Collections, Part I: `ArrayList<E>` 12

Chapter Topics

- Understanding the concept of a list as a collection
- The inheritance relationship between the `ArrayList<E>` class, the `List<E>` interface, and the `Collection<E>` interface in the Java Collections Framework
- Declaring and using references of the `ArrayList<E>` type
- Creating unmodifiable lists
- Creating, modifying, querying, and traversing `ArrayList`s
- Interoperability between arrays and `ArrayList`s
- Comparison of arrays and `ArrayList`s

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	§12.1, p. 644, *to* §12.8, p. 662.

- ❍ *Only* `ArrayList` *is covered in this chapter.*
- ❍ *For arrays, see* §3.9, p. 117.
- ❍ *For comparing elements, see* §14.4, p. 761, *and* §14.5, p. 769.
- ❍ *For list, set, map, and deque collections, see* **Chapter 15**, p. 781.

Java SE 11 Developer Exam Objectives

[5.2] Use a Java array and List, Set, Map and Deque collections, including convenience methods	§12.1, p. 644, *to* §12.8, p. 662.

- ❍ *Only* `ArrayList` *is covered in this chapter.*
- ❍ *For arrays, see* §3.9, p. 117.

Java SE 17 Developer Exam Objectives

- ❍ *For list, set, map, and deque collections, see* **Chapter 15**, **p. 781**.

A program manipulates data, so naturally, organizing and using data efficiently is important in a program. *Data structures* allow data to be organized in an efficient way. Java uses the term *collection* to mean a data structure that can maintain a group of objects so that the objects can be manipulated as a *single entity* or *unit*. Objects can be stored, retrieved, and manipulated as *elements* of a collection. The term *container* is also used in the literature for such data structures. Arrays are one example of such collections. Other examples include lists, sets, queues, and stacks, among many others.

The Java Collections Framework provides the support for collections in Java. This chapter only covers the core API of the `ArrayList<E>` class that implements dynamic lists. We will have more to say about the `ArrayList<E>` class when we discuss the other collections in the Java Collections Framework in **Chapter 15**, **p. 781**. Diving deep into the Java Collections Framework is a beneficial exercise that is highly recommended for all Java programmers.

As the collections in the Java Collections Framework are implemented as generic types, knowledge of at least the basics of generics in Java is essential to utilize these collections effectively (**Chapter 11**, **p. 563**).

12.1 Lists

Once an array is created, its length cannot be changed. This inflexibility can be a significant drawback when the amount of data to be stored in an array is not known a priori. In Java, the structures known as lists alleviate this shortcoming. Lists are collections that maintain their elements *in order* and can contain duplicates. The order of elements in a list is *positional order*, and individual elements can be accessed according to their position in the list. 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, analogous to accessing elements in an array. However, unlike in an array, the position of an element in a list can change as elements are inserted or deleted from the list—that is, as the list changes dynamically.

Sorting implies ordering the elements in a collection according to some *ranking criteria*, usually based on the *values* of the elements. However, elements is an `ArrayList` are maintained in the order they are inserted in the list, known as the *insertion order*. The elements in such a list are therefore *ordered*, but they are *not* sorted, as it is not the values of the elements that determine their ranking in the list. Thus ordering does *not* necessarily imply sorting.

Overview of the Java Collections Framework

The `Collection<E>` interface in the `java.util` package (also known as the Java Collections Framework) defines the general operations that a collection should provide (see **Figure 12.1**). Note that the `Collection<E>` interface extends the `Iterable<E>` interface, so all collections in this framework can be traversed using the `for(:)` loop. Other subinterfaces in the Java Collections Framework augment this interface to provide specific operations for particular kinds of collections. The `java.util.List<E>` interface extends the `java.util.Collection<E>` interface with the operations necessary to maintain the collection as a list. In addition to the operations inherited from the `java.util.Collection<E>` interface, the `java.util.List<E>` interface 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 creation of customized iterators to iterate over a list. For methods used in this chapter, we will indicate which interface they are defined in. The impatient reader can refer to **Chapter 15**, **p. 781**, at any time for more details on these interfaces.

