A basic interface that defines operations for maintaining mappings of keys to values.	`HashMap<K,V>,` `Hashtable<K,V>,` `LinkedHashMap<K,V>`

Interface	Description	Concrete classes
`SortedMap<K,V>` `extends Map<K,V>`	The `SortedMap<K, V>` interface extends the `Map<K, V>` interface for maps that maintain their mappings sorted in key order.	`TreeMap<K,V>`
`NavigableMap<K,V>` `extends` `SortedMap<K,V>`	The `NavigableMap<K, V>` interface extends and replaces the `SortedMap<K, V>` interface for sorted maps.	`TreeMap<K,V>`

- A set of *implementations* (i.e., concrete classes, listed in **Table 15.1**) that are specific implementations of the core interfaces, providing data structures that a program can readily use.
- *Algorithms* that are an assortment of static *utility methods* found in the `Collections` and `Arrays` *classes* that can be used to perform various operations on collections and arrays, such as sorting and searching, or creating customized collections (**§15.11**, **p. 856**, and **§15.12**, **p. 864**).

Core Interfaces

The generic `Collection<E>` interface is a generalized interface for maintaining collections, and is the root of the interface inheritance hierarchy for collections shown in **Figure 15.1a**. These generic subinterfaces are summarized in **Table 15.1**. The `Collection<E>` interface extends the `Iterable<E>` interface that specifies an *iterator* to sequentially access the elements of an `Iterable` object (**p. 791**).

The elements in a `Set<E>` must be unique—that is, no two elements in the set can be equal. The order of elements in a `List<E>` is *positional*, and individual elements can be accessed randomly according to their position in the list.

Queues and deques, represented respectively by the `Queue<E>` and the `Deque<E>` interfaces, define collections whose elements can be processed according to various policies.

As can be seen in **Figure 15.1b**, the `Map<K, V>` interface does not extend the `Collection<E>` interface because conceptually, a map is not a collection. A map does not contain elements. It contains *mappings* (also called *entries*) from a set of *key* objects to a set of *value* objects. A key can, at most, be associated with one value— that is, it must be unique. As the name implies, the `SortedMap<K, V>` interface extends the `Map<K, V>` interface to maintain its mappings sorted in *key order*. It is superseded by the `NavigableMap<K, V>` interface which should be the preferred choice in new code.

Implementations

The `java.util` package provides implementations of a selection of well-known abstract data types, based on the core interfaces. **Figures 15.2** and **15.3** show the inheritance relationship between the core interfaces and the corresponding implementations. None of the concrete implementations inherit directly from the `Collection<E>` interface. The abstract classes that provide the basis on which concrete classes are implemented are not shown in **Figures 15.2** and **15.3**.

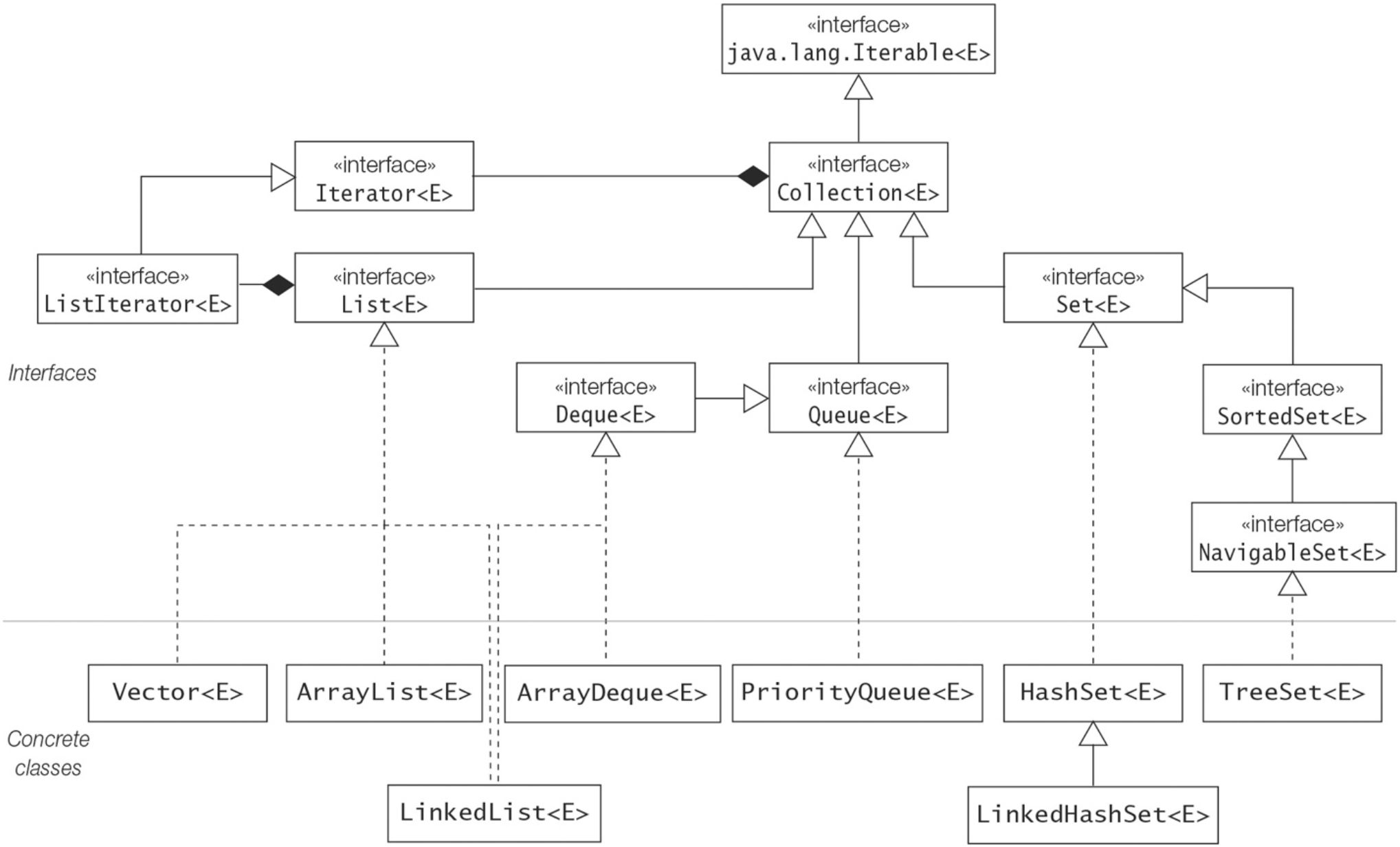
«interface»
java.lang.Iterable<E>
«interface»
Iterator<E>
«interface»
Collection<E>
«interface»
ListIterator<E>
«interface»
List<E>
«interface»
Set<E>
Interfaces
«interface»
Deque<E>
«interface»
Queue<E>
«interface»
SortedSet<E>
«interface»
NavigableSet<E>
Vector<E>
ArrayList<E>
ArrayDeque<E>
PriorityQueue<E>
HashSet<E>
TreeSet<E>
Concrete
classes
LinkedList<E>
LinkedHashSet<E>

Figure 15.2 *The Core Collection Interfaces and Their Implementations*

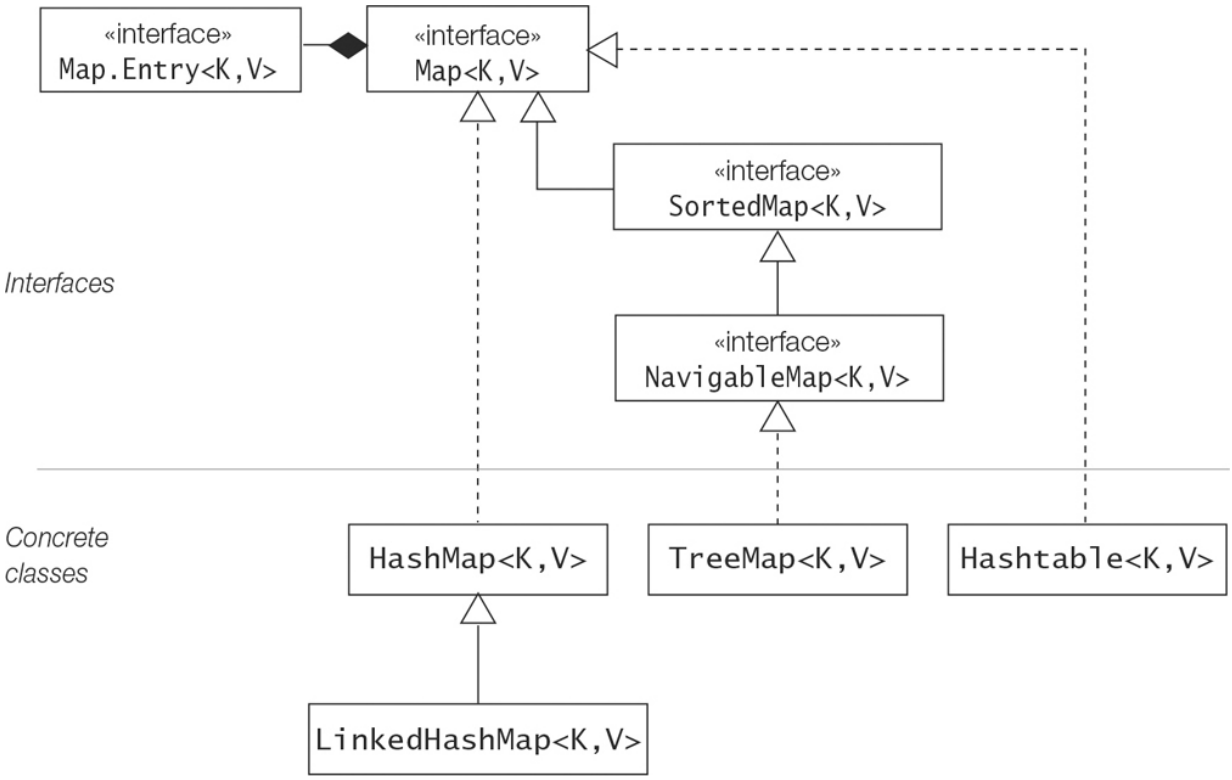

Figure 15.3 *The Core Map Interfaces and Their Implementations*

By convention, each of the collection implementation classes provides a constructor for creating a collection based on the elements of another `Collection` object passed as an argument. This allows the implementation of a collection to be changed by merely passing the collection to the constructor of the desired implementation. This interchangeability is also true between `Map` implementations. But collections and maps are not interchangeable. Note that a collection (or a map) only stores reference values of objects, and not the actual objects.

The collections framework is *interface-based*, meaning that collections are manipulated according to their interface types, rather than by the implementation types. By using these interfaces wherever collections of objects are used, various implementations can be used interchangeably.

All the concrete classes shown in **Figures 15.2** and **15.3** implement the `Serializable` and the `Cloneable` interfaces; therefore, the objects of these classes can be serialized and also cloned.

A summary of collection and map implementations is given in **Table 15.2**. The contents of this table will be the focus as each core interface and its corresponding implementations are discussed in the subsequent sections.

Table 15.2 *Summary of Collection and Map Implementations*

Concrete collections and maps	Interface	Duplicates	Ordered/Sorted	Methods the implementations expect the elements to provide	Data structures on which implementation is based
`ArrayList<E>`	`List<E>`	Allowed	Insertion order	`equals()`	Resizable array
`LinkedList<E>`	`List<E>`, `Deque<E>`	Allowed	Insertion/priority/deque order	`equals()`	Linked list
`Vector<E>`	`List<E>`	Allowed	Insertion order	`equals()`	Resizable array
`HashSet<E>`	`Set<E>`	Unique elements	No order	`equals()`, `hashCode()`	Hash table
`LinkedHashSet<E>`	`Set<E>`	Unique elements	Insertion order	`equals()`, `hashCode()`	Hash table and doubly linked list
`TreeSet<E>`	`NavigableSet<E>`	Unique elements (not `null`)	Sort order	`equals()`, `hashCode()`, `compareTo()`	Balanced tree
`PriorityQueue<E>`	`Queue<E>`	Allowed (not `null`)	Access according to priority order	`equals()`, `compareTo()`	Priority heap (tree-like structure)
`ArrayDeque<E>`	`Deque<E>`	Allowed (not `null`)	Deque order	`equals()`	Resizable array

Concrete collections and maps	Interface	Duplicates	Ordered/Sorted	Methods the implementations expect the elements to provide	Data structures on which implementation is based
`HashMap<K,V>`	`Map<K,V>`	Unique keys	No order	`equals()`, `hashCode()`	Hash table
`LinkedHashMap<K,V>`	`Map<K,V>`	Unique keys	Key insertion order/Access order of entries	`equals()`, `hashCode()`	Hash table and doubly linked list
`Hashtable<K,V>`	`Map<K,V>`	Unique keys (no `null` key and no `null` values)	No order	`equals()`, `hashCode()`	Hash table
`TreeMap<K,V>`	`NavigableMap<K,V>`	Unique keys (no `null` key)	Sorted in key order	`equals()`, `hashCode()`, `compareTo()`	Balanced tree

In **Table 15.2**, we see that only lists and queues allow *duplicates*—that is, the same value can be stored multiple times in the collection. The `TreeSet<E>` class and the queue classes `PriorityQueue<E>` and `ArrayDeque<E>` do *not* allow the `null` value to be stored in the collection. All maps require that the keys are *unique*—that is, no two keys can be equal, and only the classes `Hashtable<K, V>` and `TreeMap<K, V>` disallow the `null` value as a key. A `Hashtable<K, V>`, in addition, does not allow the `null` value to be associated with a key.

From **Table 15.2**, we also see that elements in a `LinkedHashSet<E>` are ordered, in a `TreeSet<E>` they are sorted, and in a `HashSet<E>` they have no order (i.e., they are *unordered*). *Sorting implies ordering* the elements in a collection according to some ranking criteria, usually based on the *values* of the elements. However, elements in an `ArrayList<E>` are maintained in the order they are inserted in the list, known as the *insertion order*. The el-

ements in such a list are thus *ordered*, but they are *not* sorted, as it is not the values of the elements that determine their ranking in the list but their position in the list. Thus ordering does *not* necessarily imply sorting. In a `HashSet<E>`, the elements are unordered. No ranking of elements is implied in such a set. Whether a collection is sorted, ordered, or unordered also has implications when iterating over the collection (**p. 791**).

In order for sorting and searching to work properly, the collections and maps in **Table 15.2** also require that their elements implement the appropriate methods for object comparison. Comparing objects is covered in **Chapter 14**, **p. 741**. Sorting and searching in collections is covered in **§15.11**, **p. 856**.

Methods defined for collections that specify functional interfaces as a parameter and require lambda expressions as arguments are covered in **Chapter 13**, **p. 673**.

Using streams with collections is covered in **Chapter 16**, **p. 879**.

The collection and map implementations discussed in this chapter, except for `Vector<E>` and `Hashtable<K, V>`, are not *thread-safe*—that is, their integrity can be jeopardized by concurrent access. Concurrent collections and maps are covered in **Chapter 22**, **p. 1365**.

The Java Collections Framework provides a plethora of collections and maps for use in single-threaded and concurrent applications, which should eliminate the need to implement one from scratch.

15.2 Collections

The `Collection<E>` interface specifies the contract that all collections should implement. Some of the operations in the interface are *optional*, meaning that a collection may choose to provide a stub implementation of such an operation that throws an `UnsupportedOperationException` when invoked—for example, when a collection is unmodifiable. The implementations of collections from the `java.util` package support all the optional operations in the `Collection<E>` interface (see **Figure 15.2** and **Table 15.2**).

Many of the methods return a `boolean` value to indicate whether the collection was modified as a result of the operation.

Basic Operations

The basic operations are used to query a collection about its contents and allow elements to be added to and removed from a collection. Many examples in this chapter make use of these operations.

Click here to view code image

```
int size()
boolean isEmpty()
```

```
boolean contains(Object element)
boolean add(E element)                Optional
boolean remove(Object element)        Optional
```

The `size()` method returns the number of elements in the collection, and the `isEmpty()` method determines whether there are any elements in the collection.

The `contains()` method checks for membership of the argument object in the collection, using object value equality.

The `add()` and `remove()` methods return `true` if the collection was modified as a result of the operation.

By returning the value `false`, the `add()` method indicates that the collection excludes duplicates, and that the collection already contains an object equal to the argument object.

Note that we can only add an object of a specific type (`E`). However, a collection allows us to determine whether it has an element equal to an arbitrary object, or remove an element that is equal to an arbitrary object.

Bulk Operations

These operations are performed on a collection as a single entity. See §15.4, p. 804, for an example.

Click here to view code image

```
boolean containsAll(Collection<?> c)
boolean addAll(Collection<? extends E> c)       Optional
boolean removeAll(Collection<?> c)              Optional
boolean retainAll(Collection<?> c)              Optional
void    clear()                                 Optional
```

These bulk operations can be used to perform the equivalent of set logic on *arbitrary collections* (i.e., also lists and not just sets). The `containsAll()` method returns `true` if all elements of the specified collection are also contained in the current collection.

The `addAll()`, `removeAll()`, and `retainAll()` methods are *destructive* in the sense that the collection on which they are invoked can be modified. The operations performed by these methods are visualized by Venn diagrams in Figure 15.4.

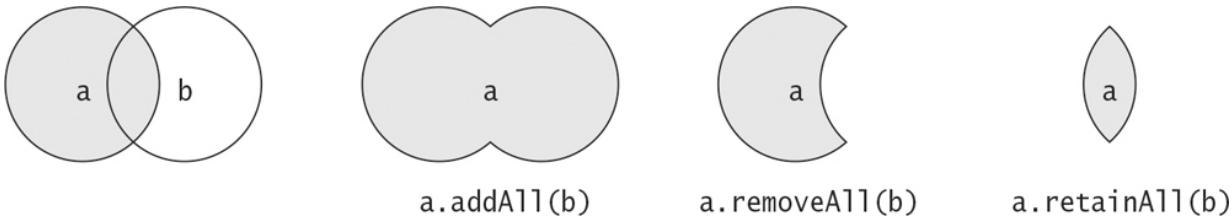

Figure 15.4 *Bulk Operations on Collections*

The `addAll()` method requires that the element type of the other collection is the same as, or a subtype of, the element type of the current collection. The `removeAll()` and `retainAll()` operations can be performed with collections of any type.

The `clear()` method removes all elements in the collection so that the collection is empty.

Iterating Over a Collection

A common operation on a collection is to iterate over all elements in a collection in order to perform a particular operation on each individual element. We explore several ways in which this operation can be implemented:

- Using the `hasNext()` and `next()` methods of the `Iterator<E>` interface (p. 793) to access the elements in the underlying collection
- Using the `forEachRemaining()` default method of the `Iterator<E>` interface to access the elements in the underlying collection
- Using the enhanced `for(:)` loop to iterate over an object that implements the `Iterable<E>` interface (see below)
- Using the `forEach()` default method of the `Iterable<E>` interface
- Using a sequential or parallel stream to iterate over a collection (p. 798)

The `Iterable<E>` interface defines the methods shown below. The enhanced `for(:)` loop can be used to iterate over objects that implement this interface. The `Collection<E>` interface extends the `Iterable<E>` interface, making all collections iterable.

Click here to view code image

```
Iterator<E> iterator()                                  From Iterable<E> interface.
```

Returns an iterator that can be used to iterate over the elements in this collection. Any guarantee regarding the order in which the elements are returned is provided by the individual collection.

Click here to view code image

```
default void forEach(Consumer<? super T> action)  From Iterable<E> interface.
```

With this method, the specified `action` is performed on each of the remaining elements of an `Iterable`, unless an exception is thrown, which is then relayed to the caller. The functional interface `Consumer<T>` is covered in §13.7, p. 709.

Example 15.1 illustrates various ways to iterate over a collection that is created at (1). The example uses an `ArrayList<E>` for a collection of names that are `String`s. First an empty `ArrayList` is created and names are added to the collection. The operation performed on the collection prints each name in uppercase.

Example 15.1 *Iterating Over a Collection*

Click here to view code image

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;

public class IterationOverCollections {
  public static void main(String[] args) {

    // Create a collection of names.                                    (1)
    Collection<String> collectionOfNames = new ArrayList<>();
    Collections.addAll(collectionOfNames, "Alice", "Bob", "Tom","Dick","Harriet");
    System.out.println(collectionOfNames);

    // Using an explicit iterator.
    Iterator<String> iterator = collectionOfNames.iterator();        // (2)
    while (iterator.hasNext()) {                                     // (3)
      String name = iterator.next();                                 // (4)
      System.out.print(name.toUpperCase() + " ");
    }
    System.out.println();

    // Using the forEachRemaining() method of the Iterator interface.
    iterator = collectionOfNames.iterator();                         // (5)
    iterator.forEachRemaining(name ->                                // (6)
        System.out.print(name.toUpperCase() + " "));
    System.out.println();

    // Using the for(:) loop on an Iterable.
    for (String name : collectionOfNames) {                          // (7)
      System.out.print(name.toUpperCase() + " ");
    }
    System.out.println();

    // Using the forEach() method of the Collection interface.
    collectionOfNames.forEach(name ->                                // (8)
        System.out.print(name.toUpperCase() + " "));
```

```
    System.out.println();

    // Filtering using an explicit iterator.
    iterator = collectionOfNames.iterator();                          // (9)
    while (iterator.hasNext()) {                                      // (10)
      String name = iterator.next();                                  // (11)
      if (name.length() == 3)
        iterator.remove();                                            // (12)
    }
    System.out.println(collectionOfNames);

    // Filtering using the removeIf() method of the Collection interface.
    collectionOfNames.removeIf(name -> name.startsWith("A"));         // (13)
        System.out.println(collectionOfNames);

    // Using the stream() method of the Collection interface.
    collectionOfNames.stream()                                        // (14)
                     .forEach(name -> System.out.print(name.toUpperCase() + " "));
    System.out.println();
  }
}
```

Output from the program:

Click here to view code image

```
[Alice, Bob, Tom, Dick, Harriet]
ALICE BOB TOM DICK HARRIET
ALICE BOB TOM DICK HARRIET
ALICE BOB TOM DICK HARRIET
ALICE BOB TOM DICK HARRIET
[Alice, Dick, Harriet]
[Dick, Harriet]
DICK HARRIET
```

Using an Explicit Iterator on a Collection

A collection provides an iterator which allows sequential access to the elements of a collection. An iterator can be obtained by calling the `iterator()` method of the `Collection<E>` interface. An iterator is defined by the generic interface `java.util.Iterator<E>` that has the following methods:

Click here to view code image

```
boolean hasNext()                                  From Iterator<E> interface.
```

Returns `true` if the underlying collection still has elements left to return. A future call to the `next()` method will return the next element from the collection.

Click here to view code image

```
E next()                                                    From Iterator<E> interface.
```

Moves the iterator to the next element in the underlying collection, and returns the current element. If there are no more elements left to return, it throws a `NoSuchElementException`.

Click here to view code image

```
void remove()                  Optional       From Iterator<E> interface.
```

Removes the element that was returned by the last call to the `next()` method from the underlying collection. Invoking this method results in an `Illegal-StateException` if the `next()` method has not yet been called, or when the `remove()` method has already been called after the last call to the `next()` method. This method is optional for an iterator—that is, it throws an `Unsupported-OperationException` if the remove operation is not supported.

Click here to view code image

```
default void
    forEachRemaining(Consumer<? super E> action) From Iterator<E> interface.
```

The specified `action` is performed on each of the remaining elements in the collection associated with an iterator, unless an exception is thrown during its execution. The functional interface `Consumer<T>` is covered in §13.7, p. 709.

After obtaining the iterator for a collection, the methods provided by the `Iterator<E>` interface can be used systematically to iterate over the elements of the underlying collection one at a time.

In **Example 15.1**, an explicit iterator is obtained at (2) and used in the loop at (3) to iterate over all elements in the underlying collection. A call to the `hasNext()` method in the condition of the loop ensures that there is still an element to retrieve from the collection. At (4) the current element is retrieved by the iterator from the collection by calling the `next()` method. No casts are necessary at (4), as the compiler guarantees that the iterator will return a `String` object from the underlying collection.

Click here to view code image

```
Iterator<String> iterator = collectionOfNames.iterator();        // (2)
while (iterator.hasNext()) {                                     // (3)
  String name = iterator.next();                                 // (4)
  System.out.print(name.toUpperCase() + " ");
}
```

Note that the methods are invoked on the iterator, not the underlying collection. In **Example 15.1**, the `hasNext()` method is called *before* the `next()` method to ensure that there is still an element remaining to be processed; otherwise, a `java.util.NoSuchElementException` can be raised at runtime. Using an explicit iterator puts this responsibility on the user code.

In **Example 15.1**, we used an iterator in a `while` loop at (3) for iterating over the collection. It is quite common to use an iterator in a `for(;;)` loop for this purpose, where the iterator is obtained in the initialization part, and the increment part is left empty:

Click here to view code image

```
for (Iterator<String> iterator = collectionOfNames.iterator();
     iterator.hasNext(); /* Empty increment. */) {
  String name = iterator.next();
  System.out.print(name.toUpperCase() + " ");
}
```

The majority of the iterators provided in the `java.util` package are said to be *fail-fast*. When an iterator has already been obtained, structurally modifying the underlying collection by other means will invalidate the iterator. Subsequent use of this iterator will throw a `ConcurrentModificationException`, as the iterator checks to see if the collection has been structurally modified every time it accesses the collection. The `remove()` method of an iterator is the only recommended way to delete elements from the underlying collection during iteration with an iterator (**p. 796**).

The order in which the iterator will return the elements from an underlying collection depends on the iteration order supported by the collection. For example, an iterator for a list will iterate over the elements in the sequential order they have in the list, whereas the iteration order for the elements in an ordinary set is not predetermined. An iterator for a sorted collection will make the elements available in a given sort order. Iteration order will be discussed together with the individual concrete collection classes.

Using the `forEachRemaining()` *Method*

A convenient way to perform a task on each element of a collection is to use the `forEachRemaining()` method of the `Iterator<E>` interface. This method requires a consumer as an argument (**§13.7**, **p. 709**). The lambda expression defining the consumer is executed for each *remaining* element of the collection. In **Example 15.1**, a new iterator is obtained at (5) as the previous one was exhausted, before calling the method at (6). The lambda expression passed as an argument prints the name in uppercase (postfixed with a space for readability in the output). Note that the method is invoked on the iterator, and not directly on the collection.

Click here to view code image

```
iterator = collectionOfNames.iterator();                    // (5)
iterator.forEachRemaining(name ->                           // (6)
```

```
      System.out.print(name.toUpperCase() + " "));
```

In **Example 15.1**, the `toString()` method of the collection class is used implicitly in print statements to generate a text representation for the collection. The collection classes override the `Object.toString()` method to provide a text representation of their contents. The standard text representation generated by the `toString()` method for a collection is

`[` *element* $_1$ `,` *element* $_2$ `, ..., ` *element* $_n$ `]`

where each *element*$_i$, where `1 <= i <= n`, is the text representation generated by the `toString()` method of the individual elements in the collection.

Using the `for(:)` *Loop to Iterate Over a Collection*

A `for(:)` loop can be used to iterate over an array or a data structure that implements the `java.lang.Iterable<E>` interface. This interface requires the implementation of the `iterator()` method to obtain an iterator that is used behind the scenes to iterate over elements of the data structure in a `for(:)` loop.

In the `for(:)` loop construct

`for` (*type variable* `:` *expression* `)` *statement*

the value of *expression* can be a reference value that refers to a collection that implements the `Iterable<E>` interface. In **Figure 15.2** we see that the `Collection<E>` interface extends the `Iterable<E>` interface, and therefore, all collections that implement the `Collection<E>` interface can be iterated over using the `for(:)` loop. A collection that implements the `Collection<E>` interface and thereby the `Iterable<E>` interface has the element type `E`. This element type `E` must be assignable to the *type* of the *variable* in the `for(:)` loop. The *variable* is assigned the reference value of a new element in the collection each time the body of the loop is executed.

The semantics of the `for(:)` loop also apply when iterating over a collection. In particular, any structural change to the collection (adding or removing elements) in the `for(:)` loop will result in a `ConcurrentModificationException`—in other words, the underlying collection is structurally immutable.

Example 15.1 illustrates using a `for(:)` loop to iterate over a collection. The collection of names is iterated over in the `for(:)` loop at (7), printing each name in uppercase.

Click here to view code image

```
for (String name : collectionOfNames) {                          // (7)
  System.out.print(name.toUpperCase() + " ");
}
```

Behind the scenes, however, an appropriate iterator is used to iterate over the collection, but the `for(:)` loop simplifies iterating over a collection in the source code.

Note that if the collection is ordered or sorted, the iterator will iterate over the collection in the ordering used to maintain the elements in the collection. For example, in the case of an `ArrayList`, the iterator will yield the elements in the same order as the insertion order. In the case of a `TreeSet<E>`, the iterator will yield the elements in the sort order used to maintain the elements in the set. If the collection is unordered, the order in which the iterator will yield the elements is not predictable. Thus we cannot be sure in which order the elements of a `HashSet` will be iterated by the `for(:)` loop.

Using the `forEach()` *Method to Iterate Over a Collection*

The `default` method `forEach()` of the `Iterable<E>` interface (which the `Collection<E>` interface implements) allows us to do away with an explicit iterator in order to perform an operation on each element of the collection. The method requires a consumer as the argument (§13.7, p. 709). The lambda expression defining the consumer is executed successively for each element of the collection.

In **Example 15.1**, the lambda expression passed as the argument at (8) prints each name in uppercase. Note that the method is invoked directly on the collection.

Click here to view code image

```
collectionOfNames.forEach(name ->
    System.out.print(name.toUpperCase() + " "));                // (8)
```

Filtering

We illustrate here two ways of filtering a collection—in this case, removing elements from the collection that satisfy certain criteria.

Using the `remove()` *Method of an Iterator to Filter a Collection*

Example 15.1 shows how an explicit iterator can be used in a loop to *remove* elements from a collection that fulfills certain criteria. As we have seen earlier, an explicit iterator is obtained at (9) and used in the loop at (10). The order of calls to the `hasNext()` and `next()` methods on the iterator at (10) and (11), respectively, ensures that each element in the underlying collection is retrieved in the loop. The call to the `remove()` method in the loop at (12) removes the current element from the underlying collection, if the criteria in the `if` statement is satisfied.

Click here to view code image

```
// [Alice, Bob, Tom, Dick, Harriet]
iterator = collectionOfNames.iterator();                        // (9)
```

```
while (iterator.hasNext()) {                                    // (10)
  String name = iterator.next();                                // (11)
  if (name.length() == 3)
    iterator.remove();                                          // (12)
}
// [Alice, Dick, Harriet]
```

Best practices dictate that the three methods of the iterator should be used *in lockstep* inside a loop, as shown in **Example 15.1**. In particular, the `next()` method *must* be called *before* the `remove()` method for each element in the collection; otherwise, a `java.lang.IllegalStateException` or an `UnsupportedOperationException` is raised at runtime, depending on whether the collection provides this optional operation or not. As noted earlier, using an explicit iterator places the responsibility of bookkeeping on the user code.

Using the `removeIf()` *Method to Filter a Collection*

The following `default` method of the `Collection<E>` interface makes the task of filtering a collection much easier, as no explicit iteration is necessary:

Click here to view code image

```
default boolean removeIf(Predicate<? super E> filter)
```

All elements that satisfy the predicate defined by the specified `filter` are removed from this collection. The method returns `true` or `false` depending on whether the collection was modified or not. Predicates are covered in §13.6, p. 703.

In **Example 15.1**, the call to the `removeIf()` method at (13) takes a `Predicate` to test whether a name starts with the letter `A`, resulting in those names satisfying the predicate to be removed from the collection.

Click here to view code image

```
// [Alice, Dick, Harriet]
collectionOfNames.removeIf(name -> name.startsWith("A"));       // (13)
// [Dick, Harriet]
```

Ample examples of filtering can also be found in §13.3, p. 691, and §16.5, p. 910.

Streams

The two methods of the `Collections` interface shown below allow a collection to be used as the *source of a stream*. A stream makes the elements of a collection available as a *sequence* of

elements on which *sequential* and *parallel aggregate operations* can be performed.

Click here to view code image

```
default Stream<E> stream()
default Stream<E> parallelStream()
```

Return a sequential stream or a possibly parallel stream with this collection as its source, respectively.

Example 15.1 illustrates at (14) how a stream can be used to iterate over the collection of names in order to print them in uppercase.

Click here to view code image

```
collectionOfNames.stream()                                    // (14)
                 .forEach(name -> System.out.print(name.toUpperCase() + " "));
```

Details are revealed in §16.4, p. 897, which is devoted entirely to streams, providing many examples of using streams on collections.

Array Operations

The following operations convert collections to arrays. See also array operations on array lists (§12.6, p. 658).

```
Object[] toArray()
<T> T[] toArray(T[] a)
```

The first `toArray()` method returns an array of type `Object` filled with all the elements of the collection. The second method is a generic method that stores the elements of a collection in an array of type `T`. The default method uses the `generator` function to allocate the returned array.

If the given array is big enough, the elements are stored in this array. If there is room to spare in the array; that is, the length of the array is greater than the number of elements in the collection—the spare room is filled with `null` values before the array is returned. If the array is too small, a new array of type `T` and appropriate size is created. If `T` is not a supertype of the runtime type of every element in the collection, an `ArrayStoreException` is thrown.

Click here to view code image

```
default <T> T[] toArray(IntFunction<T[]> generator)
```

Allows creation of an array of a particular runtime type given by the parameterization of the type parameter `T[]`, using the specified `generator` function (§13.8, p. 717) to allocate the array of the desired type and the specified length. The `default` implementation calls the `generator` function with 0 and then passes the resulting array of length 0 to the `toArray(T[])` generic method.

Example 15.2 illustrates converting a set to an array. At (1), the call to the non-generic `toArray()` method returns an array of type `Object`. Since an array of type `Object` is not a subtype of an array of type `String`, the compiler reports an error at (2).

At (3), the call to the generic `toArray()` method returns an array of size 3 and type `Object[]`, but element type `String`, when the method was passed a zero-length array of type `Object`. In other words, the method created a suitable-size array of type `Object`, since the array passed in the argument was too small. This array was filled with the elements of the set, which are strings. Although the array returned is of type `Object`, the objects stored in it are of type `String`. The output from the program confirms these observations.

At (4), the call to the generic `toArray()` method returns an array of size 3 and type `String[]`, having element type `String`, when the method was passed a zero-length array of type `String`. Now the method creates a new suitable-size array of type `String` and fills it with the elements of the set, which are strings. The output from the program shows that the array passed in the argument is not the same as the array returned by the method call.

At (5), the call to the generic `toArray()` method returns the same array it was passed in the argument, since it is of appropriate size and type. In other words, the array passed in the argument is filled with the elements of the list, and returned. This is corroborated by the output from the program.

At (6), the actual type parameter is `Integer`. The generic `toArray()` method throws an `ArrayStoreException` because `String` objects in the set denoted by `strSet` cannot be stored in an array of type `Integer`.

Lastly, Example 15.2 illustrates converting a set to an array using the default `toArray(IntFunction<T[]>)` generic method, which is a convenience method that actually leverages the `toArray(T[])` generic method.

Click here to view code image

```
IntFunction<String[]> createStrArray = nn -> new String[nn];           // (7)
String[] strArray5 = strSet.toArray(createStrArray);                   // (8a)
String[] strArray6 = strSet.toArray(String[]::new);                    // (8b)
String[] strArray7 = strSet.toArray(createStrArray.apply(0));          // (8c)
```

The lambda expression at (7) creates an array of length `nn` and of type `String[]`. This lambda expression is passed to the `toArray(IntFunction<String[]>)` method at (8a). Equivalently, we can pass a method reference to the method, as shown at (8b). Of course, the lambda expression at (7) only creates an array when the `apply()` method of the `IntFunction<T[]>` interface is called with a value for the length of the array. The default behavior of the `toArray(IntFunction<T[]>)` generic method is illustrated at (8c): The `apply()` method of the `IntFunction<String[]>` interface is explicitly called with the value `0` to create a zero-length `String` array which is then passed as a parameter to the `toArray(String[])` method.

Example 15.2 *Converting Collections to Arrays*

Click here to view code image

```
import java.util.Arrays;
import java.util.Collection;
import java.util.HashSet;
import java.util.function.IntFunction;

public class CollectionToArray {
  public static void main(String[] args) {

    Collection<String> strSet = new HashSet<>();
    strSet.add("2021"); strSet.add("2022"); strSet.add("2023");
    int n = strSet.size();

    Object[] objects = strSet.toArray();                // (1)
//  String[] string = strSet.toArray();                 // (2) Compile-time error!

    Object[] objArray = strSet.toArray(new Object[0]);                    // (3)
    System.out.println("Array size: " + objArray.length);
    System.out.println("Array type: " + objArray.getClass().getName());
    System.out.println("Actual element type: " +
                       objArray[0].getClass().getName());

    String[] strArray1 = new String[0];
    String[] strArray2 = strSet.toArray(strArray1);                       // (4)
    System.out.println("strArray1 == strArray2: " + (strArray1 == strArray2));

    String[] strArray3 = new String[n];
    String[] strArray4 = strSet.toArray(strArray3);                       // (5)
    System.out.println("strArray3 == strArray4: " + (strArray3 == strArray4));

//  Integer[] intArray = strSet.toArray(new Integer[n]);     // (6) Runtime error!

    IntFunction<String[]> createStrArray = nn -> new String[nn];          // (7)
    String[] strArray5 = strSet.toArray(createStrArray);                  // (8a)
    String[] strArray6 = strSet.toArray(String[]::new);                   // (8b)
    String[] strArray7 = strSet.toArray(createStrArray.apply(0));         // (8c)
    System.out.println("strArray5: " + Arrays.toString(strArray5));
```

```
    System.out.println("strArray6: " + Arrays.toString(strArray6));
    System.out.println("strArray7: " + Arrays.toString(strArray7));
  }
}
```

Output from the program:

Click here to view code image

```
Array size: 3
Array type: [Ljava.lang.Object;
Actual element type: java.lang.String
strArray1 == strArray2: false
strArray3 == strArray4: true
strArray5: [2023, 2022, 2021]
strArray6: [2023, 2022, 2021]
strArray7: [2023, 2022, 2021]
```

15.3 Lists

Lists are collections that maintain their elements *in order* and can contain duplicates. The elements in a list are *ordered*. Each element, therefore, has a position in the list. A zero-based index can be used to access the element at the position designated by the index value. The position of an element can change as elements are inserted or deleted from the list—that is, as the list is changed structurally.

The `List<E>` Interface

In addition to the operations inherited from the `Collection<E>` interface, the `List<E>` interface also defines operations that work specifically on lists: position-based access of the list elements, searching in a list, operations on parts of a list (called *open range-view* operations), and applying an operator to each element. These list operations are covered in **Chapter 12**, **p. 643**.

Sorting and searching in lists is covered in **§15.11**, **p. 856**.

The `List<E>` interface also provides the overloaded static method `of()` to create *unmodifiable* lists (**§12.2**, **p. 649**).

Additionally, the `List<E>` interface provides two customized list iterators:

Click here to view code image

```
ListIterator<E> listIterator()
ListIterator<E> listIterator(int index)
```

The iterator from the first method iterates over the elements consecutively, starting with the first element of the list, whereas the iterator from the second method starts iterating over the list from the element designated by the specified index.

The declaration of the `ListIterator<E>` interface is shown below:

Click here to view code image

```
interface ListIterator<E> extends Iterator<E> {
  boolean hasNext();
  boolean hasPrevious();
  E next();               // Element after the cursor
  E previous();           // Element before the cursor
  int nextIndex();        // Index of element after the cursor
  int previousIndex();    // Index of element before the cursor
  void remove();          // Optional
  void set(E o);          // Optional
  void add(E o);          // Optional
}
```

The `ListIterator<E>` interface is a bidirectional iterator for lists. It extends the `Iterator<E>` interface and allows the list to be iterated over in either direction. When iterating over lists, it can be helpful to imagine a *cursor* moving forward or backward *between* the elements when calls are made to the `next()` and the `previous()` methods, respectively. The element that the cursor passes over is returned. When the `remove()` method is called, the element last passed over is removed from the list.

The `ArrayList<E>`, `LinkedList<E>`, and `Vector<E>` Classes

Three implementations of the `List<E>` interface are provided in the `java.util` package: `ArrayList<E>`, `LinkedList<E>`, and `Vector<E>`.

The `ArrayList<E>` class implements the `List<E>` interface (Chapter 12, p. 643). The `Vector<E>` class is a legacy class that has been retrofitted to implement the `List<E>` interface, and will not be discussed in detail. The `Vector<E>` and `ArrayList<E>` classes are implemented using dynamically resizable arrays, providing fast *random access by index* (i.e., position-based access) and fast list iteration—very much like using an ordinary array. Unlike the `ArrayList<E>` class, the `Vector<E>` class is thread-safe, meaning that concurrent calls to the vector will not compromise its integrity. The `LinkedList<E>` implementation uses a doubly linked list. Insertions and deletions in a doubly linked list are very efficient.

The `ArrayList<E>` and `Vector<E>` classes offer comparable performance, but `Vector`s suffer a performance penalty due to synchronization. Position-based access has constant-time performance for the `ArrayList<E>` and `Vector<E>` classes. However, position-based access is in linear time for a `LinkedList<E>`, owing to iteration in a doubly linked list. When fre-

quent insertions and deletions occur inside a list, a `LinkedList<E>` can be worth considering. In most cases, the `ArrayList<E>` implementation is the overall best choice for implementing lists.

In addition to the `List<E>` interface, the `LinkedList<E>` class also implements two other interfaces that allow it to be used for stacks and different kinds of queues (p. 814).

Example 15.3 illustrates some basic operations on lists. The user gets one shot at guessing a five-digit code. The solution is hardwired in the example as a list of five `Integer` objects. The `secretSolution` list is created at (1), and populated using the `Collections.addAll()` static method (p. 862). The guess specified at the command line is placed in a separate list, called `guess`, at (2).

The number of digits that are correctly guessed is determined at (3). The solution is first duplicated, and each digit in the guess is removed from the duplicated solution. The number of deletions corresponds to the number of correct digits in the `guess` list. A digit at a particular index in the guess list is returned by the `get()` method. The `remove()` method returns `true` if the duplicate list was modified—that is, the digit from the guess was found and removed from the duplicated solution. Of course, one could use the `retainAll()` method, as shown below, but the idea in Example 15.3 is to use positional access on the `guess` list.

Click here to view code image

```
// Find the number of digits that were correctly included.                (3)
List<Integer> duplicate = new ArrayList<>(secretSolution);
duplicate.retainAll(guess);
numOfDigitsIncluded = duplicate.size();
```

Finding the number of digits that are correctly placed is achieved by using two list iterators at (4), which allow digits in the same position in the `guess` and the `secretSolution` lists to be compared.

Example 15.3 *Using Lists*

Click here to view code image

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.ListIterator;

public class TakeAGuess {
  static final int NUM_DIGITS = 5;

  public static void main(String[] args) {

    // Sanity check on the given data.
```

```
    try {
      if (args.length != 1 || args[0].length() != NUM_DIGITS)
        throw new IllegalArgumentException();
      Integer.parseInt(args[0]);
    } catch(IllegalArgumentException nfe) {
      System.err.println("Guess should be " + NUM_DIGITS + " digits.");
      return;
    }
    String guessStr = args[0];
    System.out.println("Guess: " + guessStr);

    /* Initialize the solution list. This program has a fixed solution: 53272 */
    List<Integer> secretSolution = new ArrayList<>();                    // (1)
    Collections.addAll(secretSolution, 5, 3, 2, 7, 2);

    // Convert the guess from string to a list of Integers.                (2)
    List<Integer> guess = new ArrayList<>();
    for (int i = 0; i < guessStr.length(); i++)
      guess.add(Character.getNumericValue(guessStr.charAt(i)));

    // Find the number of digits that were correctly included.             (3)
    List<Integer> duplicate = new ArrayList<>(secretSolution);
    int numOfDigitsIncluded = 0;
    for (int i = 0; i < NUM_DIGITS; i++)
      if (duplicate.remove(guess.get(i))) numOfDigitsIncluded++;

    /* Find the number of digits correctly placed by comparing the two
       lists, element by element, counting each correct placement. */
    // Need two iterators to traverse through the guess and solution lists.  (4)
    ListIterator<Integer> correct = secretSolution.listIterator();
    ListIterator<Integer> attempt = guess.listIterator();
    int numOfDigitsPlaced = 0;
    while (correct.hasNext())
      if (correct.next().equals(attempt.next())) numOfDigitsPlaced++;

    // Print the results.
    System.out.println(numOfDigitsIncluded + " digit(s) correctly included.");
    System.out.println(numOfDigitsPlaced +   " digit(s) correctly placed.");
  }
}
```

Running the program with the following command:

```
>java TakeAGuess 32227
```

gives the following output:

Click here to view code image

```
Guess: 32227
4 digit(s) correctly included.
1 digit(s) correctly placed.
```

15.4 Sets

The Java Collections Framework provides the `Set<E>` interface to model the mathematical *set* abstraction.