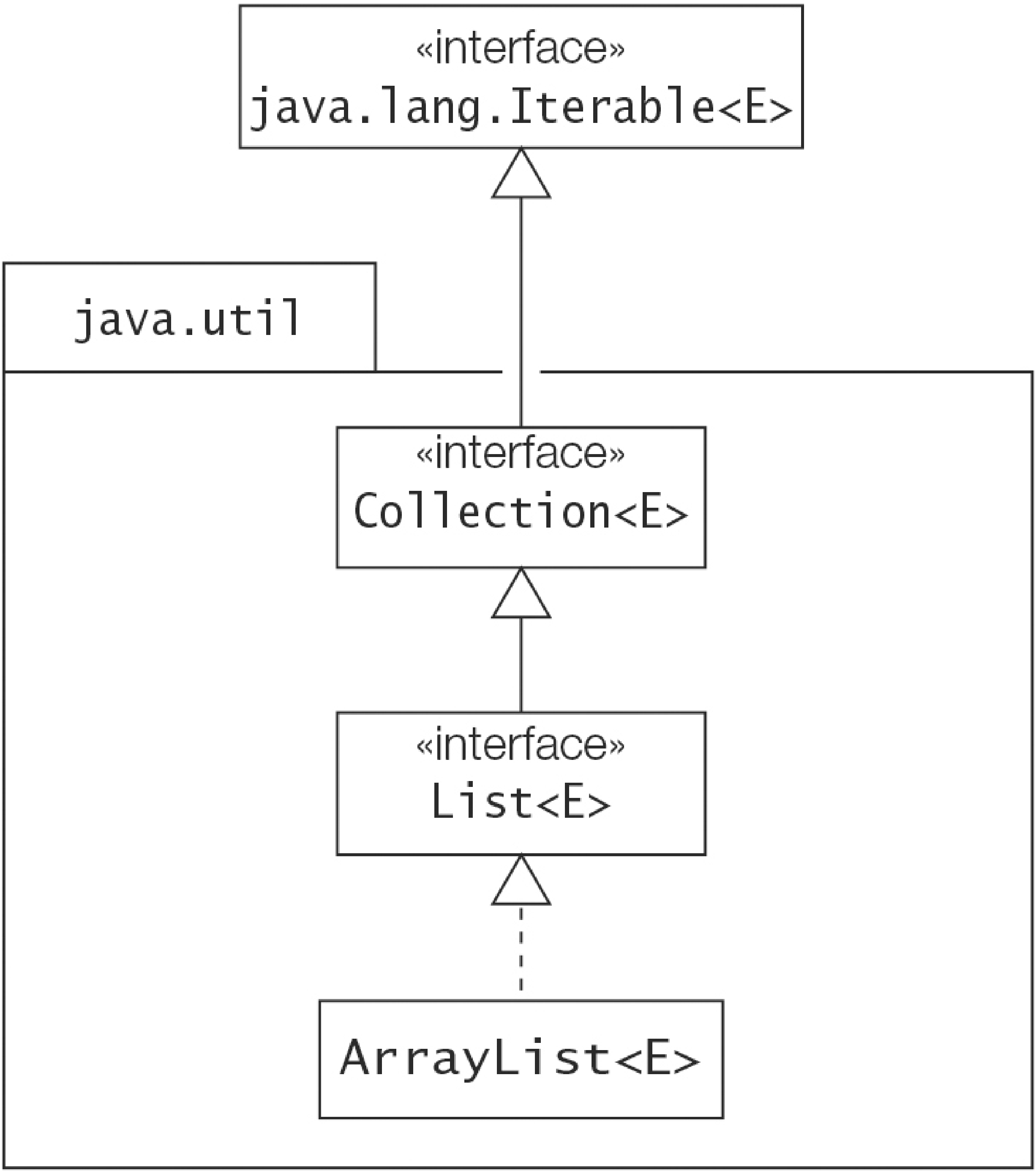

Figure 12.1 *Partial* `ArrayList` *Inheritance Hierarchy*

The *generic* class `java.util.ArrayList<E>` implements the `java.util.List<E>` interface. The type parameter `E` represents the type of the element in the list. Use of a generic type requires a *concrete* reference type to be substituted for the type parameter `E`. For example, the parameterized class `ArrayList<String>` is an `ArrayList` of `String`, where the type parameter `T` is substituted with the concrete class `String`.

The `ArrayList<E>` class is a dynamically resizable implementation of the `List<E>` interface using arrays (also known as *dynamic arrays*), providing fast random access (i.e., position-based access in constant time) and fast list traversal—very much like using an ordinary array. The `ArrayList<E>` class is not *thread-safe*; that is, its integrity can be jeopardized by concurrent access. The Java Collections Framework provides other implementations of the `List<E>` interface, but in most cases the `ArrayList<E>` implementation is the overall best choice for implementing lists.

12.2 Declaring References and Constructing `ArrayList` s

In the discussion that follows, we assume that any class or interface used from the `java.util` package has been imported with an appropriate `import` statement.

The code below illustrates how we can create an empty `ArrayList` of a specific element type, and assign its reference value to a reference:

Click here to view code image

```
ArrayList<String> palindromes = new ArrayList<String>(); // (1)
```

The element type is specified using angle brackets (`<>`). The reference `palindromes` can refer to any `ArrayList` whose element type is `String` . The type parameter `E` of the class `ArrayList` in **Figure 12.1** is replaced by the concrete class `String` . The compiler ensures that the reference `palindromes` can only refer to an `ArrayList` whose elements are of type `String` , and any operations on this list via this reference are type-safe.

The simplest way to construct an `ArrayList` is to use the zero-argument constructor to create an empty `ArrayList` , as shown in the declaration above. The zero-argument constructor creates an empty list with the initial capacity of 10. The *capacity* of a list refers to how many elements it can contain at any given time, not how many elements are actually in the list (which is called the *size*). The capacity of a list and its size can change dynamically as the list is manipulated. The `Array-List<String>` created at (1) can only contain elements of type `String` .

The assignment in the declaration statement (1) is valid because the types on both sides are assignment compatible—an `ArrayList` of `String` . The reference `palindromes` can now be used to manipulate the `ArrayList<String>` that it denotes.

We can use the *diamond operator* (`<>`) in the `ArrayList` creation expression on the right-hand side of the declaration statement. In this particular context, the compiler can infer the element type of the `ArrayList` from the declaration of the reference type on the left-hand side.

Click here to view code image

```
ArrayList<String> palindromes = new ArrayList<>(); // Using the diamond operator
```

However, if the diamond operator is omitted, the compiler will issue an *unchecked conversion warning*, as shown at (2) in the next code snippet. A new `ArrayList` is created based on an `ArrayList` of `Integer` that is passed as an argument to the constructor.

The `ArrayList` of `Integer` is created at (1). The reference `newList1` of type `ArrayList<String>` refers to an `ArrayList` whose element type is `Integer`, not `String`. The code at (2) compiles, but we get a `ClassCastException` at runtime at (3) when we retrieve an element from this list. The `get()` method call at (3) expects a `String` in the `ArrayList`, but gets an `Integer`. If the diamond operator is used, as shown at (4), the compiler reports a compile-time error, and the problem described at (3) cannot occur at runtime. By issuing an unchecked conversion warning at (2), the compiler alerts us to the fact that it cannot guarantee type-safety of the list created at (2).

Click here to view code image

```
ArrayList<Integer> intList = new ArrayList<>();         // (1) ArrayList of Integer
intList.add(10); intList.add(100); intList.add(1000);
ArrayList<String> newList1 = new ArrayList(intList);    // (2) Unchecked conversion
                                                        //     warning
System.out.println(newList1.get(0));                    // (3) ClassCastException!

ArrayList<String> newList2 = new ArrayList<>(intList); // (4) Compile-time error!
```

Best practices advocate *programming to an interface*. In practical terms, this means using references of an interface type to manipulate objects of a concrete class that implement this interface. Since the class `java.util.ArrayList<E>` implements the `java.util.List<E>` interface, the declaration at (1) can be written as shown in the next code snippet. This declaration is valid, since the reference value of a subtype object (`ArrayList<String>`) can be assigned to a reference of its supertype (`List<String>`).

Click here to view code image

```
List<String> palindromes = new ArrayList<>();    // (2) List<String> reference
```

This best practice provides great flexibility in substituting other objects for a task when necessary. The current concrete class can easily be replaced by another concrete class that implements the same interface. Only code creating objects needs to be changed. As it happens, the Java Collections Framework provides another implementation of lists: the `java.util.LinkedList<E>` class, which also implements the `List<E>` interface. If this class is found to be more conducive for maintaining palindromes in a list, we need simply change the name of the class in declaration (2), and continue using the reference `palindromes` in the program:

Click here to view code image

```
List<String> palindromes = new LinkedList<>(); // Changing implementation.
```

The `ArrayList<E>` class also provides a constructor that allows an empty `ArrayList` to be created with a specific initial capacity.

Click here to view code image

```
List<String> palindromes = new ArrayList<>(20); // Initial capacity is 20.
```

The `ArrayList` class provides the `add(E)` method to append an element to the end of the list. The new element is added after the last element in the list, thereby increasing the list size by 1.