The `Set<E>` Interface

Unlike other implementations of the `Collection<E>` interface, implementations of the `Set<E>` interface do not allow duplicate elements. The `Set<E>` interface does not define any new methods, and its `add()` and `addAll()` methods will not store duplicates. If an element is not currently in the set, two consecutive calls to the `add()` method to insert the element will first return `true`, then `false`. A `Set<E>` models a mathematical set (see **Table 15.3**)—that is, it is an unordered collection of distinct objects.

Table 15.3 *Bulk Operations and Set Logic*

Set methods (`a` and `b` are sets)	Corresponding mathematical operations
`a.containsAll(b)`	`b` ⊆ `a` (subset)
`a.addAll(b)`	`a = a` ∪ `b` (union)
`a.removeAll(b)`	`a = a - b` (difference)
`a.retainAll(b)`	`a = a` ∩ `b` (intersection)
`a.clear()`	`a =` ∅ (empty set)

Multisets (also called *bags*) that allow duplicate elements cannot be implemented using the `Set<E>` interface, since this interface requires that elements are unique in the collection. An implementation of the `Set<E>` interface can choose to allow the `null` value, but the concrete class `TreeSet<E>` does *not*.

Creating Unmodifiable Sets

Unmodifiable collections are useful to prevent a collection from accidently being modified, as doing so might cause the program to behave incorrectly.

Creating and using unmodifiable sets is very similar to creating and using unmodifiable lists. Not surprisingly, the discussion here on unmodifiable sets reflects the discussion on unmodi-

fiable lists (§12.2, p. 649). Later we will discuss unmodifiable maps (p. 832) and unmodifiable view collections (p. 856).

The `Set<E>` interface provides generic static methods to create *unmodifiable* sets that have the following characteristics:

- An unmodifiable set cannot be modified *structurally*; for example, elements cannot be added, removed, or replaced in such a set. Any such attempt will result in an `UnsupportedOperationException` to be thrown. However, if the elements themselves are mutable, the elements may appear modified.
- Both duplicates and `null` elements are not allowed, and will result in a `NullPointerException` if an attempt is made to create them with such elements.
- The order of the elements in such a set is unspecified.
- Such a set can be serialized if its elements are serializable (§20.5, p. 1261).

Click here to view code image

```
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5,
                     E e6, E e7, E e8, E e9, E e10)
```

This method is overloaded, accepting any number of elements from 0 to 10. It returns an unmodifiable set containing the number of elements specified. It throws a `NullPointerException` if an element is `null`.

Click here to view code image

```
@SafeVarargs static <E> Set<E> of(E... elements)
```

This variable arity method returns an unmodifiable set containing an arbitrary number of elements. It throws a `NullPointerException` if an element is `null` or if the array is `null`. The annotation suppresses the heap pollution warning in its declaration and unchecked generic array creation warning at the call sites.

Click here to view code image

```
static <E> Set<E> copyOf(Collection<? extends E> collection)
```

This generic method returns an unmodifiable set containing the elements of the specified collection, in its iteration order. The specified collection must not be `null`, and it must not contain any `null` elements—otherwise, a `NullPointer-Exception` is thrown. If the specified collection is subsequently modified, the returned set will not reflect such modifications.

The code below shows that a set created by the `Set.of()` method cannot be modified. The set returned is also not an instance of the class `HashSet`.

Click here to view code image

```
Set<String> set = Set.of("Tom", "Dick", "Harriet");
//  set.add("Harry");                          // UnsupportedOperationException
//  set.remove(2);                             // UnsupportedOperationException
System.out.println(set);                       // [Harriet, Tom, Dick]
System.out.println(set instanceof HashSet);    // false
```

The `Set.of()` method does not allow `null` elements:

Click here to view code image

```
Set<String> coinSet = Set.of("dime", "nickel", null); // NullPointerException
```

For arguments up to 10, an appropriate fixed-arity `Set.of()` method is called. Above 10 arguments, the variable arity `Set.of(E...)` method is called, passing an implicitly created array containing the arguments.

Click here to view code image

```
Set<Integer> intSet1 = Set.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);     // Fixed-arity
Set<Integer> intSet2 = Set.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11); // Varargs
System.out.println(intSet1);        // [9, 8, 7, 6, 5, 4, 3, 2, 1, 10]
System.out.println(intSet2);        // [7, 6, 5, 4, 3, 2, 1, 11, 10, 9, 8]
```

At (1) below, an explicit array is passed as an argument, resulting in the variable arity `Set.of(E...)` method being called, creating a set of `String`. At (2), the method call explicitly specifies the type of its argument as `String[]`. In this case the one-argument `Set.of(E)` method is called, creating a set of size 1 and whose element type is `String[]`.

Click here to view code image

```
String[] strArray = {"Tom", "Dick", "Harriet"};
Set<String> strSet = Set.of(strArray);                   // (1) Set of String
Set<String[]> strArraySet = Set.<String[]>of(strArray);  // (2) Set of String[]
System.out.println(strSet);                  // [Harriet, Dick, Tom]
System.out.println(strArraySet);             // [[Ljava.lang.String;@3b22cdd0]
```

The code below shows how we can make a copy of a collection, in this case, a set. The `copyOf()` method creates a copy of the set passed as an argument at (1). The set created is unmodifiable analogous to the sets created with the `Set.of()` methods. The code also shows that modifying the original set does *not* reflect in the copy of the set.

Click here to view code image

```
Set<String> fab4 = new HashSet<>();
fab4.add("John"); fab4.add("Paul"); fab4.add("George"); fab4.add("Ringo");
System.out.println(fab4);                    // [George, John, Ringo, Paul]
Set<String> fabAlways = Set.copyOf(fab4);    // (1)
fab4.remove("John"); fab4.remove("George");  // Modify original set
System.out.println(fab4);                    // [Ringo, Paul]
System.out.println(fabAlways);               // [John, Paul, Ringo, George]
```

The `HashSet<E>` and `LinkedHashSet<E>` Classes

The `HashSet<E>` class implements the `Set<E>` interface . Since this implementation uses a hash table, it offers near-constant-time performance for most operations. A `HashSet<E>` does not guarantee any ordering of the elements. However, the `LinkedHashSet<E>` subclass of `HashSet<E>` guarantees insertion order. It is also the implementation of choice if frequent iteration is necessary over the set. The sorted counterpart is `TreeSet<E>`, which implements the `SortedSet<E>` and the `NavigableSet<E>` interfaces and has logarithmic time complexity (p. 810).

A `HashSet<E>` relies on the implementation of the `hashCode()` and `equals()` methods of its elements (§14.2, p. 744, and §14.3, p. 753). The `hashCode()` method is used for hashing the elements, and the `equals()` method is needed for comparing elements for equality. In fact, the equality and the hash codes of `HashSet`s are defined in terms of the equality and the hash codes of their elements.

```
HashSet()
```

Constructs a new, empty set.

Click here to view code image

```
HashSet(Collection<? extends E> c)
```

Constructs a new set containing the elements in the specified collection. The new set will not contain any duplicates. This offers a convenient way to remove duplicates from a collection.

```
HashSet(int initialCapacity)
```

Constructs a new, empty set with the specified initial capacity.

Click here to view code image

```
HashSet(int initialCapacity, float loadFactor)
```

Constructs a new, empty set with the specified initial capacity and the specified load factor.

As mentioned earlier, the `LinkedHashSet<E>` implementation is a subclass of the `HashSet<E>` class. It works similarly to a `HashSet<E>` except for one important detail. Unlike a `HashSet<E>`, a `LinkedHashSet<E>` guarantees that the iterator will access the elements in *insertion order*—that is, in the order in which the elements were inserted into the `LinkedHashSet<E>`.

The `LinkedHashSet<E>` class offers constructors analogous to the ones in the `Hash-Set<E>` class. The initial *capacity* (i.e., the number of buckets in the hash table) and its *load factor* (i.e., the ratio of number of elements stored to its current capacity) can be tuned when the set is created. The default values for these parameters will, under most circumstances, provide acceptable performance.

Example 15.4 demonstrates iterating over a `HashSet<E>` and a `LinkedHashSet<E>` created at (1) and (2), respectively. Regardless of the order in which elements are inserted into a `HashSet<E>`, we cannot depend on the order in which the `for(:)` loop will iterate over the elements in the set, as is evident from the program output. A `LinkedHashSet<E>`, on the other hand, is always iterated over in *insertion order*—that is, the first element inserted is the first element retrieved by the `for(:)` loop. The program output confirms this behavior, as the meal that was inserted last into the `LinkedHashSet<E>` is served first. The same behavior will be exhibited if an explicit iterator is used to iterate over the sets.

Example 15.4 *Iterating Over Sets*

Click here to view code image

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
public class IterationHashSetAndLinkedHashSet {
  public static void main(String[] args) {
    Set<String> set1 = new HashSet<>();                              // (1)
    set1.add("breakfast"); set1.add("lunch"); set1.add("dinner");
    System.out.println("Serving meals from a HashSet (order can vary):");
    for (String meal : set1) {
      System.out.println(meal);
    }
    Set<String> set2 = new LinkedHashSet<>();                        // (2)
    set2.add("breakfast"); set2.add("lunch"); set2.add("dinner");
    System.out.println("Serving meals from a LinkedHashSet" +
                       " (always insertion order):");
    for (String meal : set2) {
      System.out.println(meal);
```

```
    }
  }
}
```

Output from the program:

Click here to view code image

```
Serving meals from a HashSet (order can vary):
dinner
breakfast
lunch
Serving meals from a LinkedHashSet (always insertion order):
breakfast
lunch
dinner
```

Example 15.5 demonstrates set operations. It determines the following relationships between two sets of characters:

- Whether they are disjunct—that is, have no elements in common
- Whether they have the same elements—that is, are equivalent
- Whether one is a subset of the other
- Whether one is a superset of the other
- Whether they have a common subset

Given a list of words as program arguments, each argument is turned into a set of characters. This character set is compared with the set of all characters encountered so far in previous arguments.

The set `encountered` created at (1) accumulates characters as each argument is processed. For each argument, an empty set of characters is created at (2). This `characters` set is populated with the characters of the current argument at (3). The program first determines if there is a common subset between the two sets at (4)—that is, whether the current argument has any characters that were in previous arguments:

Click here to view code image

```
// Determine whether a common subset exists:                (4)
Set<Character> commonSubset = new HashSet<>(encountered);
commonSubset.retainAll(characters);
boolean areDisjunct = commonSubset.size()==0;
```

Note that the `retainAll()` operation is destructive. The code at (4) does not affect the `encountered` and the `characters` sets. If the size of the common subset is zero, the sets are disjunct; otherwise, the relationship must be narrowed down. The subset and superset relations are determined at (5), using the `containsAll()` method.

Click here to view code image

```
// Determine superset and subset relations.                (5)
boolean isSubset = encountered.containsAll(characters);
boolean isSuperset = characters.containsAll(encountered);
```

The sets are equivalent if both of the previous relations are `true`. If the relations are both `false`—that is, no subset or superset relationship exists, the sets only have the subset computed at (4) in common. The `encountered` set is updated with the contents of the `characters` set to accumulate all characters encountered so far. The `addAll()` method is used for this purpose at (6):

Click here to view code image

```
encountered.addAll(characters);                          // (6)
```

As we can see from the output, the program prints the contents of the sets in the standard text representation for collections.

Example 15.5 *Using Sets*

Click here to view code image

```
import java.util.HashSet;
import java.util.Set;

public class CharacterSets {
  public static void main(String[] args) {

    // A set for keeping track of all characters previously encountered.
    Set<Character> encountered = new HashSet<>();               // (1)

    // For each program argument in the command line ...
    for (String argument : args) {

      // Convert the current argument to a set of characters.
      Set<Character> characters = new HashSet<>();              // (2)
      int size = argument.length();
      // For each character in the argument...
      for (int j = 0; j < size; j++)
        // add character to the characters set.
        characters.add(argument.charAt(j));                     // (3)

      // Determine whether a common subset exists:                (4)
      Set<Character> commonSubset = new HashSet<>(encountered);
      commonSubset.retainAll(characters);
      boolean areDisjunct = commonSubset.size()==0;

      if (areDisjunct) {
```

```
          System.out.println(characters + " and " + encountered + " are disjunct.");
        } else {
          // Determine superset and subset relations.                (5)
          boolean isSubset = encountered.containsAll(characters);
          boolean isSuperset = characters.containsAll(encountered);
          if (isSubset && isSuperset)
            System.out.println(characters + " is equivalent to " + encountered);
          else if (isSubset)
            System.out.println(characters + " is a subset of " + encountered);
          else if (isSuperset)
            System.out.println(characters + " is a superset of " + encountered);
          else
            System.out.println(characters + " and " + encountered + " have " +
                               commonSubset + " in common.");
        }

        // Update the set of characters encountered so far.
        encountered.addAll(characters);                           // (6)
      }
    }
  }
```

Running the program with the following arguments:

Click here to view code image

```
>java CharacterSets i said i am maids
```

results in the following output:

Click here to view code image

```
[i] and [] are disjunct.
[d, a, s, i] is a superset of [i]
[i] is a subset of [d, a, s, i]
[a, m] and [d, a, s, i] have [a] in common.
[d, a, s, m, i] is equivalent to [d, a, s, m, i]
```

15.5 Sorted Sets and Navigable Sets

Before reading this subsection, it is a good idea to review the `Comparable<E>` interface (§14.4, p. 761) for defining the natural ordering for objects, and the `Comparator<E>` interface (§14.5, p. 769) for defining a particular total ordering for objects.

The `SortedSet<E>` Interface

The `SortedSet<E>` interface extends the `Set<E>` interface to provide the functionality for handling sorted sets. Since the elements are sorted, iterating over the set using either the

`for(:)` loop or an iterator will access the elements according to the ordering used by the set.

```
// First-last elements
E first()
E last()
```

The `first()` method returns the first element currently in this sorted set, and the `last()` method returns the last element currently in this sorted set. The elements are chosen based on the ordering used by the sorted set. Both throw a `NoSuchElementException` if the sorted set is empty.

Click here to view code image

```
// Range-view operations
SortedSet<E> headSet(<E> toElement)
SortedSet<E> tailSet(<E> fromElement)
SortedSet<E> subSet(<E> fromElement, <E> toElement)
```

The `headSet()` method returns a *view* of a portion of this sorted set, whose elements are strictly less than the specified element. Similarly, the `tailSet()` method returns a view of the portion of this sorted set, whose elements are greater than or equal to the specified element. The `subSet()` method returns a view of the portion of this sorted set, whose elements range from `fromElement`, inclusive, to `toElement`, exclusive (also called *half-open interval*). It throws an `IllegalArgumentException` if the `fromElement` is greater than the `toElement`.

Note that the views present the elements sorted in the same order as the underlying sorted set. Also, changes made through views are reflected in the underlying sorted set, and vice versa.

Click here to view code image

```
// Comparator access
Comparator<? super E> comparator()
```

Returns the comparator associated with this sorted set, or `null` if it uses the natural ordering of its elements. This comparator, if defined, is used by default when a sorted set is constructed, and used when copying elements into new sorted sets.

The `NavigableSet<E>` Interface

The `NavigableSet<E>` interface extends the `SortedSet<E>` interface with *navigation methods* to find the closest matches for given search targets. By navigation we mean operations that require searching for elements in the navigable set. In the absence of elements, these op-

erations return `null` rather than throwing a `NoSuchElement-Exception`, as is the case for the methods in the `SortedSet<E>` interface.

The `NavigableSet<E>` interface replaces the `SortedSet<E>` interface and is the preferred choice when a sorted set is required. In addition to the methods of the `Sorted-Set<E>` interface, the `NavigableSet<E>` interface adds the following *new* methods:

```
// First-last elements
E pollFirst()
E pollLast()
```

The `pollFirst()` method removes and returns the first element and the `poll-Last()` method removes and returns the last element currently in this navigable set. The element is determined according to some policy employed by the set—for example, queue policy. Both return `null` if the sorted set is empty.

Click here to view code image

```
// Range-view operations
NavigableSet<E> headSet(E toElement,   boolean inclusive)
NavigableSet<E> tailSet(E fromElement, boolean inclusive)
NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
                       E toElement,   boolean toInclusive)
```

These operations are analogous to the ones in the `SortedSet<E>` interface (**p. 810**), returning different views of the underlying navigable set, depending on the bound elements. However, the bound elements can be *excluded or included* by the operation, depending on the value of the `boolean` argument `inclusive`.

```
// Closest-matches
E ceiling(E e)
E floor(E e)
E higher(E e)
E lower(E e)
```

The method `ceiling()` returns the least element in the navigable set greater than or equal to argument `e`. The method `floor()` returns the greatest element in the navigable set less than or equal to argument `e`. The method `higher()` returns the least element in the navigable set strictly greater than argument `e`. The method `lower()` returns the greatest element in the navigable set strictly less than argument `e`. All methods return `null` if the required element is not found.

Click here to view code image

```
// Reverse order
Iterator<E> descendingIterator()
NavigableSet<E> descendingSet()
```

The first method returns a reverse-order iterator for the navigable set. The second method returns a reverse-order view of the elements in the navigable set.

The `TreeSet<E>` Class

The `TreeSet<E>` class implements the `NavigableSet<E>` interface and thereby the `SortedSet<E>` interface. By default, operations on a sorted set rely on the natural ordering of the elements. However, a total ordering can be specified by passing a customized comparator to the constructor.

The `TreeSet<E>` class maintains non-`null` elements in sort order, and iteration is also in the same sort order.

The `TreeSet<E>` implementation uses balanced trees, which deliver excellent (logarithmic) performance for all operations. However, searching in a `HashSet<E>` can be faster than in a `TreeSet<E>` because hashing algorithms usually offer better performance than the search algorithms for balanced trees. The `TreeSet<E>` class is preferred if elements are to be maintained in sort order and if fast insertion and retrieval of individual elements is desired.

The `TreeSet<E>` class provides four constructors:

```
TreeSet()
```

The default constructor that creates a new, empty sorted set, according to the natural ordering of the elements.

Click here to view code image

```
TreeSet(Comparator<? super E> comparator)
```

A constructor that takes an explicit comparator for specific total ordering of the elements.

Click here to view code image

```
TreeSet(Collection<? extends E> collection)
```

A constructor that creates a sorted set based on a collection, according to the natural ordering of the elements.