Click here to view code image

```
palindromes.add("level"); palindromes.add("Ada"); palindromes.add("kayak");
System.out.println(palindromes);
```

The print statement calls the `toString()` method in the `ArrayList<E>` class to print the elements in the list. This `toString()` method applies the `toString()` method of the individual elements to create a text representation in the following default format:

```
[level, Ada, kayak]
```

A third constructor allows an `ArrayList` to be constructed from another collection. The following code creates a list of words from a list of palindromes. The order of the elements in the new `ArrayList<String>` is the same as that in the `Array-List<String>` that was passed as an argument in the constructor.

Click here to view code image

```
List<String> wordList = new ArrayList<>(palindromes);
System.out.println(wordList); // [level, Ada, kayak]
wordList.add("Naan");
System.out.println(wordList); // [level, Ada, kayak, Naan]
```

The next examples illustrate the creation of empty lists of different types of elements. The compiler ensures that operations on the `ArrayList` are type-safe with respect to the element type. Declaration (3) shows how we can create nested list structures (i.e., a list of lists), analogous to an array of arrays. Note that the diamond operator is not nested at (3). Declaration (4) shows that the element type cannot be a primitive type; rather, it must be a reference type.

Click here to view code image

```
List<StringBuilder> synonyms   = new ArrayList<>(); // List of StringBuilder
List<Integer> attendance       = new ArrayList<>(); // List of Integer
List<List<String>> listOfLists = new ArrayList<>(); // (3) List of List of String
List<int> frequencies          = new ArrayList<>(); // (4) Compile-time error!
```

When comparing arrays and `ArrayList`s, there is one other significant difference that concerns the subtype relationship.

Click here to view code image

```
Object[] objArray = new String[10];                  // (5) OK!
```

In declaration (5), since `String` is a subtype of `Object`, `String[]` is a subtype of `Object[]`. Thus we can manipulate the array of `String` using the `objArray` reference.

Click here to view code image

```
objArray[2] = "Green";                               // (6) OK!
objArray[1] = Integer.valueOf(2016);                 // ArrayStoreException!
```

The preceding assignment requires a runtime check to guarantee that the assignment is type compatible. Otherwise, an `ArrayStoreException` is thrown at runtime.

For the `ArrayList<E>`, the following declarations will not compile:

Click here to view code image

```
ArrayList<Object> objList1 = new ArrayList<String>();// (7) Compile-time error!
List<Object> objList2 = new ArrayList<String>();     // (8) Compile-time error!
```

Although `String` is a subtype of `Object`, it is not the case that an `ArrayList<String>` is a subtype of `ArrayList<Object>`. If this were the case, we could use the `objList1` reference to add other types of objects to the `ArrayList` of `String`, thereby jeopardizing its type-safety. Since there is no information about the element type `E` available at runtime to carry out a type compatibility check, as in the case of arrays, the subtype relationship is not allowed at (7). For the same reason, (8) will also not compile: `ArrayList<String>` is not a subtype of `List<Object>`. In general, the *subtype covariant relationship* does not hold for generic types. The Java language provides *wildcards* to overcome this restriction (§11.4, p. 579).

The `ArrayList<E>` constructors are summarized here:

Click here to view code image

```
ArrayList()
ArrayList(int initialCapacity)
ArrayList(Collection<? extends E> c)
```

The zero-argument constructor creates a new, empty `ArrayList` with an initial capacity of 10.

The second constructor creates a new, empty `ArrayList` with the specified initial capacity.

The third constructor creates a new ArrayList containing the elements in the specified collection. The declaration of the parameter c essentially means that parameter c can refer to any collection whose element type is E or a subtype of E. The new ArrayList<E> will retain any duplicates. The ordering in the ArrayList<E> will be determined by the traversal order of the iterator for the collection passed as an argument.

In a constructor call, the element type of the list is specified enclosed in angle brackets or by the diamond operator after the class name if it is to be inferred by the compiler. A raw `ArrayList` is created if the angle brackets are omitted, and the compiler will issue an unchecked warning.

Creating Unmodifiable Lists

Unmodifiable collections are useful to prevent a collection from accidently being modified, as doing so might cause the program to behave incorrectly. Such collections are also stored efficiently, as no bookkeeping is required to support any further modifications and data in the collection can be packed more densely since it can never change.

Here we look at how to create unmodifiable lists. Later we will discuss unmodifiable sets (§15.4, p. 804) and unmodifiable maps (§15.8, p. 832), and we will also contrast *unmodifiable collections* with *unmodifiable views of collections* (§15.11, p. 856).