```
TreeSet(SortedSet<E> set)
```

A constructor that creates a new set containing the same elements as the specified sorted set, with the same ordering.

Example 15.6 illustrates some selected navigation operations on a `TreeSet<E>`. Keep in mind that the Unicode values of uppercase letters are less than the Unicode values of lowercase letters—that is, the former occur before the latter in the Unicode standard.

The set is created at (1), and populated by calling the `Collections.addAll()` method at (2). The elements are maintained according to the natural ordering for `String`s— that is, the one defined by the `compareTo()` method of the `Comparable<String>` interface implemented by the `String` class. The subset-view operations at (3) show how the bounds can be inclusive or exclusive. Note also how the closest-match methods at (4) behave. A sorted set with the reverse order corresponding to the natural ordering is created at (5). The methods `pollFirst()` and `pollLast()` remove the element that is retrieved—that is, they change the set structurally.

The following code shows how we can create a sorted set with a specific total ordering, by supplying a comparator in the constructor call:

Click here to view code image

```
  NavigableSet<String> strSetB = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
  Collections.addAll(strSetB, "strictly", "dancing", "Java", "Ballroom");
  System.out.println(strSetB);          // [Ballroom, dancing, Java, strictly]
```

Example 15.6 *Using Navigable Sets*

Click here to view code image

```
import java.util.Collections;
import java.util.NavigableSet;
import java.util.TreeSet;
import static java.lang.System.out;

public class SetNavigation {
  public static void main(String[] args) {

    NavigableSet<String> strSetA = new TreeSet<>();                        // (1)
    Collections.addAll(strSetA, "Strictly", "Java", "dancing", "ballroom"); // (2)
    out.println("Before: " + strSetA);      // [Java, Strictly, ballroom, dancing]

    out.println("\nSubset-views:");                  // (3)
    out.println(strSetA.headSet("ballroom", true));  // [Java, Strictly, ballroom]
```

```
    out.println(strSetA.headSet("ballroom", false)); // [Java, Strictly]
    out.println(strSetA.tailSet("Strictly", true));  // [Strictly, ballroom,
                                                     //  dancing]
    out.println(strSetA.tailSet("Strictly", false)); // [ballroom, dancing]
    out.println(strSetA.subSet("A", false, "Z", false )); // [Java, Strictly]
    out.println(strSetA.subSet("a", false, "z", false )); // [ballroom, dancing]

    out.println("\nClosest-matches:");               // (4)
    out.println(strSetA.ceiling("ball"));            // ballroom
    out.println(strSetA.floor("ball"));              // Strictly
    out.println(strSetA.higher("ballroom"));         // dancing
    out.println(strSetA.lower("ballroom"));          // Strictly

    out.println("\nReverse order:");                 // (5)
    out.println(strSetA.descendingSet());   // [dancing, ballroom, Strictly, Java]

    out.println("\nFirst-last elements:");
    out.println(strSetA.pollFirst());                // Java
    out.println(strSetA.pollLast());                 // dancing

    out.println("\nAfter: " + strSetA);             // [Strictly, ballroom]
  }
}
```

Output from the program:

Click here to view code image

```
Before: [Java, Strictly, ballroom, dancing]

Subset-views:
[Java, Strictly, ballroom]
[Java, Strictly]
[Strictly, ballroom, dancing]
[ballroom, dancing]
[Java, Strictly]
[ballroom, dancing]

Closest-matches:
ballroom
Strictly
dancing
Strictly

Reverse order:
[dancing, ballroom, Strictly, Java]

First-last elements:
Java
dancing
```

```
After: [Strictly, ballroom]
```

15.6 Queues

In this section we look at the different types of queues provided by the Java Collections Framework.

The `Queue<E>` Interface

The `Queue<E>` interface extends the `Collection<E>` interface to specify a general contract for queues. A *queue* is a collection that maintains elements in *processing order*. An implementation of the `Queue<E>` interface provides the queue policy for yielding the next element for processing. A *head* position in the queue specifies where the next element for processing can be obtained. A basic queue usually maintains its elements in FIFO (*first in, first out*) ordering, but other orderings are also quite common: LIFO (*last in, first out*) ordering (also called *stacks*) and priority ordering (also called *priority queues*). The order in which elements of a queue can be retrieved for processing is dictated either by the natural ordering of the elements or by a comparator. A queue can be *unbounded* or *capacity-restricted*, depending on its implementation.

The `Queue<E>` interface extends the `Collection<E>` interface with the following methods. A summary of these methods is presented in **Table 15.4**.

Table 15.4 *Summary of Methods in the Queue Interface*

Operation	Throws exception	Returns special value
Insert at the tail	`add(e)` *can throw* `IllegalArgumentException`	`offer(e)` *returns* `true` *or* `false`
Remove from the head	`remove()` *can throw* `NoSuchElementException`	`poll()` *returns head element or* `null`
Examine element at the head	`element()` *can throw* `NoSuchElementException`	`peek()` *returns head element or* `null`

```
// Insert
boolean add(E element)
boolean offer(E element)
```

Both methods insert the specified element in the queue. The return value indicates the success or failure of the operation. The `add()` method inherited from the `Collection<E>` inter-

face throws an `IllegalStateException` if the queue is full, but the `offer()` method does not.

```
// Remove
E remove()
E poll()
```

Both methods retrieve the head element and remove it from the queue. If the queue is empty, the `remove()` method throws a `NoSuchElementException`, but the `poll()` method returns the `null` value.

```
// Examine
E element()
E peek()
```

Both methods retrieve the head element, but do *not* remove it from the queue. If the queue is empty, the `element()` method throws a `NoSuchElementException`, but the `peek()` method returns the `null` value.

The `PriorityQueue<E>` and `LinkedList<E>` Classes

Both the `PriorityQueue<E>` and `LinkedList<E>` classes implement the `Queue<E>` interface. Unless bidirectional iteration is necessary, other queue implementations should be considered, and not the `LinkedList<E>` class. (The `LinkedList<E>` class is also eclipsed by the introduction of the `ArrayDeque<E>` class when it comes to implementing deques, as we will see shortly.)

As the name suggests, the `PriorityQueue<E>` class is the obvious implementation for a queue with priority ordering. The implementation is based on a *priority heap*, a tree-like structure that yields an element at the head of the queue according to the priority ordering, which is defined either by the natural ordering of its elements or by a comparator. In the case of several elements having the same priority, one of them is chosen arbitrarily.

Elements of a `PriorityQueue<E>` are *not* sorted. The queue only guarantees that elements can be *removed* in priority order, and any iteration using an iterator does *not* guarantee to abide by the priority order.

The `PriorityQueue<E>` class provides the following constructors:

Click here to view code image

```
PriorityQueue()
PriorityQueue(int initialCapacity)
```

The default constructor creates a new, empty `PriorityQueue` with default initial capacity and natural ordering. The second constructor creates a new, empty `PriorityQueue` with the specified initial capacity and natural ordering.

Click here to view code image

```
PriorityQueue(Comparator<? super E> comparator)
PriorityQueue(int initialCapacity, Comparator<? super E> comparator)
```

Both constructors create a new, empty `PriorityQueue` where the ordering is defined by the specified `comparator`. The priority queue created by the first and the second constructors has the default initial capacity and the specified initial capacity, respectively.

Click here to view code image

```
PriorityQueue(PriorityQueue<? extends E> pq)
PriorityQueue(SortedSet<? extends E> set)
PriorityQueue(Collection<? extends E> c)
```

The first and the second constructors create a new `PriorityQueue` with the ordering and the elements from the specified priority queue or sorted set, respectively.

The last constructor creates a new `PriorityQueue` containing the elements in the specified collection. It will have natural ordering of its elements, unless the specified collection is either a `SortedSet<E>` or another `PriorityQueue`, in which case, the collection's ordering will be used.

Example 15.7 illustrates using priority queues. The example shows how priority queues maintain objects of the `Task` class. The natural ordering of the objects in this class is based on the *task number* (`Integer`). This natural ordering will result in the priority queue yielding its elements in *ascending* order of the task numbers—that is, tasks with *smaller* task numbers will have *higher* priority.

In **Example 15.7**, the `main()` method in the class `TaskExecutor` creates an array with some tasks at (1). The method essentially creates empty priority queues with different priority orderings, at (2) through (7), and calls the `testPQ()` method at (8) to execute tasks passed in the array using the supplied priority queue.

Click here to view code image

```
private static void testPQ(Task[] taskArray, PriorityQueue<Task> pq) {    // (8)
...
}
```

The `testPQ()` method at (8) loads the queue at (9) from the array of tasks. It calls the `offer()` method to insert a task in the priority queue. The method then calls the `peek()` method at (10) to examine the task at the head of the queue. The tasks are executed by removing them one by one at (11) by calling the `poll()` method. The output shows the order in which the tasks are executed, depending on the priority ordering.

The priority queue `pq1` at (2) has its priority ordering defined by the natural ordering of the tasks.

Click here to view code image

```
PriorityQueue<Task> pq1 = new PriorityQueue<>();                    // (2)
```

Note that the text representation of the queue in the output

Click here to view code image

```
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
```

does *not* reflect the tasks in priority order. It just shows what tasks are in the queue. The text representation of the queue is generated by the print method running an iterator over the queue. The iterator is under no obligation to take the priority order into consideration. The output also shows that the task with the highest priority (i.e., the smallest task number) is at the head of the queue:

Click here to view code image

```
Task at the head: 100@breakfast
```

The call to the `poll()` method in the `while` loop at (11) removes tasks in priority order, as verified by the output:

Click here to view code image

```
Doing tasks: 100@breakfast 200@tea 200@lunch 300@dinner
```

Since two of the tasks have the same priority, the queue selects which one should be chosen first. The queue is empty when the `peek()` method returns `null`.

The priority queue `pq2` at (3) has its priority ordering defined by the reverse natural ordering returned by the static method `reverseOrder()` of the `Comparator<E>` interface:

Click here to view code image

```
PriorityQueue<Task> pq2 = new PriorityQueue<>(Comparator.reverseOrder());
```

Both priority queues `pq3` and `pq4` use reversed ordering based on the *task name*. This ordering is defined for `pq3` and `pq4` by a lambda expression at (4) and by methods of the `Comparator<E>` interface at (7), respectively. The latter implementation is obviously to be preferred:

Click here to view code image

```
PriorityQueue<Task> pq4 = new PriorityQueue<>(                          // (5)
    Comparator.comparing(Task::getTaskName).reversed()
);
```

The static method `comparing()` extracts a comparator that uses the `getName()` method of the `Task` class, and the default method `reversed()` reverses this comparator.

The priority queues `pq5` and `pq6` use total ordering based on multiple fields of the `Task` class: on the task number, followed by the task name. This ordering is defined for `pq5` and `pq6` by a lambda expression at (6) and by methods of the `Comparator<E>` interface at (7), respectively. The latter implementation first extracts a comparator based on the task number, and chains it with one based on the task name:

Click here to view code image

```
PriorityQueue<Task> pq6 = new PriorityQueue<>(                          // (7)
      Comparator.comparing(Task::getTaskNumber)
                .thenComparing(Task::getTaskName)
);
```

We leave it to the reader to verify that the output conforms to the priority ordering of priority queues at (3) through (7).

Example 15.7 *Using Priority Queues*

Click here to view code image

```
/** Represents a task. */
public class Task implements Comparable<Task> {
  private Integer taskNumber;
  private String  taskName;

  public Task(Integer tp, String tn) {
    taskNumber = tp;
    taskName   = tn;
  }

  @Override
  public boolean equals(Object obj) {// Equality based on the task number.
    return (this == obj)
        || (obj instanceof Task other
```

```
            && this.taskNumber.equals(other.taskNumber));
  }
  @Override
  public int compareTo(Task task2) { // Natural ordering based on the task number.
    return this.taskNumber.compareTo(task2.taskNumber);
  }
  @Override
  public int hashCode() {            // Hash code based on the task number.
    return this.taskNumber.hashCode();
  }
  @Override
  public String toString() { return taskNumber + "@" + taskName; }

  public String  getTaskName()   { return taskName; }
  public Integer getTaskNumber() { return taskNumber;}
}
```

Click here to view code image

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.PriorityQueue;
import static java.lang.System.out;
/** Executes tasks. */
public class TaskExecutor {

  public static void main(String[] args) {
    // Array with some tasks.                                          (1)
    Task[] taskArray = {
        new Task(200, "lunch"), new Task(200, "tea"),
        new Task(300, "dinner"), new Task(100, "breakfast"),
    };
    out.println("Array of tasks: " + Arrays.toString(taskArray));

    out.println("Priority queue using natural ordering (task number).");
    PriorityQueue<Task> pq1 = new PriorityQueue<>();                    // (2)
    testPQ(taskArray, pq1);

    out.println("Priority queue using reverse natural ordering.");      // (3)
    PriorityQueue<Task> pq2 = new PriorityQueue<>(Comparator.reverseOrder());
    testPQ(taskArray, pq2);

    out.println("Priority queue using reversed ordering"
        + " on task name (lambda expression).");
    PriorityQueue<Task> pq3 = new PriorityQueue<>(                      // (4)
        (task1, task2) -> {
          String taskName1 = task1.getTaskName();
          String taskName2 = task2.getTaskName();
          return -taskName1.compareTo(taskName2);
         }
      );
```

```
    testPQ(taskArray, pq3);

    out.println("Priority queue using reversed ordering"
        + " on task name (extracted comparator).");
    PriorityQueue<Task> pq4 = new PriorityQueue<>(                          // (5)
        Comparator.comparing(Task::getTaskName).reversed()
    );
    testPQ(taskArray, pq4);

    out.println("Priority queue using total ordering based on task number,"
        + "\nfollowed by task name (lambda expression).");
    PriorityQueue<Task> pq5 = new PriorityQueue<>(                          // (6)
      (task1, task2) -> {
        Integer taskNumber1 = task1.getTaskNumber();
        Integer taskNumber2 = task2.getTaskNumber();
        if (!taskNumber1.equals(taskNumber2))
          return taskNumber1.compareTo(taskNumber2);
        String taskName1 = task1.getTaskName();
        String taskName2 = task2.getTaskName();
        if (!taskName1.equals(taskName2))
          return taskName1.compareTo(taskName2);
        return 0;
       }
    );
    testPQ(taskArray, pq5);

    out.println("Priority queue using total ordering based on task number,"
        + "\nfollowed by task name (extracted comparators).");
    PriorityQueue<Task> pq6 = new PriorityQueue<>(                          // (7)
        Comparator.comparing(Task::getTaskNumber)
                  .thenComparing(Task::getTaskName)
    );
    testPQ(taskArray, pq6);
  }

  // Runs tasks.
  private static void testPQ(Task[] taskArray, PriorityQueue<Task> pq) {    // (8)
    // Load the tasks:                                                        (9)
    for (Task task : taskArray)
      pq.offer(task);
    out.println("Queue before executing tasks: " + pq);

    // Peek at the head:                                                     (10)
    out.println("Task at the head: " + pq.peek());

    // Do the tasks:                                                         (11)
    out.print("Doing tasks: ");
    while (!pq.isEmpty()) {
      Task task = pq.poll();
      out.print(task + " ");
    }
    out.println();
```

```
    out.println();
  }
}
```

Output from the program:

Click here to view code image

```
Array of tasks: [200@lunch, 200@tea, 300@dinner, 100@breakfast]
Priority queue using natural ordering (task number).
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
Task at the head: 100@breakfast
Doing tasks: 100@breakfast 200@tea 200@lunch 300@dinner

Priority queue using reverse natural ordering.
Queue before executing tasks: [300@dinner, 200@tea, 200@lunch, 100@breakfast]
Task at the head: 300@dinner
Doing tasks: 300@dinner 200@tea 200@lunch 100@breakfast

Priority queue using reversed ordering on task name (lambda expression).
Queue before executing tasks: [200@tea, 200@lunch, 300@dinner, 100@breakfast]
Task at the head: 200@tea
Doing tasks: 200@tea 200@lunch 300@dinner 100@breakfast

Priority queue using reversed ordering on task name (extracted comparator).
Queue before executing tasks: [200@tea, 200@lunch, 300@dinner, 100@breakfast]
Task at the head: 200@tea
Doing tasks: 200@tea 200@lunch 300@dinner 100@breakfast

Priority queue using total ordering based on task number,
followed by task name (lambda expression).
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
Task at the head: 100@breakfast
Doing tasks: 100@breakfast 200@lunch 200@tea 300@dinner

Priority queue using total ordering based on task number,
followed by task name (extracted comparators).
Queue before executing tasks: [100@breakfast, 200@lunch, 300@dinner, 200@tea]
Task at the head: 100@breakfast
Doing tasks: 100@breakfast 200@lunch 200@tea 300@dinner
```

15.7 Deques

In this section we look at *deques*—that is, linear collections that allow processing of elements from both ends.

The `Deque<E>` Interface

The `Deque<E>` interface extends the `Queue<E>` interface to allow *double-ended queues*. Such a queue is called a *deque*. It allows operations not just at its *head* as a queue, but also at its *tail*. It is a linear unbounded structure in which elements can be inserted at or removed from *either* end. Various synonyms are used in the literature for the head and tail of a deque: front and back, first and last, start and end.

A deque can be used as a *FIFO queue*, where elements added at the tail are presented at the head for inspection or removal in the same order, thus implementing FIFO ordering. A deque can also be used as a *stack*, where elements are added to and removed from the *same* end, thus implementing LIFO ordering.

The `Deque<E>` interface defines symmetrical operations at its head and tail. Which end is in question is made evident by the method name. A `XXXFirst()` method and a `XXXLast()` method process an element at the *head* and at the *tail*, respectively. Below, equivalent methods from the `Queue<E>` interface are also identified. The `push()` and `pop()` methods are convenient for implementing stacks.

Click here to view code image

```
// Insert
boolean offerFirst(E element)
boolean offerLast(E element)                Queue equivalent: offer()
void addFirst(E element)
void addLast(E element)                     Queue equivalent: add()
void push(E element)                        Synonym: addFirst()
```

Insert the specified element in the deque. They all throw a `NullPointerException` if the specified element is `null`. The `addFirst()` and `addLast()` methods throw an `IllegalStateException` if the element cannot be added, but the `offerFirst()` and `offerLast()` methods do not.

Click here to view code image

```
// Remove
E removeFirst()                             Queue equivalent: remove()
E removeLast()
E pollFirst()                               Queue equivalent: poll()
E pollLast()
E pop()                                     Synonym: removeFirst()
boolean removeFirstOccurence(Object obj)
boolean removeLastOccurence(Object obj)
```

Remove an element from the deque. The `removeFirst()` and `removeLast()` methods throw a `NoSuchElementException` if the deque is empty. The `pollFirst()` and `pollLast()` meth-

ods return `null` if the deque is empty.

Click here to view code image

```
// Examine
E getFirst()                                    Queue equivalent: element()
E getLast()
E peekFirst()                                   Queue equivalent: peek()
E peekLast()
```

Retrieve an element from the deque, but do not remove it from the deque. The `getFirst()` and `getLast()` methods throw a `NoSuchElementException` if the deque is empty. The `peekFirst()` and `peekLast()` methods return `null` if the deque is empty.

Click here to view code image

```
// Misc.
Iterator<E> descendingIterator()
```

Returns an iterator to iterate over the deque in reverse order—that is, from the tail to the head.

Table 15.5 summarizes the methods provided by the `Deque<E>` interface, showing which operations can be performed at the head and which ones at the tail of the deque. Each row indicates the runtime behavior of the methods in that row, whether they throw an exception or not. Any method whose name starts with either `"offer"`, `"poll"`, or `"peek"` does not throw an exception. Counterpart methods inherited from the `Queue<E>` interface are marked by an asterisk (`*`) in the table.

Table 15.5 *Summary of Deque Methods*

Insert at the head	Insert at the tail	Runtime behavior on failure
`offerFirst()`	`offerLast()`, `offer()*`	*Returns* `false` *if full*
`addFirst()`	`addLast()`, `add()*`	*Throws* `IllegalStateException`
Remove from the head	**Remove from the tail**	**Runtime behavior on failure**
`pollFirst()`, `poll()*`	`pollLast()`	*Returns* `null` *if empty*

Insert at the head	Insert at the tail	Runtime behavior on failure
`removeFirst()`, `remove()*`	`removeLast()`	*Throws* `NoSuchElementException`
`peekFirst()`, `peek()*`	`peekLast()`	*Returns* `null` *if empty*
`getFirst()`, `element()*`	`getLast()`	*Throws* `NoSuchElementException`

The `ArrayDeque<E>` and `LinkedList<E>` Classes

The `ArrayDeque<E>` and `LinkedList<E>` classes implement the `Deque<E>` interface. The `ArrayDeque<E>` class provides better performance than the `LinkedList<E>` class for implementing FIFO queues, and is also a better choice than the `java.util.Stack` class for implementing stacks.

An `ArrayDeque<E>` is also an `Iterable<E>`, and iteration is always from the head to the tail. The class provides the `descendingIterator()` method for iterating in reverse order. Since deques are not lists, positional access is not possible, nor can they be sorted. They are also not thread-safe, and `null` values are not allowed as elements.

The `ArrayDeque<E>` class provides the following constructors, analogous to the ones in the `ArrayList<E>` class:

```
ArrayDeque()
ArrayDeque(int numOfElements)
```

The first constructor creates a new, empty `ArrayDeque` with an initial capacity to hold 16 elements.

The second constructor creates a new, empty `ArrayDeque` with initial capacity required to hold the specified number of elements.

Click here to view code image

```
ArrayDeque(Collection<? extends E> c)
```

Creates a new `ArrayDeque` containing the elements in the specified collection. The ordering in the `ArrayDeque` is determined by the iteration order of the iterator for the collection passed as an argument.

The `LinkedList<E>` class provides constructors that are analogous to the first and the last constructors.

Example 15.8 illustrates the methods for inserting, examining, and removing elements from a deque. The program output shows how each method affects the deque when inserting, examining, and removing elements from either the head or the tail of a deque.