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

- An unmodifiable list cannot be modified *structurally*; for example, elements cannot be added, removed, replaced, or sorted in such a list. Any such attempt will result in an `UnsupportedOperationException` to be thrown. However, if the elements themselves are mutable, the elements may appear modified.

- Although duplicates are allowed, unmodifiable lists do not allow `null` elements, and will result in a `NullPointerException` if an attempt is made to create them with the `null` elements.
- The order of the elements in an unmodifiable list is the same as the order of the arguments or the order of the elements in the array of the variable arity argument of the static method.
- An unmodifiable list can be serialized if its elements are serializable (§20.5, p. 1261).

Click here to view code image

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

The `of()` method is overloaded. The 11 overloaded methods are fixed-argument methods for accepting 0 to 10 arguments. They return an unmodifiable list containing the number of elements specified. They throw a `NullPointerException`, if an element is `null`. These overloaded methods are convenient for creating short lists.

Click here to view code image

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

This variable arity method returns an unmodifiable list containing an arbitrary number of elements specified by its variable arity argument. It throws a `NullPointerException`, if an element is `null` or if the array of the variable arity parameter is `null`.

The `@SafeVarargs` annotation suppresses the heap pollution warning in the method declaration and also the unchecked generic array creation warning at the call sites (§25.5, p. 1585).

Click here to view code image

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

This generic method returns an unmodifiable list 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 `NullPointerException` is thrown. If the specified collection is subsequently modified, the returned list will not reflect such modifications.

The code below shows that a list created by the `List.of()` method cannot be modified. The list returned is also not an instance of `ArrayList`.

Click here to view code image

```
List<String> list = List.of("Tom", "Dick", "Harriet");
//  list.add("Harry");                             // UnsupportedOperationException
//  list.remove(2);                                // UnsupportedOperationException
//  list.set(0, "Tommy");                          // UnsupportedOperationException
System.out.println(list);                          // [Tom, Dick, Harriet]
System.out.println(list instanceof ArrayList);    // false
```

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

Click here to view code image

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

For arguments up to 10, an appropriate fixed-arity `List.of()` method is called. For more than 10 arguments, the variable arity `List.of(E...)` method is called, passing an implicitly created array that contains the arguments.

Click here to view code image

```
List<Integer> intList1 = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);    // Fixed-arity
List<Integer> intList2 = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11); // Varargs
System.out.println(intList1);          // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
System.out.println(intList2);          // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

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

Click here to view code image

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

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

Click here to view code image

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

12.3 Modifying an `ArrayList<E>`

The `ArrayList<E>` class provides methods to append, insert, replace, and remove elements from a list. In addition, it has methods to modify the capacity of a list.

Adding Elements

The various add methods allow elements to be *appended at the end of a list* and also *inserted at a specified index* in the list.

Click here to view code image

```
boolean add(E element)                        From List<E> interface.
void add(int index, E element)                From List<E> interface.
```

The first method will append the specified element to the *end* of the list. It returns `true` if the collection was modified as a result of the operation.

The second method inserts the specified element at the specified index. If necessary, it shifts the element previously at this index and any subsequent elements one position toward the end of the list. The method will throw an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index > size()`).

The type parameter `E` represents the element type of the list.

Click here to view code image

```
boolean addAll(Collection<? extends E> c)              From List<E> interface.
boolean addAll(int index,                              From List<E> interface.
               Collection<? extends E> c)
```

The first method inserts the elements from the specified collection at the end of the list. The second method inserts the elements from the specified collection at the specified index; that is, the method splices the elements of the specified collection into the list at the specified index. These methods return `true` if any elements were added. Elements are inserted using an iterator of the specified collection (§15.2, p. 791). The second method will throw an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index > size()`).

The declaration of the parameter `c` essentially means that parameter `c` can refer to any collection whose element type is `E` or whose element type is a subtype of `E`.

Replacing Elements

The following methods replace elements in a list with new elements.

Click here to view code image

```
E set(int index, E element)                            From List<E> interface.
```

Replaces the element at the specified index with the specified element. It returns the previous element at the specified index. The method throws an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index >= size()`).

Click here to view code image

```
default void replaceAll(UnaryOperator<E> operator)     From List<E> interface.
```

Replaces each element of this list with the result of applying the unary operator (§13.10, p. 720) to that element. See also the `List<E>` interface (§15.3, p. 801).

Removing Elements

A summary of selected methods that can remove elements of a list is given here:

Click here to view code image

```
void clear()                                    From List<E> interface.
```

Deletes all elements from the list. The list is empty after the call, so it has size 0.

Click here to view code image

```
E remove(int index)                             From List<E> interface.
boolean remove(Object element)                  From List<E> interface.
```

The first method deletes and returns the element at the specified index. The method throws an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index >= size()`).

The second method removes the *first* occurrence of the element from the list, using object value equality. The method returns `true` if the call was successful. This method does not throw an exception if an element value is `null`, or if it is passed a `null` value.

Both methods will contract the list accordingly if any elements are removed.

Click here to view code image

```
boolean removeAll(Collection<?> c)              From List<E> interface.
boolean removeIf(Predicate<? super E> filter)   From Collection<E> interface.
```

The first method removes from this list all elements that are contained in the specified collection.

The second method removes from this list all elements that satisfy the filtering criteria defined by a lambda expression that implements the `Predicate<T>` functional interface (§13.6, p. 703). See also filtering a list (§15.2, p. 797).

Both methods return `true` if the call was successful. The list is contracted accordingly if any elements are removed.

Modifying Capacity

The following two methods can be used to modify the capacity of a list. There is no method that returns the current capacity of an `ArrayList<E>`.

```
void trimToSize()
```

Trims the capacity of this list to its current size.

Click here to view code image

```
void ensureCapacity(int minCapacity)                    From List<E> interface.
```

Ensures that the capacity of this list is large enough to hold at least the number of elements specified by the minimum capacity.

All the code snippets in this section can be found in **Example 12.1**, **p. 663**. The method `printListWithIndex()` at (16) in **Example 12.1** prints the elements prefixed with their index in the list, making it easier to see how the list changes structurally:

Click here to view code image

```
[0:level, 1:Ada, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]
```

We have seen that the `add(E)` method appends an element to the end of the list. The following code adds the strings from an array of `String` to an `ArrayList` of `String`. The output from **Example 12.1** at (2) shows how the elements are added at the end of the list.

Click here to view code image

```
System.out.println("\n(2) Add elements to list:");
for (String str : wordArray) {
  strList.add(str);
  printListWithIndex(strList);
}
```

We can insert a new element at a specific index using the overloaded method `add(int, E)`. The output from the following code shows how inserting an element at index 2 shifted the elements structurally in the list.

Click here to view code image

```
                             // [0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]
strList.add(2, "Java");      // Insert an element at index 2 in the list.
```

```
printListWithIndex(strList); // [0:level, 1:Ada, 2:Java, 3:kayak, 4:Bob,
                             //  5:Rotator, 6:Bob]
```

Note that an index value equal to 0 or the size of the list is always allowed for the method `add(int, E)`.

Click here to view code image

```
List<String> list1 = new ArrayList<>();   // []
list1.add(0, "First");                    // [First]
list1.add(list1.size(), "Last");          // [First, Last]
```

We can replace an element at a specified index using the `set(int, E)` method. The method returns the element that was replaced.

Click here to view code image

```
System.out.println("(3) Replace the element at index 1:");
String oldElement = strList.set(1, "Naan");
System.out.println("Element that was replaced: " + oldElement);   // "Ada"
printListWithIndex(strList); // [0:level, 1:Naan, 2:Java, 3:kayak, 4:Bob,
                             //  5:Rotator, 6:Bob]
```

We can remove or empty a list of all its elements using the `clear()` method:

Click here to view code image

```
// list1 is [First, Last].
list1.clear();                            // []
```

We can also remove elements from a list, with the list being contracted accordingly.

Click here to view code image

```
System.out.println("(4) Remove the element at index 0:");
System.out.println("Element removed: " + strList.remove(0));      // "level"
printListWithIndex(strList); // [0:Naan, 1:Java, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]

System.out.println("(5) Remove the first occurrence of \"Java\":");
System.out.println("Element removed: " + strList.remove("Java")); // true
printListWithIndex(strList); // [0:Naan, 1:kayak, 2:Bob, 3:Rotator, 4:Bob]
```

The `remove(int)` removes the element at the specified index. The method `remove(Object)` needs to search the list and compare the argument object with elements in the list for object value equality. This test requires that the argument object override the `equals()` method from the `Object` class, which merely determines reference value equality. The `String` class provides the appropriate `equals()` method. However, the following code will not give the expected result because the `StringBuilder` class does not provide its own `equals()` method.

Click here to view code image