Example 15.8 *Demonstrating Deque Operations*

Click here to view code image

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeOperations {
  public static void main(String[] args) {
    Deque<String> deque = new ArrayDeque<String>();
    System.out.println("After creating the deque: " + deque);

    // Insert elements:
    deque.offerFirst("A (H)");              // Insert at the head
    System.out.println("After inserting at the head: " + deque);
    deque.offerLast("B (T)");               // Insert at the tail
    System.out.println("After inserting at the tail: " + deque);
    deque.push("C (H)");                    // Insert at the head
    System.out.println("After inserting at the head: " + deque);
    deque.addFirst("D (H)");                // Insert at the head
    System.out.println("After inserting at the head: " + deque);
    deque.addLast("E (T)");                 // Insert at the tail
    System.out.println("After inserting at the tail: " + deque);

    // Examine element:
    System.out.println("Examine at the head: " + deque.getFirst());
    System.out.println("Examine at the tail: " + deque.getLast());
    System.out.println("Examine at the head: " + deque.peekFirst());
    System.out.println("Examine at the tail: " + deque.peekLast());

    // Remove elements:
    deque.removeFirst();                    // Remove from the head
    System.out.println("After removing from the head: " + deque);
    deque.removeLast();                     // Remove from the tail
    System.out.println("After removing from the tail: " + deque);
    deque.pollFirst();                      // Remove from the head
    System.out.println("After removing from the head: " + deque);
    deque.pollLast();                       // Remove from the tail
    System.out.println("After removing from the tail: " + deque);
    deque.pop();                            // Remove from the head
    System.out.println("After removing from the head: " + deque);
```

```
  }
}
```

Output from the program:

Click here to view code image

```
After creating the deque: []
After inserting at the head: [A (H)]
After inserting at the tail: [A (H), B (T)]
After inserting at the head: [C (H), A (H), B (T)]
After inserting at the head: [D (H), C (H), A (H), B (T)]
After inserting at the tail: [D (H), C (H), A (H), B (T), E (T)]
Examine at the head: D (H)
Examine at the tail: E (T)
Examine at the head: D (H)
Examine at the tail: E (T)
After removing from the head: [C (H), A (H), B (T), E (T)]
After removing from the tail: [C (H), A (H), B (T)]
After removing from the head: [A (H), B (T)]
After removing from the tail: [A (H)]
After removing from the head: []
```

Example 15.9 illustrates using an `ArrayDeque` both as a LIFO stack and as a FIFO queue. Elements from an array are pushed onto the stack at (3), and then popped from the stack at (5). The call to the `isEmpty()` method in the `while` loop at (4) determines whether the stack is empty. The output shows that the elements were popped in the reverse order to the order in which they were inserted—that is, LIFO ordering.

Similarly, elements from an array are inserted at the tail of a FIFO queue at (8), and then removed from the head of the FIFO queue at (10). The call to the `isEmpty()` method in the `while` loop at (4) determines whether the FIFO queue is empty. The output shows that the elements were removed in the same order they were inserted—that is, FIFO ordering.

Note that in **Example 15.9** the stack grows at the head of the deque, but the FIFO queue grows at the tail of the deque.

Example 15.9 *Using Deques as a LIFO Stack and as a FIFO Queue*

Click here to view code image

```
import java.util.ArrayDeque;
import java.util.Arrays;

/** Executes tasks. */
public class TaskExecutor2 {

  public static void main(String[] args) {
```

```
    String[] elementArray = {"sway", "and", "twist", "stacks", "tall"};     // (1)
    System.out.println("Array of elements: " + Arrays.toString(elementArray));

    // Using ArrayDeque as a stack:                                          (2)
    ArrayDeque<String> stack = new ArrayDeque<>();
    for (String string : elementArray)
      stack.push(string);                             // (3) Push elements.
    System.out.println("Stack before: TOP->" + stack + "<-BOTTOM");
    System.out.print("Popping stack: ");
    while (!stack.isEmpty()) {                        // (4)
      System.out.print(stack.pop() + " ");            // (5) Pop elements.
    }
    System.out.println("\n");

    // Using ArrayDeque as a FIFO queue:                  (6)
    elementArray = new String[] {"Waiting", "in", "queues", "is", "boring"};// (7)
    System.out.println("Array of elements: " + Arrays.toString(elementArray));
    ArrayDeque<String> fifoQueue = new ArrayDeque<>();
    for (String string : elementArray)
      fifoQueue.offerLast(string);                    // (8) Insert at tail.
    System.out.println("Queue before: HEAD->" + fifoQueue  + "<-TAIL");
    System.out.print("Polling queue: ");
    while (!fifoQueue.isEmpty()) {                    // (9)
      String string = fifoQueue.pollFirst();          // (10) Remove from head.
      System.out.print(string.toUpperCase() + " ");
    }
    System.out.println();
  }
}
```

Output from the program:

Click here to view code image

```
Array of elements: [sway, and, twist, stacks, tall]
Stack before: TOP->[tall, stacks, twist, and, sway]<-BOTTOM
Popping stack: tall stacks twist and sway

Array of elements: [Waiting, in, queues, is, boring]
Queue before: HEAD->[Waiting, in, queues, is, boring]<-TAIL
Polling queue: WAITING IN QUEUES IS BORING
```

Review Questions

15.1 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class RQ400A100 {
  public static void main(String[] args) {
    int sum = 0;
    for (int i : makeCollection()) {
      sum += i;
    }
    System.out.println(sum);
  }

  static Collection<Integer> makeCollection() {
    System.out.println("A collection coming up.");
    Collection<Integer> collection = new ArrayList<>();
    collection.add(10); collection.add(20); collection.add(30);
    return collection;
  }
}
```

Select the one correct answer.

a. The program will print

```
A collection coming up.
60
```

b. The program will print

```
A collection coming up.
A collection coming up.
A collection coming up. p. 60
```

c. The program will fail to compile.

d. None of the above

15.2 Given the following code:

Click here to view code image

```
import java.util.*;
class Fruity {
  private String fName;
  Fruity(String fName) { this.fName = fName; }

  public void setName(String newName) { this.fName = newName; }
  public String toString() { return fName; }
  public boolean equals(Object other) {
    if (this == other) return true;
```

```
    if (!(other instanceof Fruity)) return false;
    return fName.equalsIgnoreCase(((Fruity)other).fName);
  }
}

public class RQ400A50 {
  public static void main(String[] args) {
    Fruity apple = new Fruity("Apple");
    Fruity orange = new Fruity("Orange");
    List<Fruity> list = new ArrayList<>();
    list.add(apple); list.add(orange); list.add(apple);
    System.out.println("Before: " + list);
    // (1) INSERT CODE HERE
    System.out.println("After:  " + list);
  }
}
```

Which code, when inserted at (1), will result in the following output from the program:

Click here to view code image

```
Before: [Apple, Orange, Apple]
After:  [Orange]
```

Select the two correct answers.

a.

```
for (Fruity f : list) {
  if (f.equals(apple))
    list.remove(f);
}
```

b.

```
int i = 0;
for (Fruity f : list) {
  if (f.equals(apple))

    list.remove(i);
  i++;
}
```

c.

Click here to view code image

```
for (int j = 0; j < list.size(); j++) {
  Fruity f = list.get(j);
  if (f.equals(apple))
    list.remove(j);
}
```

d.

Click here to view code image

```
Iterator<Fruity> itr = list.iterator();
while (itr.hasNext()) {
  Fruity f = itr.next();

  if (f.equals(apple))
    itr.remove();
}
```

15.3 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
class AnotherListIterator<T> implements Iterable<T>{

  private List<T> lst;
  public AnotherListIterator(List<T> lst) { this.lst = lst; }

  public Iterator<T> iterator() {
    return new Iterator<T>() {
      private int next = lst.size() - 1;

      public boolean hasNext() { return (next >= 0); }
      public T next() {
        T element = lst.get(next);
        next--;
        return element;
      }
    };
  }

  public static void main(String[] args) {
    List<String> lst = List.of("Hi", "Howdy", "Hello");
    AnotherListIterator<String> rlt = new AnotherListIterator<>(lst);
    for (String str : rlt) {
      System.out.print("|" + str + "|");
    }
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile, but it will throw an exception when run.

c. The program will compile and print `|Hi||Howdy||Hello|` at runtime.

d. The program will compile and print `|Hello||Howdy||Hi|` at runtime.

e. The program will compile and print the strings in an unpredictable order at runtime.

15.4 Which of the following statements are true about collections? Select the two correct answers.

a. Methods calling optional operations in a collection must either catch an `UnsupportedOperationException`, or declare it in their `throws` clause.

b. A `List<E>` can have duplicate elements.

c. An `ArrayList<E>` can only accommodate a fixed number of elements.

d. A `Set<E>` allows at most one `null` element.

15.5 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class Sets {
  public static void main(String[] args) {
    HashSet<Integer> set1 = new HashSet<>();
    addRange(set1, 1);
    ArrayList<Integer> list1 = new ArrayList<>();
    addRange(list1, 2);
    TreeSet<Integer> set2 = new TreeSet<>();
    addRange(set2, 3);
    ArrayDeque<Integer> deque = new ArrayDeque<>();
    addRange(deque, 5);
    set1.removeAll(list1);
    list1.addAll(set2);
    deque.addAll(list1);
    set1.removeAll(deque);
    System.out.println(set1);
  }
  static void addRange(Collection<Integer> col, int step) {
    for (int i = step * 2; i <= 25; i += step) {
      col.add(i);
    }
```

```
  }
}
```

Select the one correct answer.

a. The program will fail to compile, since operations are performed on incompatible collection implementations.

b. The program will fail to compile, since no `Comparator` is supplied to the `TreeSet` constructor for sorting the elements.

c. The program will compile. When run, it will throw an `UnsupportedOperation-Exception`.

d. The program will compile. When run, it will print all primes below 25.

15.6 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class Iterate {
  public static void main(String[] args) {

    List<String> l = new ArrayList<>();
    l.add("A"); l.add("B"); l.add("C"); l.add("D"); l.add("E");
    ListIterator<String> i = l.listIterator();
    i.next(); i.next(); i.next(); i.next();
    i.remove();
    i.previous(); i.previous();
    i.remove();
    System.out.println(l);
  }
}
```

Select the one correct answer.

a. It will print `[A, B, C, D, E]`.

b. It will print `[A, C, E]`.

c. It will print `[B, D, E]`.

d. It will print `[A, B, D]`.

e. It will print `[B, C, E]`.

f. It will throw a `NoSuchElementException`.

15.7 Which of the following statements, when inserted independently at (1), will guarantee that the following program will print `[1, 9]`?

Click here to view code image

```
import java.util.*;
public class RightOne {
  public static void main(String[] args) {
    // (1) INSERT DECLARATION HERE
    collection.add(1); collection.add(9); collection.add(1);
    System.out.println(collection);
  }
}
```

Select the four correct answers.

a.

Click here to view code image

```
Collection<Integer> collection = new HashSet<>();
```

b.

Click here to view code image

```
Set<Integer> collection = new HashSet<>();
```

c.

Click here to view code image

```
HashSet<Integer> collection = new LinkedHashSet<>();
```

d.

Click here to view code image

```
Set<Integer> collection = new LinkedHashSet<>();
```

e.

Click here to view code image

```
Collection<Integer> collection = new TreeSet<>();
```

f.

Click here to view code image

```
NavigableSet<Integer> collection = new TreeSet<>();
```

15.8 Which of the following statements, when inserted independently at (1), will result in program output that does not include the word `"shell"`?

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;
public class RQ400A400 {
  public static void main(String[] args) {
    NavigableSet<String> strSetA = new TreeSet<>();
    Collections.addAll(strSetA, "sea", "shell", "soap", "swan");
    // (1) INSERT STATEMENT HERE
  }
}
```

Select the two correct answers.

a.

Click here to view code image

```
out.println(strSetA.headSet("soap", true));
```

b.

Click here to view code image

```
out.println(strSetA.headSet("soap", false));
```

c.

Click here to view code image

```
out.println(strSetA.tailSet("soap", true));
```

d.

Click here to view code image

```
out.println(strSetA.tailSet("soap", false));
```

e.

Click here to view code image

```
out.println(strSetA.subSet("sea", false, "soap", true));
```

f.

Click here to view code image

```
out.println(strSetA.subSet("sea", true, "soap", false));
```

15.8 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 most, implementing what is called a *single-valued map*. Thus there is a *many-to-one* relationship between keys and values. For example, in a student-grade map, many students (keys) can be awarded the same grade (value), but each student has only one grade. Replacing 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<K, V>` interface does not extend the `Collection<E>` 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 (p. 844).

The `Map<K, V>` 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<K, V>` interface (see Table 15.2, p. 788, and Figure 15.3, p. 787).

The `Map<K,V>` interface and its subinterfaces `SortedMap<K,V>` and `NavigableMap<K,V>` provide a versatile set of methods to implement 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. Many of the methods will be used in examples presented in this chapter.

Click here to view code image

```
V 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.

Click here to view code image

```
V 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`.

Click here to view code image

```
V 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`.

Click here to view code image

```
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`.

Click here to view code image

```
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`.

Creating Unmodifiable Maps

Creating unmodifiable maps is analogous to creating unmodifiable lists (§12.2, p. 649) or unmodifiable sets (p. 804). The `Map<K, V>` interface provides factory methods to create *unmodifiable maps* that have the following characteristics:

- Keys and values in such maps cannot be added, removed, or updated. Any such attempt will result in an `UnsupportedOperationException` to be thrown. However, if the keys and values themselves are mutable, the map may appear to be modified.
- The `null` value cannot be used for keys and values, and will result in a `NullPointerException` if an attempt is made to create such a map with `null` keys or `null` values.
- Duplicate keys are rejected when creating such a map, resulting in an `IllegalArgumentException`.
- The iteration order of mappings in such maps is unspecified.
- Such maps are serializable if their keys and values are serializable (§20.5, p. 1261).

Click here to view code image

```
static <K,?V> Map<K,?V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4,
                           K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8,
                           K k9, V v9, K k10, V v10)
```

This method is overloaded, accepting any number of entries (`k`, `v`) from 0 to 10. It returns an unmodifiable map containing the number of mappings specified. It throws a

`NullPointerException` if any key or value is `null`. It throws an `IllegalArgumentException` if there are any duplicate keys.

Click here to view code image

```
@SafeVarargs static <K,V> Map<K,V> ofEntries(
             Map.Entry<? extends K,? extends V>... entries)
```

This variable arity method returns an unmodifiable map containing an arbitrary number of entries. It throws a `NullPointerException` if a key or a value is `null`, or if the variable arity parameter `entries` is `null`. The annotation suppresses the heap pollution warning in its declaration and unchecked generic array creation warning at the call sites. See the method `entry()` below to create individual entries.

Click here to view code image

```
static <K,V> Map.Entry<K,V> entry(K k, V v)
```

This generic method returns an unmodifiable `Map.Entry` object containing the specified key and value. Attempts to create an entry using a `null` key or a `null` value result in a `NullPointerException`.

Each entry—that is, <*key*, *value*> pair—is represented by an object implementing the nested `Map.Entry<K, V>` interface. An entry can be manipulated by methods defined in this interface, which are self-explanatory:

Click here to view code image

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

    V getValue();
    V setValue(V value);   // Only if the entry is modifiable.
}

static <K,V> Map<K,V> copyOf(Map<? extends K, ? extends V> map)
```

This generic method returns an unmodifiable map containing copies of the entries in the specified map. The specified map must not be `null`, and it must not contain any `null` keys or values—otherwise, a `NullPointerException` is thrown. If the specified map is subsequently modified, the returned `Map<K,V>` will not reflect such modifications.

The code below shows that a map created by the `Map.of()` method cannot be modified. We cannot change or remove any entry in the map. Note that the `Map.of()` method allows up to

10 entries, and there is no variable arity `Map.of()` method. The map returned is also not an instance of the `HashMap` class.

Click here to view code image

```
Map<Integer, String> jCourses = Map.of(
                    200, "Basic Java",    300, "Intermediate Java",
                    400, "Advanced Java", 500, "Kickass Java");
//  jCourses.put(200, "Java Jive");              // UnsupportedOperationException
//  jCourses.remove(500);                        // UnsupportedOperationException
System.out.println(jCourses instanceof HashMap); // false
System.out.println(jCourses);
// {200=Basic Java, 400=Advanced Java, 300=Intermediate Java, 500=Kickass Java}
```

The `Map.of()` method does not allow duplicate keys, and keys or values cannot be `null`:

Click here to view code image

```
Map<Integer, String> coursesMap1
        = Map.of(101, "Java 1.1", 101, "Java 17");  // IllegalArgumentException
Map<Integer, String> coursesMap2
        = Map.of(101, "Java 1.1", 101, null);       // NullPointerException
```

The following code creates unmodifiable map entries, where both the key and the value are immutable:

Click here to view code image

```
//  Map.Entry<Integer, String> e0 = Map.entry(100, null); // NullPointerException
Map.Entry<Integer, String> e1 = Map.entry(200, "Basic Java");
Map.Entry<Integer, String> e2 = Map.entry(300, "Intermediate Java");
Map.Entry<Integer, String> e3 = Map.entry(400, "Advanced Java");
Map.Entry<Integer, String> e4 = Map.entry(500, "Kickass Java");
```

The variable arity `Map.ofEntries()` method can be used to create an unmodifiable map from an arbitrary number of unmodifiable entries.

Click here to view code image

```
Map<Integer, String> unmodCourseMap = Map.ofEntries(e1, e2, e3, e4);  // Varargs
// {300=Intermediate Java, 500=Kickass Java, 200=Basic Java, 400=Advanced Java}
//unmodCourseMap.replace(200, "Java Jive");    // UnsupportedOperationException
//unmodCourseMap.remove(500);                  // UnsupportedOperationException
```

The following code creates mutable course names that we will use in the next map.

Click here to view code image

```
StringBuilder mc1 = new StringBuilder("Basic Java");
StringBuilder mc2 = new StringBuilder("Intermediate Java");
StringBuilder mc3 = new StringBuilder("Advanced Java");
StringBuilder mc4 = new StringBuilder("Kickass Java");
```

The following code creates unmodifiable map entries, where the keys are immutable but the values are mutable:

Click here to view code image

```
Map.Entry<Integer, StringBuilder> me1 = Map.entry(200, mc1);
Map.Entry<Integer, StringBuilder> me2 = Map.entry(300, mc2);
Map.Entry<Integer, StringBuilder> me3 = Map.entry(400, mc3);
Map.Entry<Integer, StringBuilder> me4 = Map.entry(500, mc4);
```

We can use the variable arity `Map.ofEntries()` method to create an unmodifiable map, where trying to replace or remove a course results in an `UnsupportedOperation-Exception`:

Click here to view code image

```
Map<Integer, StringBuilder> unmodMapWithMutableCourses
             = Map.ofEntries(me1, me2, me3, me4);         // Varargs
System.out.println(unmodMapWithMutableCourses);
// {200=Basic Java, 500=Kickass Java, 300=Intermediate Java, 400=Advanced Java}

// unmodMapWithMutableCourses.replace(200, mc4); // UnsupportedOperationException
// unmodMapWithMutableCourses.remove(400);       // UnsupportedOperationException
```

However, the mutable values in the map can be modified:

Click here to view code image

```
StringBuilder mutableCourse = unmodMapWithMutableCourses.get(500);
mutableCourse.replace(0, 7, "Smartass");
```

Click here to view code image

```
System.out.println(unmodMapWithMutableCourses);
// {400=Advanced Java, 500=Smartass Java, 200=Basic Java, 300=Intermediate Java}
```

The code below shows how we can make a copy of a map. The `Map.copyOf()` method creates a copy of the map passed as an argument at (1). The map created is unmodifiable analogous to the maps created with the `Map.of()` or `Map.ofEntries()` methods. The code also shows that modifying the original map does *not* reflect in the copy of the map.

Click here to view code image

```
// Original map:
Map<Integer, StringBuilder> courseMap = new HashMap<>();
courseMap.put(200, mc1); courseMap.put(300, mc2);
courseMap.put(400, mc3); courseMap.put(500, mc4);

// Unmodifiable copy of the map:
Map<Integer, StringBuilder> copyCourseMap = Map.copyOf(courseMap); // (1)

// Modify original map:
courseMap.remove(200);
courseMap.remove(400);
System.out.println("Original: " + courseMap);
System.out.println("Copy: " + copyCourseMap);
```

The code above prints the contents of the maps, showing that the copy of the map was not modified:

Click here to view code image

```
Original: {500=Smartass Java, 300=Intermediate Java}
Copy: {300=Intermediate Java, 500=Smartass Java, 200=Basic Java,
       400=Advanced Java}
```

Advanced Key-Based Operations

The following methods for a map take a *function* (implemented by a lambda expression or a method reference) as a parameter to implement various scenarios that manipulate the value associated with a key.

Click here to view code image

```
default V merge(K key, V value,
                BiFunction<? super V,? super V,? extends V> remappingFunc)
```

If the specified `key` has no entry or is associated with a `null` value, the method associates the `key` with the specified non-`null value`. Otherwise, it associates the `key` with the result of applying the remapping two-arity function to the specified `value` and the currently associated value, or removes the entry for the `key` if the result is `null`.

Note that if the specified `value` is `null`, a `NullPointerException` is thrown at runtime.

Returns the new value associated with the `key`, or `null` if entry for the `key` is removed.

Click here to view code image

```
default V compute(K key,
                  BiFunction<? super K,? super V,? extends V> remappingFunc)
```

The remapping two-arity function is applied to the specified `key` and its currently associated value. If the result is non-`null`, it associates the key with the result. Otherwise, it removes any entry for the `key`.

Returns the computed value associated with the `key`, or `null` if the `key` is removed or is not associated with a value.

Click here to view code image

```
default V computeIfAbsent(K key,
                          Function<? super K,? extends V> mappingFunc)
```

If the value associated with the specified key is `null` or the key has no entry, this method applies the given function on the `key`, and the result is only associated with the `key` if the result is non-`null`.

Returns the current (existing or computed) value associated with the specified `key`, or `null` if the computed value is `null`.

Click here to view code image