```
List<StringBuilder> sbList = new ArrayList<>();
for (String str : wordArray)
  strList.add(str);
System.out.println(sbList); // [level, Ada, kayak, Bob, Rotator, Bob]
StringBuilder element = new StringBuilder("Ada");
System.out.println("Element to be removed: " + element);          // "Ada"
System.out.println("Element removed: " + sbList.remove(element)); // false
System.out.println(sbList); // [level, Ada, kayak, Bob, Rotator, Bob]
```

Once it is known that an `ArrayList<E>` will not grow in size, it might be a good idea to trim its capacity down to its size by calling the `trimToSize()` method, thereby minimizing the storage used by the `ArrayList<E>`. To reduce the number of times the capacity is increased when adding a large number of elements to the list, appropriate capacity can be set via the `ensureCapacity()` method before the operation.

Click here to view code image

```
sbList.trimToSize();           // Capacity is now same as size—that is, 6.
sbList.ensureCapacity(10000);  // Capacity is now large enough for 10000 elements.
                               // Size is still 6.
```

Primitive Values and `ArrayList`s

Since primitive values cannot be stored in an `ArrayList<E>`, we can use the wrapper classes to box such values first. In the following code, we create a list of `Integer` in which the `int` values are autoboxed in `Integer` objects and then added to the list. We try to delete the element with value `1`, but end up deleting the element at index 1 instead (i.e, the value `20`).

Click here to view code image

```
List<Integer> intList = new ArrayList<>();
intList.add(10); intList.add(20); intList.add(1);
System.out.println(intList);                          // [10, 20, 1]
System.out.println("Element to be removed: " + 1);    // 1
```

```
System.out.println("Element removed: " + intList.remove(1));  // 20
System.out.println(intList);                                   // [10, 1]
```

The method call

```
intList.remove(1)
```

has the signature

```
intList.remove(int)
```

This signature matches the overloaded method that removes the element at a specified index, so it is this method that is called at runtime. We say that this method is the *most specific* in this case. For the code to work as intended, the primitive value must be explicitly boxed.

Click here to view code image

```
System.out.println(intList);                                   // [10, 20, 1]
System.out.println("Element to be removed: " + 1);                        // 1
System.out.println("Element removed: " +
                   intList.remove(Integer.valueOf(1)));       // true
System.out.println(intList);                                   // [10, 20]
```

The method call

Click here to view code image

```
intList.remove(Integer.valueOf(1))
```

has the signature

```
intList.remove(Integer)
```

This call matches the overloaded `remove(Object)` method, since an `Integer` object can be passed to an `Object` parameter. This method is the most specific in this case, and is executed.

12.4 Querying an `ArrayList<E>`

A summary of useful methods that can be used to query a list is provided below.

Click here to view code image

```
int size()                                   From List<E> interface.
```

Returns the number of elements currently in the list. In a non-empty list, the first element is at index 0 and the last element is at `size()-1`.

Click here to view code image

```
boolean isEmpty()                            From List<E> interface.
```

Determines whether the list is empty (i.e., whether its size is 0).

Click here to view code image

```
E get(int index)                             From List<E> interface.
```

Returns the element at the specified *positional index*. The method throws an `IndexOutOfBoundsException` if the index is out of range (`index < 0 || index >= size()`).

Click here to view code image

```
boolean contains(Object element)             From List<E> interface.
```

Determines whether the argument object is contained in the collection, using object value equality. This is called the *membership test*.

Click here to view code image

```
int indexOf(Object o)                        From List<E> interface.
int lastIndexOf(Object o)                    From List<E> interface.
```

Return the indices of the first and last occurrences of the element that are equal (using object value equality) to the specified argument, respectively, if such an element exists in the list; otherwise, the value `-1` is returned. These methods provide *element search* in the list.

Click here to view code image

```
List<E> subList(int fromIndex, int toIndex)  From List<E> interface.
```

Returns a *view* of the list, which consists of the sublist of the elements from the index `fromIndex` to the index `toIndex-1` (i.e., a half-open interval). A view allows the range it represents in the underlying list to be manipulated. Any changes in the view are reflected in the underlying list, and vice versa. Views can be used to perform operations on specific ranges of a list.

Click here to view code image

```
boolean equals(Object o)                              From List<E> interface.
```

Compares the specified object with this list for object value equality. It returns `true` if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are equal according to object value equality.

The method `size()` returns the number of elements in the list, and the method `empty()` determines whether the list is empty.

Click here to view code image

```
// [Naan, kayak, Bob, Rotator, Bob]
System.out.println("The size of the list is " + strList.size());          // 5
boolean result = strList.isEmpty();
System.out.println("The list " + (result ? "is" : "is not") + " empty."); // false
```

The method `get(int)` retrieves the element at the specified index.

Click here to view code image