```
default V computeIfPresent(K key,
                           BiFunction<? super K,? super V,? extends V> remappingFunc)
```

If the specified key is already associated with a non-`null` value, the method applies the specified remapping two-arity function to the `key` and its currently associated value. The result is only associated with the `key` if it is non-`null`; *otherwise, the key is removed.*

This method returns the current (existing or computed) value associated with the specified `key`, or `null` if the `key` is absent or removed.

Table 15.6 summarizes typical scenarios when using these advanced key-based methods. Given the `value` associated with a key and the `resultValue` returned by the remapping function if it is executed, the action performed and the value returned by each method are shown for each scenario.

Table 15.6 *Summary of Scenarios Using Advanced Key-Based Operations*

The `value` associated with `key` in the map	The `resultValue` returned by the remapping function	`merge(key, givenValue, remappingBiFunction)` and `(givenValue != null)`	`compute(key, remappingBiFunction)`	`computeIfAbsent(key, mappingFunction)`	`computeIfPresent(key, remappingBiFunction)`
non-`null`	`null`	`remove(key)`, *returns* `null`.	`remove(key)`, *returns* `null`.	*No change. Returns* `value`.	`remove(key)`, *returns* `null`.
non-`null`	non-`null`	`put(key, resultValue)`, *returns* `resultValue`.	`put(key, result-Value)`, *returns* `resultValue`.	*No change. Returns* `value`.	`put(key, result-Value)`, *returns* `resultValue`.
`null`	`null`	`put(key, givenValue)`, *returns* `givenValue`.	`remove(key)`, *returns* `null`.	*No change. Returns* `null`.	*No change. Returns* `null`
`null`	non-`null`	`put(key, givenValue)`, *returns* `givenValue`.	`put(key, result-Value)`, *returns* `resultValue`.	`put(key, result-Value)`, *returns* `resultValue`.	*No change. Returns* `null`.
No mapping	`null`	`put(key, givenValue)`, *returns* `givenValue`.	*Not entered. Returns* `null`.	*Not entered. Returns* `null`.	*Not entered. Returns* `null`.
No mapping	non-`null`	`put(key, givenValue)`, *returns* `givenValue`.	`put(key, result-Value)`, *returns* `resultValue`.	`put(key, result-Value)`, *returns* `resultValue`.	*Not entered. Returns* `null`.

In the code snippets to illustrate each method, an emergency telephone number map, `etnMap`, is loaded with the same entries. This map is a `HashMap<Integer, String>` that is used to map emergency telephone numbers (`Integer`s) to countries (`String`s) where they are used. (*Disclaimer*: There is no guarantee that the information provided below is correct.)

Click here to view code image

```
Map<Integer, String> etnMap = new HashMap<>();
etnMap.put(112, "Norway");
etnMap.put(999, "UK");
etnMap.put(190, null);
etnMap.put(911, null);
// {112=Norway, 999=UK, 190=null, 911=null}
```

In the code below, six method calls are made corresponding to the column for each method in **Table 15.6**. Each method call shows the value returned by the method as a comment in the call statement.

The `merge()` method executes as follows:

- If the key is associated with the `null` value in the map, as shown at (3) and (4), or has no entry, as shown at (5) and (6), the key is associated with the *non-*`null` *given value*.
- Otherwise, the remapping two-arity function is computed.
  - If the result is `null`, the entry for the key is removed, as shown at (1).
  - If the result is non-`null`, the key is associated with the result, as shown at (2).

  Click here to view code image

```
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.merge(112, "Mordor", (oVal, value) -> null);      // (1) null, removed
etnMap.merge(999, "Mordor", (oVal, value) -> "Uganda");  // (2) Uganda, updated
etnMap.merge(190, "Mordor", (oVal, value) -> null);      // (3) Mordor, updated
etnMap.merge(911, "Mordor", (oVal, value) -> "USA");     // (4) Mordor, updated
etnMap.merge(100, "Mordor", (oVal, value) -> null);      // (5) Mordor, inserted
etnMap.merge(110, "Mordor", (oVal, value) -> "China");   // (6) Mordor, inserted
// After: {100=Mordor, 999=Uganda, 110=Mordor, 190=Mordor, 911=Mordor}
```

The `compute()` method executes as follows:

- The remapping two-arity function is executed.
  - If this result is non-`null`, as at (8), (10), and (12), this result is associated with the key whether or not the key has an entry in the map.
  - If this result is `null` and the key has an entry in the map, the entry is removed, as at (7) and (9).
  - If this result is `null` and the key has no entry in the map, no action is taken, as at (11).

  Click here to view code image

```
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.compute(112, (key, oVal) -> null);              // (7) null, removed
etnMap.compute(999, (key, oVal) -> "Uganda");          // (8) Uganda, updated
etnMap.compute(190, (key, oVal) -> null);              // (9) null, removed
etnMap.compute(911, (key, oVal) -> "USA");             // (10) USA, updated
etnMap.compute(100, (key, oVal) -> null);              // (11) null, no action
etnMap.compute(110, (key, oVal) -> "China");           // (12) China, inserted
// After: {110=China, 999=Uganda, 911=USA}
```

The `computeIfAbsent()` method executes as follows:

- If the key is associated with a `null` value or has no entry in the map, the remap-ping function is executed.
  - If the result is non-`null`, as at (16) and (18), this result is associated with the key.

Click here to view code image

```
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.computeIfAbsent(112, key -> null);              // (13) Norway, no change
etnMap.computeIfAbsent(999, key -> "Uganda");          // (14) UK, no change
etnMap.computeIfAbsent(190, key -> null);              // (15) null, no change
etnMap.computeIfAbsent(911, key -> "USA");             // (16) USA, updated
etnMap.computeIfAbsent(100, key -> null);              // (17) null, no action
etnMap.computeIfAbsent(110, key -> "China");           // (18) China, inserted
// After: {112=Norway, 110=China, 999=UK, 190=null, 911=USA}
```

The `computeIfPresent()` method executes as follows:

- If the key is associated with a non-`null` value in the map, as at (19) and (20), the remap-ping two-arity function is executed.
  - If the result is `null`, then the entry with the key is removed, as at (19).
  - If the result is non-`null`, then the key is associated with the result, as at (20).

Click here to view code image

```
// Before: {112=Norway, 999=UK, 190=null, 911=null}
etnMap.computeIfPresent(112, (key, oVal) -> null);      // (19) null, removed
etnMap.computeIfPresent(999, (key, oVal) -> "Uganda");  // (20) Uganda, updated
etnMap.computeIfPresent(190, (key, oVal) -> null);      // (21) null, no change
etnMap.computeIfPresent(911, (key, oVal) -> "USA");     // (22) null, no change
etnMap.computeIfPresent(100, (key, oVal) -> null);      // (23) null, no action
etnMap.computeIfPresent(110, (key, oVal) -> "China");   // (24) null, no action
// After: {999=Uganda, 190=null, 911=null}
```

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.

Click here to view code image

```
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.

Click here to view code image

```
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 §13.7, p. 711, where this method is illustrated. It is also used in Example 15.10, p. 844.

Click here to view code image

```
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 twoarity 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 §13.9, p. 717, 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 15.10, p. 844, illustrates collection views.

Click here to view code image

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

Create 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. An entry in the entry set view can be manipulated by the methods defined in the `Map.Entry<K, V>` interface.

Example 15.10, **p. 844**, provides an example of using collection views.

15.9 Map Implementations

Figure 15.3, **p. 787**, shows four implementations of the `Map<K, V>` interface found in the `java.util` package: `HashMap<K, V>`, `LinkedHashMap<K, V>`, `Hashtable<K, V>`, and `TreeMap<K, V>`.

The classes `HashMap<K, V>` and `Hashtable<K, V>` implement unordered maps. The class `LinkedHashMap<K, V>` implements ordered maps, and the class `TreeMap<K, V>` implements sorted maps (**p. 845**).

While the `HashMap<K, V>` class is not thread-safe and permits a `null` key and `null` values (analogous to the `LinkedHashMap<K, V>` class), the `Hashtable<K, V>` class is thread-safe and permits only non-`null` keys and values (see **Table 15.7**). The thread-safety provided by the `Hashtable<K, V>` class comes with a performance penalty. Thread-safe use of maps is also provided by the methods in the `Collections` class (**p. 856**). Like the `Vector<E>` class, the `Hashtable<K, V>` class is also a legacy class that has been retrofitted to implement the `Map<K, V>` interface.

Table 15.7 *Map Implementations*

Map	`null` as key	`null` as value	Kind of map	Thread-safe?
`HashMap<K,V>`	*Allowed*	*Allowed*	*Unordered*	*Not thread-safe*
`LinkedHashMap<K,V>` `extends HashMap<K,V>`	*Allowed*	*Allowed*	*Key insertion order/Access order*	*Not thread-safe*

Map	`null` as Key	`null` as Value	Kind of map	Thread-safe?
`Hashtable<K,V>`	*Not allowed*	*Not allowed*	*Unordered*	*Thread-safe*
`TreeMap<K,V>`	*Not allowed*	*Allowed*	*Key-sort order*	*Not thread-safe*

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 (§14.2, p. 744, and §14.3, p. 753).

The `LinkedHashMap<K, V>` implementation is a subclass of the `HashMap<K, V>` class. The relationship between the map classes `LinkedHashMap<K, V>` and `HashMap<K, V>` is analogous to the relationship between their counterpart set classes `LinkedHashSet<E>` and `HashSet<E>`. The entries of a `HashMap<K, V>` (analogous to a `HashSet<E>`) are unordered. The entries of a `LinkedHashMap<K, V>` (analogous to a `LinkedHashSet<E>`) are ordered. By default, the entries of a `LinkedHashMap<K, V>` 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 reinserted because no new entry is created if the key's entry already exists. The elements in a `LinkedHashSet<E>` are also at (element) insertion order. However, a `LinkedHashMap<K, V>` can also maintain its entries in *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<K, V>` class.

Both the `HashMap<K, V>` and the `LinkedHashMap<K, V>` classes provide comparable performance, but the `HashMap<K, V>` 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 are based on hashing 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<K, V>` can be slightly slower than in a `HashMap<K, V>`, 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<K, V>`, the iteration time is proportional to the size of the map— regardless of its capacity. However, for an underlying `HashMap<K, V>`, it is proportional to the capacity of the map.

The concrete map implementations override the `toString()` method. The standard text representation generated by the `toString()` method for a map is

{ key_1 = $value_1$, key_2 = $value_2$, ..., key_n = $value_n$ }

where each key_i and each $value_i$, where `1 <= i <= n`, is the text 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 map. Additional constructors create empty maps with given initial capacities and load factors. The `HashMap<K, V>` class provides the following constructors:

Click here to view code image

```
HashMap()
HashMap(int initialCapacity)
HashMap(int initialCapacity, float loadFactor)
```

Constructs an empty `HashMap`, using either a default or specified initial capacity and a load factor.

Click here to view code image

```
HashMap(Map<? extends K,? extends V> otherMap)
```

Constructs a new map containing the elements in the specified map.

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

Click here to view code image

```
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 15.10 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>`. It is instructive to compare this with the solution in **Example 15.11** that uses a `TreeMap<K,V>` that simplifies the solution further, when the entries are maintained in key-sort 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:
  Click here to view code image

```
// 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 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`, passed 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 straightforward solution below requires an explicit `null` check to determine whether the value returned by the `get()` method is `null`, or risks a `NullPointerException` at runtime when incrementing the `null` value in the `frequency` reference.
  Click here to view code image

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

  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 in the map, respectively. No explicit `null` check is necessary.
  Click here to view code image

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

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 which can be safely incremented.

Click here to view code image

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

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

Click here to view code image

```
groupFreqMap.compute(weightGroup, (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 `Map.keySet()` method returns a set view of keys, which is passed as an argument to a `TreeSet<E>` to create a sorted set—in this case, the natural ordering of `Integer`s is used.

Click here to view code image

```
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 an `Iterable<E>`, we can use the `forEach()` method that takes a consumer to print the histogram.

Click here to view code image

```
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 the corresponding number of `"*"`. The method `Collections.nCopies()` creates a list with an 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 15.10 *Using Maps*

Click here to view code image

```
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:

Click here to view code image

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

gives the following output:

Click here to view code image

```
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: *
```

15.10 Sorted Maps and Navigable Maps

The `SortedMap<K, V>` interface extends the `Map<K, V>` interface, and the `NavigableMap<K, V>` interface extends the `SortedMap<K, V>` interface. The two maps are analogs of the `SortedSet<E>` and the `NavigableSet<E>` interfaces, respectively.

The `SortedMap<K, V>` Interface

The `SortedMap<K, V>` interface extends the `Map<K, V>` interface to provide the functionality for implementing maps with *sorted keys*. Its operations are analogous to those of the `SortedSet<E>` interface (p. 810), applied to maps and keys rather than to sets and elements.

Click here to view code image

```
// First-last keys
K firstKey()                                    Sorted set: first()
K lastKey()                                     Sorted set: last()
```

Return the first (smallest) key and the last (largest) key in the sorted map, respectively, dependent on the key-sort order in the map. They throw a `NoSuchElementException` if the map is empty.

Click here to view code image

```
// Range-view operations
SortedMap<K,V> headMap(K toKey)                 Sorted set: headSet()
SortedMap<K,V> tailMap(K fromKey)               Sorted set: tailSet()
SortedMap<K,V> subMap(K fromKey, K toKey)       Sorted set: subSet()
```

Return different views analogous to those for a `SortedSet<E>`. The views returned are a portion of the map whose keys are strictly less than `toKey`, greater than or equal to `fromKey`, and between `fromKey` (inclusive) and `toKey` (exclusive), respectively. That is, these partial map views include `fromKey` if it is present in the map, but `toKey` is excluded.

Click here to view code image

```
// Comparator access
Comparator<? super K> comparator()
```

Returns the key comparator that defines the key-sort order, or `null` if the sorted map uses natural ordering for the keys.

The `NavigableMap<K, V>` Interface

Analogous to the `NavigableSet<E>` interface extending the `SortedSet<E>` interface, the `NavigableMap<K, V>` interface extends the `SortedMap<K, V>` interface with navigation methods to find the closest matches for specific search targets. The `NavigableMap<K, V>` interface replaces the `SortedMap<K, V>` interface and is the preferred choice when a sorted map is needed.

In addition to the methods of the `SortedMap<K, V>` interface, the `NavigableMap<K, V>` interface adds the *new* methods shown below, where the analogous methods from the `NavigableSet<E>` interface are also identified. Note that where a `NavigableMap<K, V>` method returns a `Map.Entry` object representing an entry, the corresponding `NavigableSet<E>` method returns an element of the set.

Click here to view code image

```
// First-last entries
// Remove
Map.Entry<K, V> pollFirstEntry()                Navigable set: pollFirst()
Map.Entry<K, V> pollLastEntry()                 Navigable set: pollLast()

// Examine
Map.Entry<K, V> firstEntry()
Map.Entry<K, V> lastEntry()
```

The `pollFirstEntry()` method *removes* and returns the first *entry*, and the `pollLastEntry()` method removes and returns the last *entry* currently in this navigable map. The entry is determined according to the ordering policy employed by the map—for example, natural ordering. Both return `null` if the navigable set is empty. The last two methods only retrieve, and do not remove, the value that is returned.

Click here to view code image

```
// Range-view operations
NavigableMap<K, V> headMap(K toKey,             Navigable set: headSet()
                     boolean inclusive)
NavigableMap<K, V> tailMap(K fromKey,           Navigable set: tailSet()
                     boolean inclusive)
NavigableMap<K, V> subMap(K fromKey,            Navigable set: subSet()
```

```
                        boolean fromInclusive,
                        K toKey,
                        boolean toInclusive)
```

These operations are analogous to the ones in the `SortedMap<K, V>` interface (p. 845), returning different views of the underlying navigable map, depending on the bound elements. However, the bound elements can be *excluded or included* by the operation, depending on the value of the `boolean` argument `inclusive`.

Click here to view code image

```
// Closest-matches
Map.Entry<K, V> ceilingEntry(K key)              Navigable set: ceiling()
K               ceilingKey(K key)
Map.Entry<K, V> floorEntry(K key)                Navigable set: floor()
K               floorKey(K key)
Map.Entry<K, V> higherEntry(K key)               Navigable set: higher()
K               higherKey(K key)
Map.Entry<K, V> lowerEntry(K key)                Navigable set: lower()
K               lowerKey(K key)
```

The ceiling methods return the least entry (or key) in the navigable map >= to the argument `key`. The floor methods return the greatest entry (or key) in the navigable map <= to the argument `key`. The higher methods return the least entry (or key) in the navigable map > the argument `key`. The lower methods return the greatest entry (or key) in the navigable map < the argument `key`. All methods return `null` if there is no such `key`.

Click here to view code image

```
// Navigation-views
NavigableMap<K, V> descendingMap()          Navigable set: descendingSet()
NavigableSet<K> descendingKeySet()
NavigableSet<K> navigableKeySet()
```

The first method returns a *reverse-order map view* of the entries in the navigable map. The second method returns a *reverse-order key set view* for the entries in the navigable map. The last method returns an *ascending-order key set view* for the entries in the navigable map.

The `TreeMap<K,V>` Class

The `TreeMap<K, V>` class is the analog of the `TreeSet<E>` class (p. 812), but in this case for maps. It provides an implementation that sorts its entries in a specific order (see also Figures 15.2 and 15.3, p. 786).

The `TreeMap<K, V>` class implements the `NavigableMap<K, V>` interface, and thereby the `SortedMap<K, V>` interface. By default, operations on sorted maps rely on the natural order-

ing of the keys. However, a total ordering can be specified by passing a customized comparator to the constructor.

A key in a `TreeMap<K, V>` cannot have the `null` value, but the value associated with a key in an entry can be `null`.

The `TreeMap<K, V>` implementation uses balanced trees, which deliver excellent performance for all operations. However, searching in a `HashMap<K, V>` can be faster than in a `TreeMap<K, V>`, as hashing algorithms usually offer better performance than the search algorithms for balanced trees.

The `TreeMap<K, V>` class provides four constructors, analogous to the ones in the `TreeSet<E>` class:

```
TreeMap()
```

A standard constructor used to create a new empty sorted map, according to the natural ordering of the keys.

Click here to view code image

```
TreeMap(Comparator<? super K> c)
```

A constructor that takes an explicit comparator for the keys, that is used to order the entries in the map.

Click here to view code image

```
TreeMap(Map<? extends K, ? extends V> map)
```

A constructor that can create a sorted map based on a map, according to the natural ordering of the keys of the specified map.

Click here to view code image

```
TreeMap(SortedMap<K, ? extends V> map)
```

A constructor that creates a new map containing the same entries as the specified sorted map, with the same ordering for the keys as the specified map.

Example 15.11 illustrates using navigable maps. It also prints a textual histogram like the one in **Example 15.10**, but using a `TreeMap<K, V>`, and in addition, it prints some statistics

about the navigable map. See also the output from running the example. Some remarks about **Example 15.11**:

- An empty `NavigableMap<Integer, Integer>` is created at (1), where the key is the weight group and the value is the frequency. The loop at (2) reads the weights specified as program arguments, and creates a frequency map, as described in **Example 15.10**.
- The method `printHistogram()` at (15) prints a histogram of the frequencies in a navigable map:
  Click here to view code image

  ```
  public static <K> void printHistogram(NavigableMap<K, Integer> freqMap) {...}
  ```

  It is a generic method with one type parameter, `K`, that specifies the type of the keys, and the type of the values (i.e., frequencies) is `Integer`. It prints the map and the number of entries in the map at (16) and (17), respectively. The `forEach()` method is invoked on the map at (18) to iterate over the *entries* in order to print the histogram, as explained in **Example 15.10**.
  Printing the histogram at (3) shows the number of entries ordered in ascending key order and the size of the map:
  Click here to view code image

  ```
  Group frequency map: {10=1, 60=1, 75=3, 80=1, 90=1, 95=2, 185=1}
  No. of weight groups: 7
  ...
  ```

- Calls to the methods `firstEntry()` and `lastEntry()` at (4) and (5):
  Click here to view code image

  ```
  out.println("First entry: " + groupFreqMap.firstEntry());              // (4)
  out.println("Last entry: " + groupFreqMap.lastEntry());               // (5)
  ```

  return the following entries, respectively:

  ```
  First entry: 10=1
  Last entry: 185=1
  ```

- Calls to the methods `floorEntry()` and `higherKey()` with the key value `77` at (6) and with the key value `90` at (7), respectively:
  Click here to view code image

  ```
  out.println("Greatest entry <= 77: "
              + groupFreqMap.floorEntry(77));                            // (6)
  out.println("Smallest key > 90: "
              + groupFreqMap.higherKey(90));                             // (7)
  ```

give the following output, respectively:

Click here to view code image

```
Greatest entry <= 77: 75=3
Smallest key > 90: 95
```

- Calls to the methods `tailMap(75, true)` and `headMap(75, false)` at (8) and (9) with the argument `75`:

Click here to view code image

```
...
printHistogram(groupFreqMap.tailMap(75, true));                    // (8)
...
printHistogram(groupFreqMap.headMap(75, false));                   // (9)
```

return the following map views, respectively:

Click here to view code image

```
Tail map (Groups >= 75): {75=3, 80=1, 90=1, 95=2, 185=1}
...
Head map (Groups <  75): {10=1, 60=1}
...
```

- The call to the `subMap()` method at (10)

Click here to view code image

```
printHistogram(groupFreqMap.subMap(                                // (10)
     groupFreqMap.firstEntry().getKey(), false,
     groupFreqMap.lastEntry().getKey(), false));
```

returns the map view with the first and the last entry excluded:

Click here to view code image

```
Frequency map (first and last entry excluded): {60=1, 75=3, 80=1, 90=1, 95=2}
...
```

- Polling of a navigable map is shown at (11). Polling is done directly on the navigable map, and the first entry in the map is always retrieved and removed by the `pollFirstEntry()` method at (12). A `while` loop is used to iterate over the entries in the map until the map is empty. For each entry, its key and its value is printed.

Click here to view code image

```
// Poll the navigable map:                                         // (11)
int sumValues = 0;
while (!groupFreqMap.isEmpty()) {
  Map.Entry<Integer, Integer> entry = groupFreqMap.pollFirstEntry();   // (12)
```

```
    Integer frequency = entry.getValue();
    sumValues += frequency;
    out.printf("%5s: %s%n", entry.getKey(), frequency);
  }
```

The number of weights—that is, the sum of the values in the map—is also calculated in the loop and printed at (13) and (14), respectively. After polling the map, the output shows that the map is empty.

Example 15.11 *Using Navigable Maps*

Click here to view code image

```
import java.util.Collections;
import java.util.Map;
import java.util.NavigableMap;
import java.util.TreeMap;
import static java.lang.System.out;

public class HistogramStats {
  public static void main(String[] args) {

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

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

    // Print statistics about the frequency map:
    out.print("Group frequency map: ");
    printHistogram(groupFreqMap);                                           // (3)

    out.println("First entry: " + groupFreqMap.firstEntry());               // (4)
    out.println("Last entry: " + groupFreqMap.lastEntry());                 // (5)
    out.println("Greatest entry <= 77: "
                + groupFreqMap.floorEntry(77));                             // (6)
    out.println("Smallest key > 90: "
                + groupFreqMap.higherKey(90));                              // (7)

    out.print("Tail map (Groups >= 75): ");
    printHistogram(groupFreqMap.tailMap(75, true));                         // (8)

    out.print("Head map (Groups <  75): ");
    printHistogram(groupFreqMap.headMap(75, false));                        // (9)

    out.print("Frequency map (first and last entry excluded): ");
    printHistogram(groupFreqMap.subMap(                                     // (10)
```

```
        groupFreqMap.firstEntry().getKey(), false,
        groupFreqMap.lastEntry().getKey(), false));

    // Poll the navigable map:                                          (11)
    out.println("Histogram (by polling):");
    int sumValues = 0;
    while (!groupFreqMap.isEmpty()) {
      Map.Entry<Integer, Integer> entry = groupFreqMap.pollFirstEntry();   // (12)
      Integer frequency = entry.getValue();
      sumValues += frequency;
      out.printf("%5s: %s%n", entry.getKey(), frequency);
    }
    out.println("Number of weights registered: " + sumValues);             // (13)
    out.println("Group frequency map after polling: " + groupFreqMap);     // (14)
  }

  // Prints a histogram for entries in a navigable map.
  public static <K> void printHistogram(NavigableMap<K, Integer> freqMap) {// (15)
    out.println(freqMap);                                                 // (16)
    out.println("No. of entries: " + freqMap.size());                     // (17)
    freqMap.forEach((k, v) ->                                             // (18)
        out.printf("%5s: %s%n", k,
                   String.join("", Collections.nCopies(v, "*"))));
  }
}
```

Running the program with the following arguments:

Click here to view code image

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

gives the following output:

Click here to view code image

```
Group frequency map: {10=1, 60=1, 75=3, 80=1, 90=1, 95=2, 185=1}
No. of entries: 7
   10: *
   60: *
   75: ***
   80: *
   90: *
   95: **
  185: *
First entry: 10=1
Last entry: 185=1
Greatest entry <= 77: 75=3
Smallest key > 90: 95
Tail map (Groups >= 75): {75=3, 80=1, 90=1, 95=2, 185=1}
```

```
No. of entries: 5
   75: ***

   80: *
   90: *
   95: **
  185: *
Head map (Groups <  75): {10=1, 60=1}
No. of entries: 2
   10: *

   60: *
Frequency map (first and last entry excluded): {60=1, 75=3, 80=1, 90=1, 95=2}
No. of entries: 5
   60: *

   75: ***
   80: *
   90: *
   95: **
Histogram (by polling):
   10: 1

   60: 1
   75: 3
   80: 1
   90: 1
   95: 2
  185: 1
Number of weights registered: 10
Group frequency map after polling: {}
```

Review Questions

15.9 Which of the following statements are true about maps?

Select the two correct answers.

a. The return type of the `values()` method is `Set<V>`.

b. Changes made in the set view returned by the `keySet()` method will be reflected in the underlying map.

c. The `Map<K, V>` interface extends the `Collection<E>` interface.

d. All keys in a map are unique.

e. All `Map<K, V>` implementations maintain the keys in some sort order.

15.10 Which of the following methods are defined by the `java.util.Map.Entry<K, V>` interface?

Select the three correct answers.

a. `K getKey()`

b. `K setKey(K value)`

c. `V getValue()`

d. `V setValue(V value)`

e. `void set(K key, V value)`

15.11 Given the following code:

Click here to view code image

```
import java.util.*;
public class TripleJump1 {
  public static void main(String[] args) {

    NavigableSet<String> set = new TreeSet<>(Collections.reverseOrder());
    Collections.addAll(set, "Step", "Jump", "Step", "Hop");
    System.out.println(set);
  }
}
```

What is the result?

Select the one correct answer.

a. `[Step, Step, Jump, Hop]`

b. `[Step, Jump, Hop]`

c. `[Hop, Jump, Step, Step]`

d. `[Hop, Jump, Step]`

e. The result is unpredictable.

f. The code will throw an exception at runtime.

15.12 Given the following code:

Click here to view code image

```
import java.util.*;
public class TripleJump2 {
  public static void main(String[] args) {

    NavigableSet<String> set1 = new TreeSet<>(Collections.reverseOrder());
    Collections.addAll(set1, "Step", "Jump", "Hop");
    NavigableSet<String> set2 = new TreeSet<>(set1);
    System.out.println(set2);
  }
}
```

What is the result?

Select the one correct answer.

a. `[Hop, Jump, Step]`

b. `[Step, Jump, Hop]`

c. The program will compile, but it will produce an unpredictable result when run.

d. The program will throw an exception at runtime.

e. The program will fail to compile.

15.13 Given the following code:

Click here to view code image

```
import java.util.*;
public class TripleJump3 {
  public static void main(String[] args) {

    NavigableSet<String> set1 = new TreeSet<>(Collections.reverseOrder());
    Collections.addAll(set1, "Step", "Jump", "Hop");
    NavigableSet<String> set2 = new TreeSet<>((Collection<String>)set1);
    System.out.println(set2);
  }
}
```

What is the result?

Select the one correct answer.

a. `[Hop, Jump, Step]`

b. `[Step, Jump, Hop]`

c. The program will compile, but it will produce an unpredictable result when run.

d. The program will throw an exception at runtime.

e. The program will fail to compile.

15.14 Given the following code:

Click here to view code image

```
import java.util.*;
public class TripleJump4 {
    public static void main(String[] args) {

    NavigableSet<String> set1 = new TreeSet<>(Collections.reverseOrder());
    Collections.addAll(set1, "Step", "Jump", "Hop");
    NavigableSet<String> set2 = new TreeSet<>((Collection<String>)set1);
    while (!set1.isEmpty()) {
      System.out.print(set1.pollLast() + " ");
    }
    while (!set2.isEmpty()) {
      System.out.print(set2.pollFirst() + " ");
    }
  }
}
```

What is the result?

Select the one correct answer.

a. `Hop Jump Step Hop Jump Step`

b. `Hop Jump Step Step Hop Jump`

c. `Step Jump Hop Step Jump Hop`

d. `Step Jump Hop Hop Jump Step`

e. The program will throw an exception at runtime.

15.15 What will be the output of the following program when compiled and run?

Click here to view code image

```
import java.util.*;
public class MapModify {
  public static void main(String[] args) {

    NavigableMap<String, Integer> grades = new TreeMap<>();
```

```
    grades.put("A",  5); grades.put("B", 10); grades.put("C", 15);
    grades.put("D", 20); grades.put("E", 25);

    System.out.print(grades.get(grades.firstKey()) + " ");
    System.out.print(sumValues(grades.headMap("D")) + " ");
    System.out.print(sumValues(grades.subMap("B", false, "D", true)) + " ");
    grades.subMap(grades.firstKey(), false, grades.lastKey(), false).clear();
    System.out.println(sumValues(grades));
  }
  public static <K, M extends Map<K, Integer>> int sumValues(M freqMap) {
    return freqMap.values().stream().mapToInt(i->i).sum();
  }
}
```

Select the one correct answer.

a. `5 50 35 30`

b. `5 30 35 30`

c. `5 30 25 30`

d. `5 30 35 75`

15.16 Which code, when inserted independently at (1), will result in the following output from the program: `{Soap=10, Salts=30}`?

Click here to view code image

```
import java.util.*;
public class Mapping {
  public static void main(String[] args) {

    NavigableMap<String, Integer> myMap
                  = new TreeMap<>(Collections.reverseOrder());
    myMap.put("Soap", 10); myMap.put("Shampoo", 20); myMap.put("Salts", 30);
    // (1) INSERT CODE HERE
    System.out.println(myMap);
  }
}
```

Select the three correct answers.

a.

Click here to view code image

```
for (Map.Entry<String, Integer> entry : myMap.entrySet())
  if (entry.getKey().equals("Shampoo"))
```

```
    myMap.remove("Shampoo");
```

b.

Click here to view code image

```
for (Iterator<String> iterator = myMap.keySet().iterator();
     iterator.hasNext();)
  if (iterator.next().equals("Shampoo"))
    iterator.remove();
```

c.

Click here to view code image

```
for (Iterator<String> iterator = myMap.keySet().iterator();
     iterator.hasNext();) {
  if (iterator.next().equals("Shampoo"))
    myMap.remove("Shampoo");
```

d.

Click here to view code image

```
for (Map.Entry<String, Integer> entry : myMap.entrySet())
  if (entry.getKey().equals("Shampoo"))
    myMap.remove(entry);
```

e.

Click here to view code image

```
myMap.subMap("Shampoo", true, "Shampoo", true).clear();
```

f.

Click here to view code image

```
myMap.compute("Shampoo", (k, v) -> null);
```

15.17 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class Valuables {
  public static void main(String[] args) {

    NavigableMap<Integer, Integer> iMap = new TreeMap<>();
    iMap.put(100, 1); iMap.put(200, 2); iMap.put(300, 3);
    int sumVal = 0;
    iMap.forEach((k, v) -> sumVal += v);
    System.out.println(sumVal);
  }
}
```

Select the one correct answer.

a. The program will compile and print `6` when run.

b. The program will fail to compile because the `forEach()` method requires a `Function<T, R>` and not a `BiFunction<T, U, R>`.

c. There is no method named `forEach` for maps, only for collections.

d. The program will fail to compile because the syntax of the lambda expression in the argument to the `forEach()` method is not correct.

e. None of the above

15.18 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class InitMap {
  public static void main(String[] args) {

    Map<Integer, NavigableSet<String>> etnMap = new TreeMap<>();
    etnMap.computeIfAbsent(911, key -> new TreeSet<>()).add("USA");
    etnMap.computeIfAbsent(911, key -> new TreeSet<>()).add("Canada");
    etnMap.computeIfAbsent(911, key -> new TreeSet<>()).add("Argentina");
    etnMap.computeIfAbsent(112, key -> new TreeSet<>()).add("Norway");
    System.out.println(etnMap);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile, but it will result in a runtime exception from one of the calls to the `computeIfAbsent()` method.

c. The program will compile and print the following when run: {112=[Norway], 911=[Argentina, Canada, USA]}

d. The program will compile and print the following when run: {112=[Norway], 911=[USA]}

e. The program will compile and print the following when run: `{112=Norway, 911=USA}`

f. None of the above

15.19 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.function.*;
public class Test13RQ13 {
  public static void main(String[] args) {

    Map<Integer, String> values = new HashMap<>();
    values.put(1, "ONE");   values.put(2, "TWO");
    values.put(3, "THREE"); values.put(4, "FOUR");
    values.replaceAll((k, v) -> {
      switch (k) {
        case 1: return "FIRST";
        case 2: return "SECOND";
        case 3: return "THIRD";
        case 4: return "FOURTH";
      }
      return "ZERO";
    });
    values.forEach((Integer x, String y) -> { System.out.print(y + " "); } );
  }
}
```

What is the result?

Select the one correct answer.

a. `ONE TWO THREE FOUR`

b. `FIRST SECOND THIRD FOURTH`

c. `1 2 3 4`

d. The program will compile, but it will produce no result when run.

e. The program will throw an exception at runtime.

f. The program will fail to compile.

15.11 The `Collections` Class

The Java Collections Framework also contains two utility 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 also throw a `NullPointerException` if the specified collection or array references passed to them are `null`.

The utility methods declared in the `Collections` *class are all* `public` *and* `static`*; therefore, these two modifiers will be omitted in their method header declarations in this section.*

Unmodifiable Views of Collections

Unmodifiable views of collections are created by the utility methods `unmodifiable`*Interface*`()` in the `Collections` class, where *Interface* can be any of the core interfaces in the Java Collections Framework. The name of these methods is a slight misnomer, as they create unmodified *views* of collections on the *underlying* or *backing collection* that is passed as an argument to the method.

Click here to view code image

```
<E> Collection<E>   unmodifiableCollection(Collection<? extends E> c)
<E> List<E>         unmodifiableList(List<? extends E> list)
<E> Set<E>          unmodifiableSet(Set<? extends E> set)
<E> SortedSet<E>    unmodifiableSortedSet(SortedSet<E> sortedSet)
<E> NavigableSet<E> unmodifiableNavigableSet(NavigableSet<E> navSet)
```

Return an *unmodifiable view* of the collection passed as an argument.

Query operations on the returned collection are delegated to the *underlying collection*, and changes in the underlying collection are reflected in the unmodifiable view.

Any attempt to modify the view of a collection will result in an `UnsupportedOperationException`.

The view collection returned by the `unmodifiableCollection()` method does not delegate the `equals()` and the `hashCode()` methods to the backing collection. Instead, the returned view uses the corresponding methods inherited from the `Object` class. This is to safeguard the contract of these methods when the backing collection is a set or a list. However, the views returned by the other methods above do not exhibit this behavior.

Click here to view code image

```
<K, V> Map<K, V>        unmodifiableMap(Map<? extends K, ? extends V> map)
<K, V> SortedMap<K, V>  unmodifiableSortedMap(SortedMap<K, ? extends V> sm)
<K, V> NavigableMap<K, V>
                  unmodifiableNavigableMap(NavigableMap<K, ? extends V> nm)
```

Return an *unmodifiable view* of the map passed as an argument.

Query operations on the view of the map are delegated to the *underlying map*, and changes in the underlying map are reflected in the unmodifiable view.

Any attempt to modify the returned map will result in an `UnsupportedOperationException`.

An *unmodifiable view of a collection* should not be confused with an *unmodifiable collection*. They are both unmodifiable—that is, they are *read-only* collections. No operations that can change them *structurally* are permitted (the famous `UnsupportedOperationException`). However, an unmodifiable view of a collection is *backed* by an *underlying collection*, but that is not the case for an unmodifiable collection. In the case of an unmodifiable view of a collection, any changes to the underlying collection are reflected in the unmodifiable view.

Examples of *unmodified collections* include the following:

- Unmodifiable lists created by `List.of()` and `List.copyOf()` methods (§12.2, p. 649). See also the comparison of unmodifiable lists and *list views* that are created by the `Arrays.asList()` method (§12.7, p. 660).
- Unmodifiable sets created by `Set.of()` and `Set.copyOf()` methods (p. 804).
- Unmodifiable maps created by `Map.of()`, `Map.ofEntries()`, and `Map.copyOf()` methods (p. 832).

The code below creates an *unmodifiable view* of a map whose values are mutable.

Click here to view code image

```
// Mutable courses:
StringBuilder mc1 = new StringBuilder("Java I");
StringBuilder mc2 = new StringBuilder("Java II");
StringBuilder mc3 = new StringBuilder("Java III");
StringBuilder mc4 = new StringBuilder("Java IV");

// Backing map:
Map<Integer, StringBuilder> backingMap = new HashMap<>();
backingMap.put(200, mc1); backingMap.put(300, mc2);
backingMap.put(400, mc3); backingMap.put(500, mc4);
```

Click here to view code image

```
// Unmodifiable view of a map:
Map<Integer, StringBuilder> unmodViewMap
               = Collections.unmodifiableMap(backingMap);
```

As in the case of unmodifiable collections, the code below throws an `UnsupportedOperationException` when an attempt is made to structurally change the unmodifiable view of a map.

Click here to view code image

```
 // UnsupportedOperationException at (1), (2), and (3):
unmodViewMap.put(100, new StringBuilder("Java Now"));
unmodViewMap.remove(400);
unmodViewMap.replace(200, new StringBuilder("First Java"));
```

However, changes to the backing map are reflected in the unmodifiable view:

Click here to view code image

```
backingMap.remove(200); backingMap.remove(400);
System.out.println("Backing map:       " + backingMap);
System.out.println("Unmodifiable view: " + unmodViewMap);
//  Backing map:       {500=Java Complete, 300=Java II}
//  Unmodifiable view: {500=Java Complete, 300=Java II}
```

Since the values in the unmodifiable view of a map are mutable, the code below shows that changing a value of an entry in the unmodifiable view is reflected in the backing map.

Click here to view code image

```
StringBuilder mutableCourse = unmodViewMap.get(500);  // Get the course.
mutableCourse.replace(5, 8, "Complete");              // Change the course name.
System.out.println("Backing map:       " + backingMap);
System.out.println("Unmodifiable view: " + unmodViewMap);
//  Backing map:       {400=Java III, 500=Java Complete, 200=Java I, 300=Java II}
//  Unmodifiable view: {400=Java III, 500=Java Complete, 200=Java I, 300=Java II}
```

Ordering Elements in Lists

In order to sort the elements of a collection by their *natural ordering* or by another *total ordering*, the elements must implement the `Comparable<E>` (§14.4, p. 761) or the `Comparator<E>` (§14.5, p. 769) interface, respectively.

The `Collections` class provides two generic static methods for sorting lists.

Click here to view code image

```
<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 the 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<E>`.

If the specified comparator is `null` then the natural ordering for the elements is used, requiring elements in this list to implement the `Comparable<E>` interface.

Click here to view code image

```
<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 method 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 equivalent to its namesake in the `Collections` class.

Click here to view code image

```
// 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 the natural ordering for the elements is used, requiring elements in this list to implement the `Comparable<E>` interface.

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.

Click here to view code image

```
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:

Click here to view code image

```
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 either the element type of the list must implement the `Comparable<E>` interface or a `Comparator<E>` must be provided. The following code sorts a list of `StringBuilder` according to their natural ordering, as the class `String-Builder` implements the `Comparable<StringBuilder>` interface.

Click here to view code image

```
List<StringBuilder> sbList = new ArrayList<>();
Collections.addAll(sbList, new StringBuilder("smallest"),
                   new StringBuilder("small"), new StringBuilder("smaller"));
Collections.sort(sbList);                // [small, smaller, smallest]
```

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.

Click here to view code image

```
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:

Click here to view code image

```
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 in the `Collections` class apply to *any* list, regardless of whether the elements are `Comparable<E>` or not:

```
void reverse(List<?> list)
```

Reverses the order of the elements in the list.

Click here to view code image

```
void rotate(List<?> list, int distance)
```

Rotates the elements toward the end of the list by the specified distance. A negative value for the `distance` will rotate toward the start of the list.

Click here to view code image

```
void shuffle(List<?> list)
void shuffle(List<?> list, Random rnd)
```

Randomly permutes the list—that is, *shuffles* the elements.

Click here to view code image

```
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.

Click here to view code image

```
// 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.

Click here to view code image

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

These 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.

Click here to view code image

```
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 order. Otherwise, the search results are *unpredictable*. The example below shows the results of the search when the list `str-List` 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.

Click here to view code image

```
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:

Click here to view code image

```
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*:

Click here to view code image

```
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:

Click here to view code image

```
<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)
```

The one-argument methods require that the elements have a natural order-ing—that is, are `Comparable<E>`. 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 a parameter results in a `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<E>` class (p. 810), and the methods `firstKey()` and `lastKey()` in the `SortedMap<K, V>` class (p. 845).

Replacing Elements in Collections

Both the `List<E>` interface and the `Collections` class define a `replaceAll()` method to replace elements in a collection, but they operate differently.

Click here to view code image

```
// Defined in the List<E> interface.
void replaceAll(UnaryOperator<E> operator)
```

With this method, each element is replaced with the result of applying the specified `operator` to that element of this list. Unary operators are covered in §13.10, p. 720.

Click here to view code image

```
// Defined in the Collections class.
<E> boolean replaceAll(List<E> list, E oldVal, E newVal) // Collections
```

Replaces all elements equal to `oldVal` with `newVal` in the `list`; it returns `true` if the `list` was modified.

The following code snippet illustrates how the `replaceAll()` method of the `List<E>` interface can be used to apply a `UnaryOperator` to each element of the list. The lambda expression at (1a) is executed for each string in the list `strList`, replacing each element with an uppercase version of the string. Equivalent method reference is used for the same purpose at (1b).

Click here to view code image

```
// Before: [biggest, big, bigger, Bigfoot]
strList.replaceAll(str -> str.toUpperCase());      // (1a)
strList.replaceAll(String::toUpperCase);           // (1b)
// After: [BIGGEST, BIG, BIGGER, BIGFOOT]
```

In contrast, the `replaceAll()` method of the `Collections` class can be used to replace all occurrences of a specific value in the collection with a new value. In the list `palindromes` of strings, the occurrence of the string `"road"` is replaced by the string `"anna"` in the method call below.

Click here to view code image

```
// Before: [eye, level, radar, road]
Collections.replaceAll(palindromes, "road", "anna");
// After: [eye, level, radar, anna]
```

The majority of the methods found in the `Collections` class that replace the elements of a collection operate on a `List`, while one method operates on arbitrary `Collection`s. They all change the contents of the collection in some way.

Click here to view code image

```
<E> boolean addAll(Collection<? super E> collection, E... elements)
```

Adds the specified `elements` to the specified `collection`. This is a convenient method for loading a collection with individually specified elements or an array. The method is annotated with `@SafeVarargs` because of the variable arity parameter. The annotation suppresses the heap pollution warning in its declaration and the unchecked generic array creation warning at the call sites.

Click here to view code image

```
<E> void copy(List<? super E> destination, List<? extends E> source)
```

Adds the elements from the `source` list to the `destination` list. Elements copied to the `destination` list will have the same index as in the `source` list. The `destination` list cannot be shorter than the `source` list. If it is longer, the remaining elements in the `destination` list are unaffected.

Click here to view code image

```
<E> void fill(List<? super E> list, E element)
```

Replaces all of the elements of the `list` with the specified `element`.

Click here to view code image

```
<E> List<E> nCopies(int n, E element)
```

Creates an immutable list with `n` copies of the specified `element`. The *same* reference value of the specified element is saved in the list for *all* references.

The `addAll()` method is a convenient method for loading an *existing* collection with a *few* individually specified elements or an array of small size. Several examples of its usage can be found in this chapter. The array passed should be an array of objects. Note also the autoboxing of the `int` values specified at (1) and (2). The `addAll()` method does not allow primitive arrays as a variable arity argument, as attempted at (3).

Click here to view code image

```
List<Integer> intList = new ArrayList<>();            // []
Collections.addAll(intList, 9, 1, 1);                 // (1) Varargs
// After: [9, 1, 1]
Collections.addAll(intList, new Integer[] {1, 1, 9}); // (2) An array of Integers
// After: [9, 1, 1, 1, 1, 9]
Collections.addAll(intList, new int[] {1, 9, 1});     // (3) Compile-time error!
```

As we can see at (2), the collection returned by the `Collections.addAll()` method is *not* of fixed size as more elements can be added to it. This is in contrast to the list returned by the `Array.asList()` method (§12.7, p. 659). However, the `addAll()` method of the `Collection<E>` interface can be used for adding an arbitrary collection to an existing collection.

When using the `Collections.copy()` method to copy elements from the source list to the destination list, the elements are copied to the *same* positional index in the destination list as they were in the source list.

Click here to view code image

```
List<String> dest = new ArrayList<>();
Collections.addAll(dest, "one", "two", "three", "four");// [one, two, three, four]
List<String> src = new ArrayList<>();
Collections.addAll(src, "I", "II", "III");              // [I, II, III]
Collections.copy(dest, src);                            // [I, II, III, four]
```

All elements of a list can be replaced with the *same* element, as shown at (1):

Click here to view code image

```
List<String> strList = new ArrayList<>();
Collections.addAll(strList, "liberty", "equality", "fraternity");
// Before: [liberty, equality, fraternity]
Collections.fill(strList, "CENSORED");                  // (1)
// After: [CENSORED, CENSORED, CENSORED]
```

Earlier we saw usage of the `Collections.nCopies()` method in **Example 15.10** and **Example 15.11**. The `for(;;)` loop below

Click here to view code image

```
for (int i = 0; i < 5; ++i)
  System.out.printf("%d %s%n", i, Collections.nCopies(i, "*"));
```

prints the following:

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

15.12 The `Arrays` Class

In this section we look at some selected utility methods from the `java.util.Arrays` class, in particular, for sorting, searching, and comparing arrays.

Creating *list views of arrays* using the `Arrays.asList()` method is covered in §12.7, p. 660, and can be compared with *unmodifiable lists* created using the factory methods `List.of()` and `List.copyOf()`.

Using the overloaded static method `Arrays.stream()` to create streams with arrays as the data source is covered in §16.4, p. 898.

The utility methods declared in the `Arrays` *class are all* `public` *and* `static`*. Therefore, these two modifiers will be omitted in their method header declarations in this section.*

Sorting Arrays

Sorting implies ordering the elements according to some ranking criteria, usually based on the *values* of the elements. The values of numeric data types are compared and ranked by using the relational operators that define the *numerical order* of the values. Objects are typically compared according to their *natural ordering* defined by their class. The class implements the `compareTo()` method of the `Comparable<E>` interface. The ordering defined by this method is called the *natural ordering* for the objects of the class, where `obj1.compareTo(obj2)` returns the following result:

- A *positive value* if `obj1` is *greater* than `obj2`
- The value `0` if `obj1` is *equal* to `obj2`
- A *negative value* if `obj1` is *less* than `obj2`

The wrapper classes for primitive values and the `String` class implement the `compareTo()` method (§8.3, p. 432), thereby giving their objects a natural ordering. Arrays with objects of these classes can readily be sorted as the sort algorithm can take advantage of their natural ordering.

The `Arrays` class provides enough overloaded versions of the `sort()` method to sort practically any type of array. The discussion on sorting lists (p. 858) is also applicable to sorting arrays.

Click here to view code image

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

The permitted *type* for elements includes `byte`, `char`, `double`, `float`, `int`, `long`, `short`, and `Object`.

These methods sort the elements in the array according to their *natural ordering*.

In the case of an array of objects being passed as an argument, the *objects* must be *mutually comparable* according to the *natural ordering* defined by the `Comparable<E>` interface; that is, it should be possible to compare any two objects in the array according to their natural ordering without throwing a `ClassCast-Exception`.

Click here to view code image

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

These two generic methods 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 subarray 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 the elements in parallel.

The experiment with a list of strings (p. 858) is repeated in the following with an array of strings, giving identical results. An array of strings is sorted according to different criteria.

Click here to view code image

```
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 `Arrays.sort()` methods above, respectively:

Click here to view code image

```
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).

Click here to view code image

```
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

A common operation on an array is to search the array for a given element, called the *key*. 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 (p. 861) is also applicable to searching in arrays.

Click here to view code image

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

The permitted *type* for elements includes `byte`, `char`, `double`, `float`, `int`, `long`, `short`, and `Object`.

These methods search for `key` in the `array` where the elements are sorted according to the natural ordering of the elements.

In the case where an array of objects is passed as an argument, the *objects* must be sorted in natural ordering, as defined by the `Comparable<E>` interface.

These 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.

Click here to view code image

```
<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)
```

These two generic methods 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.

An appropriate `import` statement should be included in the source code to access the `java.util.Arrays` class by its simple name. The experiment from p. 861 with a list of strings

is repeated here 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.

Click here to view code image

```
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 order. Searching in the `strArray` using reverse natural ordering when the array is sorted in natural ordering gives the wrong result:

Click here to view code image

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

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

Click here to view code image

```
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:

Click here to view code image

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

The method `binarySearch()` derives its name from the divide-and-conquer algorithm that it uses to perform the search. It repeatedly divides the remaining elements to be searched into two halves and selects the half containing the key in which to continue the search, until either the key is found or there are no more elements left to search.

Comparing Arrays

The `java.util.Arrays` class provides a rich set of static methods for comparing arrays of primitive data types and objects. In this section we only consider the basic methods for array equality, array comparison, and array mismatch. We encourage the curious reader to explore the `Arrays` class API for more flexible comparison of arrays.

Array Equality

The `Arrays.equals()` static method can be used to determine if two arrays are equal.

Click here to view code image

```
boolean equals(type[] a, type[] b)
```

The permitted *type* for elements includes all *primitive types* (`boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`) and `Object`.

Two arrays are considered *equal* if they contain the same elements in the same order—that is, they have the same length and corresponding pairs of elements are equal. Two array references are also considered equal if both are `null`.

Two primitive values `v1` and `v2` are considered *equal* if `new` *WrapperClass* `(v1).equals(new` *WrapperClass* `(v2))`, where *WrapperClass* is the wrapper class corresponding to their primitive data type.

Two objects `obj1` and `obj2` are considered *equal* if `Objects.equals(obj1, obj2)`.

Given two arrays `fruitBasketA` and `fruitBasketB` of strings, shown below graphically, we can conclude that they are equal, since they have the same length and corresponding pairs of fruits are equal.

Click here to view code image

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketB ==> [oranges, apples, plums, kiwi]
Equals: true
```

However, the arrays `fruitBasketA` and `fruitBasketC` below would not be considered equal. Although they have the same length and the same fruits, their corresponding pairs of fruits are *not* equal. The first index where this occurs is index 2.

Click here to view code image

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketC ==> [oranges, apples, kiwi, plums]
Equals: false
```

Obviously, the two arrays `fruitBasketA` and `fruitBasketE` that have different lengths are not equal, as we can see below:

Click here to view code image

```
Index               0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketE ==> [oranges, apples]
Equals: false
```

The following code demonstrates the examples provided above, where `System.out` is statically imported:

Click here to view code image

```
String[] fruitBasketA = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketB = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketC = { "oranges", "apples", "kiwi", "plums" };
String[] fruitBasketE = { "oranges", "apples" };
...
out.println("Equals: " + Arrays.equals(fruitBasketA, fruitBasketB)); // true
out.println("Equals: " + Arrays.equals(fruitBasketA, fruitBasketC)); // false
out.println("Equals: " + Arrays.equals(fruitBasketA, fruitBasketE)); // false
```

According to the `equals()` method, two `null` arrays are equal. However, the first statement below will not compile. The compiler issues an error, as the method call matches many overloaded methods named `equals` in the `Arrays` API. A cast is necessary to disambiguate the method call, as shown in the second statement.

Click here to view code image

```
out.println(Arrays.equals(null, null));             // Ambiguous method call.
                                                    //   Compile-time error!
out.println(Arrays.equals((String[]) null, null));  // true
```

Array Comparison

T`he compare()` method of the `Comparable<E>` interface allows two objects to be compared. The comparison relationship is generalized by the `Arrays.compare()` static method to *lexicographically compare* arrays.

Click here to view code image

```
int compare(type[] a, type[] b)
<T extends Comparable<? super T>> int compare(T[] a, T[] b)
```

The permitted *type* for elements includes all *primitive types* (`boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`).

These methods return the value 0 if the first and second arrays are equal; a value less than 0 if the first array is *lexicographically less than* the second array; and a value greater than 0 if the first array is *lexicographically greater than* the second array.

The second method only permits arrays whose objects implement the `compare-To()` method of the `Comparable<E>` interface.

Two `null` array references are considered equal, and a `null` array reference is considered lexicographically less than a non-`null` array reference.

Consider the case where the two arrays `diceA` and `diceD` that represent the result of rolling a dice a fixed number of times. The arrays have a *prefix* (`[5, 2]`) which is common to both arrays. At index 2, `diceA[2]` (value `6`) is *greater than* `diceD[2]` (value `3`). In this case, the `compare()` method returns a *positive* value to indicate that array `diceA` is *lexicographically greater* than array `diceD`. Note that the index 2 is equal to the length of the prefix `[5, 2]` and it is a valid index in both arrays. The prefix [5, 2] is called the *common prefix* of `diceA` and `diceD`.

```
Index      0  1  2  3
diceA ==> [5, 2, 6, 3]
diceD ==> [5, 2, 3]
Common prefix: [5,2]
Compare value: 1
```

In the example below, the `compare()` method returns a *negative* value to indicate that `diceA` is *lexicographically less* than `diceE` because `diceA[1]` (value `2`) is less than `diceE[1]` (value `6`), where the common prefix `[5]` has length 1.

```
Index      0  1  2  3
diceA ==> [5, 2, 6, 3]
diceE ==> [5, 6]
Common prefix: [5]
Compare value: -1
```

If the arrays share a common prefix, as in the examples above, the lexicographical comparison is determined by the corresponding pair of values at the index given by the length of the common prefix. The length of a common prefix is always greater than or equal to 0 and less than the minimum length of the two arrays. By this definition, the length of a common prefix can never be equal to the length of the two arrays. A common prefix of length 0 means that the elements at index 0 from each array determine the comparison result.

Where the prefix is equal to one of the arrays, we need to consider the lengths of the arrays:

- If the lengths are equal, as shown in the example below, the arrays must be lexicographically equal and the `compare()` method returns the value `0`.

```
Index      0  1  2  3
diceA ==> [5, 2, 6, 3]
diceB ==> [5, 2, 6, 3]
Prefix: [5, 2, 6, 3]
Compare value: 0
```

- If the lengths are *not* equal, as shown in the example below, lexicographical comparison is based on the lengths of the arrays. The longer array (`diceC`, length 4) is *lexicographically greater* than the shorter array (`diceD`, length 3). The `compare()` method returns a *positive* value. Note that the length of the prefix, 3 in this example, is a valid index in the longer array, `diceC`, but not in the shorter array, `diceD`.

```
Index      0  1  2  3
diceC ==> [5, 2, 3, 6]
diceD ==> [5, 2, 3]
Proper prefix: [5, 2, 3]
Compare value: 1
```

  In the example above, the shorter array `diceD` is a prefix for the longer array `diceC`. Such a prefix is called a *proper prefix*—that is, it is equal to one of the arrays, where the lengths of the arrays are different.

The following code can be used to demonstrate lexicographic comparison of arrays, where `System.out` is statically imported:

Click here to view code image

```
int[] diceA = { 5, 2, 6, 3 };
int[] diceB = { 5, 2, 6, 3 };
int[] diceC = { 5, 2, 3, 6 };
int[] diceD = { 5, 2, 3 };
int[] diceE = { 5, 6 };
...
out.println("Compare value: " + Arrays.compare(diceA, diceD)); // 1
out.println("Compare value: " + Arrays.compare(diceA, diceE)); // -1
out.println("Compare value: " + Arrays.compare(diceA, diceB)); // 0
out.println("Compare value: " + Arrays.compare(diceC, diceD)); // 1
```

Array Mismatch

The `Arrays.mismatch()` static method returns the *first* index where a mismatch occurs between two arrays. If there is no mismatch, the arrays are equal, and the value `-1` is re-

turned. The index returned is determined by the prefix of the arrays. The discussion on common and proper prefixes in the subsection on comparing arrays is highly relevant for determining mismatch (p. 869).

Click here to view code image

```
int mismatch(type[] a, type[] b)
```

The permitted *type* for elements includes all *primitive types* (`boolean` , `byte` , `char` , `double` , `float` , `int` , `long` , `short`) and `Object` .

This method returns the *index* of the *first* mismatch between two arrays, where `0 <=` *index* `<= Math.min(a.length, b.length)` . Otherwise, it returns `-1` if no mismatch is found.

The method throws a `NullPointerException` if either of the arrays is `null` .

The examples below illustrate how the mismatch is determined.

The two arrays `fruitBasketA` and `fruitBasketC` have a *common prefix* `[oranges, apples]` . Mismatch therefore occurs at the index given by the length of the prefix (2). The elements `fruitBasketA[2]` and `fruitBasketC[2]` mismatch—that is, the element strings are not equal. The `mismatch()` method returns the value `2` , the same as the length of the common prefix.

Click here to view code image

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketC ==> [oranges, apples, kiwi, plums]
Common prefix: [oranges, apples]
Mismatch index: 2
```

In the following example, the common prefix is empty. Its length is 0. The elements at index 0, `fruitBasketA[0]` and `fruitBasketG[0]` , mismatch. The index `0` is returned.

Click here to view code image

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketG ==> [apples]
Common prefix: []
Mismatch index: 0
```

In the next example, the array lengths are equal and the prefix is the same as both arrays. There is no mismatch between the two arrays. The value `-1` is returned.

Click here to view code image

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketB ==> [oranges, apples, plums, kiwi]
Prefix: [oranges, apples, plums, kiwi]
Mismatch value: -1
```

In the example below, the arrays have a *proper prefix*, `[oranges, apples]`, as the prefix is the same as one of the arrays and the arrays have different lengths. Mismatch therefore occurs at the index given by the length of the proper prefix (2) in the longer array, `fruitBasketA`. Index 2 is only valid in this array. The `mismatch()` method returns the index `2`.

Click here to view code image

```
Index                0       1       2     3
fruitBasketA ==> [oranges, apples, plums, kiwi]
fruitBasketE ==> [oranges, apples]
Proper prefix: [oranges, apples]
Mismatch index: 2
```

The following code can be used to demonstrate the index value returned by the `mismatch()` method in the examples above, where `System.out` is statically imported:

Click here to view code image

```
String[] fruitBasketA = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketB = { "oranges", "apples", "plums", "kiwi" };
String[] fruitBasketC = { "oranges", "apples", "kiwi", "plums" };
String[] fruitBasketE = { "oranges", "apples" };
String[] fruitBasketG = { "apples" };
...
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketC)); //2
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketG)); //0
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketB)); //-1
out.println("Mismatch index: "+Arrays.mismatch(fruitBasketA, fruitBasketE)); //2
```

Miscellaneous Utility Methods in the `Arrays` Class

The methods `toString()` and `fill()` (**Example 15.10**, **Example 15.11**) have previously been used in this chapter.

Click here to view code image

```
String toString(type[] array)
String deepToString(Object[] array)
```

Return 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—that is, it creates a *one-dimensional* text representation of an array. The second method *goes deeper* and creates a multidimensional text representation for an arrays of arrays—that is, arrays that have arrays as elements. The *type* can be any primitive type.

Click here to view code image

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

In these methods, the *type* can be any primitive type, or `Object`. The methods assign 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 below 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.

Click here to view code image

```
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 text representation. The `Arrays.deepToString()` method can be used to convert the contents of a *multidimensional* array to a text representation, where each element that is an array is also converted to a text representation.

Click here to view code image

```
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.

Click here to view code image

```
<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.

Click here to view code image

```
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]
```

Review Questions

15.20 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class WhatIsThis {
  public static void main(String[] args) {
    List<StringBuilder> list = new ArrayList<>();
    list.add(new StringBuilder("B"));
    list.add(new StringBuilder("A"));
    list.add(new StringBuilder("C"));
```

```
    list.sort(Collections.reverseOrder());
    System.out.println(list.subList(1,2));
  }
}
```

Select the one correct answer.

a. The program will compile. When run, it will print `[B]`.

b. The program will compile. When run, it will print `[B, A]`.

c. The program will compile. When run, it will throw an exception.

d. The program will fail to compile.

15.21 Which of the following statements are true about the following program?

Click here to view code image

```
import java.util.*;
public class InitOnly {
  public static void main(String[] args) {
    List<String> list1 = new ArrayList<>();
    Collections.addAll(list1, "Hi", "Hello");
    Collections.addAll(list1, "Howdy");
    System.out.println(list1);                    // (1)

    List<String> list2 = new ArrayList<>(list1);
    list2 = Arrays.asList("Hi", "Hello");
    list2 = Arrays.asList("Howdy");
    System.out.println(list2);                    // (2)
    List<String> list3 = new ArrayList<>();
    list3.addAll(list1);
    System.out.println(list3);                    // (3)

    List<String> list4 = new ArrayList<>(list2);
    System.out.println(list4);                    // (4)
  }
}
```

Select the four correct answers.

a. Line (1) will print `[Howdy]`.

b. Line (1) will print `[Hi, Hello, Howdy]`.

c. Line (2) will print `[Howdy]`.

d. Line (2) will print `[Hi, Hello, Howdy]`.

e. Line (3) will print `[Howdy]`.

f. Line (3) will print `[Hi, Hello, Howdy]`.

g. Line (4) will print `[Howdy]`.

h. Line (4) will print `[Hi, Hello, Howdy]`.

15.22 Which of the following statements are true about the following program?

Click here to view code image

```
import java.util.Arrays;
public class GetThatIndex {
  public static void main(String[] args) {

    if (args.length != 1) return;
    printIndex(args[0]);
  }

  public static void printIndex(String key) {
    String[] strings = {"small", "smaller", "smallest", "tiny"};
    System.out.println(Arrays.binarySearch(strings, key));
  }
}
```

Select the two correct answers.

a. The largest value ever printed by the `printIndex()` method is `3`.

b. The largest value ever printed by the `printIndex()` method is `4`.

c. The largest value ever printed by the `printIndex()` method is `5`.

d. The smallest value ever printed by the `printIndex()` method is `0`.

e. The smallest value ever printed by the `printIndex()` method is `-4`.

f. The smallest value ever printed by the `printIndex()` method is `-5`.

g. The smallest value ever printed by the `printIndex()` method is `-3`.

15.23 Given the following code:

Click here to view code image

```
import java.util.*;
public class Q36 {
  public static void main(String[] args) {
```

```
    String s1 = "London";
    String s2 = "Bergen";
    String s3 = "Saratov";
    Deque<String> trip = new ArrayDeque<>();
    String r1 = trip.pollFirst();
    trip.offerFirst(s1);
    trip.offerFirst(s2);
    String r2 = trip.pollFirst();
    String r3 = trip.peekFirst();
    trip.offerLast(s3);
    trip.offerLast(s1);
    String r4 = trip.pollLast();
    String r5 = trip.peekLast();
    System.out.println(r1 + " " + r2 + " " + r3 + " " + r4 + " " + r5);
  }
}
```

What is the result?

Select the one correct answer.

a. An `UnsupportedOperationException` is thrown at runtime.

b. A `NullPointerException` is thrown at runtime.

c. `null Bergen London London` Saratov

d. `null Bergen Bergen London London`

e. `Bergen London London Saratov`

f. `Bergen Bergen London London`

15.24 Given the following code:

Click here to view code image

```
import java.util.*;
public class Q37 {
  public static void main(String[] args) {
    Integer x = 1, y = 2, z = 3;
    Set<Integer> coordinates = new TreeSet<>();
    coordinates.add(x);
    coordinates.add(y);
    coordinates.add(y);
    coordinates.add(z);
    coordinates.remove(x);
    System.out.print(coordinates);
```

```
  }
}
```

What is the result?

Select the one correct answer.

a. `[1, 2, 3]`

b. `[2, 3]`

c. `[2, 2, 3]`

d. `[1, 2, 2, 3]`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

15.25 Given the following code:

Click here to view code image

```
import java.util.*;
public class Q38 {
  public static void main(String[] args) {
    List<Integer> prices = new ArrayList<>();
    prices.add(1);
    prices.add(2);
    prices.add(2, null);
    prices.add(3, 3);
    prices.add(2, 4);
    prices.set(2, 3);
    prices.remove(2);
    prices.add(2, 2);
    System.out.print(prices);
  }
}
```

What is the result?

Select the one correct answer.

a. `[1, 2, null, 3]`

b. `[1, 2, 3]`

c. `[1, 2, 2, 3]`

d. `[1, 2, 2, null, 3]`

e. `[1, 2, null, null]`

f. `[1, null, 2, 2, 3]`

g. The program will throw an exception at runtime.

h. The program will fail to compile.