```
// [Naan, kayak, Bob, Rotator, Bob]
System.out.println("First element: " + strList.get(0);                 // Naan
System.out.println("Last element: " + strList.get(strList.size()-1));  // Bob
```

The `equals()` method of the `ArrayList` class can be used to compare two lists for equality with regard to size and corresponding elements being equal in each list.

Click here to view code image

```
List<String> strList2 = new ArrayList<>(strList);
boolean trueOrFalse = strList.equals(strList2);                        // true
```

The method `subList()` returns a *view* of a list—that is, a sublist of the list. As the view is backed by the underlying list, operations on the sublist will be reflected in the underlying list, as demonstrated by the following code:

Click here to view code image

```
out.println("Underlying list: " + strList); // [Naan, kayak, Bob, Rotator, Bob]
List<String> strList3 = strList.subList(1, 4);
out.println("Sublist before remove: " + strList3);    // [kayak, Bob, Rotator]
out.println("Remove: " + strList3.get(0));  // "kayak"
strList3.remove(0);                         // Remove element at index 0
out.println("Sublist after remove: " + strList3);     // [Bob, Rotator]
out.println("Underlying list: " + strList); // [Naan, Bob, Rotator, Bob]
```

The membership test is carried out by the `contains(Object)` method. We can find the index of a specified element in the list by using the `indexOf()` and `lastIndexOf()` methods.

Click here to view code image

```
boolean found = strList.contains("Naan");     // true
int index = strList.indexOf("Bob");           // 2
index = strList.indexOf("BOB");               // -1 (Not found)
index = strList.lastIndexOf("Bob");           // 4 (Last occurrence)
```

Again, these methods require that the element type provide a meaningful `equals()` method for object value equality testing.

12.5 Iterating Over an `ArrayList<E>`

Various methods for iterating over collections are discussed in §15.2, p. 791. Here we look at a very common task of iterating over a list to perform some operation on each element of the list.

We can use positional access to iterate over a list with the `for(;;)` loop. The generic method `printListWithIndex()` in **Example 12.1** uses the `for(;;)` loop to create a new `ArrayList` of `String` that contains each element of the argument list prefixed with the index of the element.

Click here to view code image

```
public static <E> void printListWithIndex(List<E> list) {
  List<String> newList = new ArrayList<>();
  for (int i = 0; i < list.size(); i++) {
    newList.add(i + ":" + list.get(i));
  }
```

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

Sample output from the method call `printListWithIndex(strList)` is shown here:

Click here to view code image

```
[0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]
```

The method `printListWithIndex()` in **Example 12.1** can print *any* list in this format. Its header declaration says that it accepts a list of element type `E`. The element type `E` is determined from the method call. In the preceding example, `E` is determined to be `String`, as a `List` of `String` is passed in the method call.

Since the `ArrayList<E>` class implements the `Iterable<E>` interface (i.e., the class provides an iterator), we can use the `for(:)` loop to iterate over a list.

```
for (String str : strList) {
  System.out.print(str + " ");
}
```

The `ArrayList<E>` also provides specialized iterators to iterate over a list (§15.3, p. 801).

For performing a given action on each element of a list, the `forEach()` method can be used (§15.2, p. 796), where the action is specified by a consumer (§13.7, p. 709).

One pertinent question to ask is how to remove elements from the list when iterating over the list. The `for(:)` loop does not allow the list structure to be modified:

Click here to view code image

```
for (String str : strList) {
  if (str.length() <= 3) {
    strList.remove(str);                  // Throws ConcurrentModificationException
  }
}
```

We can use positional access in a loop to iterate over the list, but we must be careful in updating the loop variable, as the list contracts when an element is removed. Better solutions for this purpose are discussed in §15.2, p. 796.

12.6 Converting an `ArrayList<E>` to an Array

The following methods are specified by the `Collection<E>` interface and can be used to convert a collection to an array. List and set implementations in the `java.util` package provide customized versions of the first two methods for this purpose. In this section we consider how to convert lists to arrays.

Click here to view code image

```
Object[] toArray()                          From Collection<E> interface.
<T> T[] toArray(T[] a)                      From Collection<E> interface.
```

The first method returns an array of type `Object` filled with all the elements of a collection. The returned array can be modified independently of the list from which it was created.

The second method is a generic method that stores the elements of a collection in an array of type `T`. If the specified array is big enough, the elements are stored in this array. If there is room to spare in the array—that is, if the length of the array is greater than the number of elements in the collection—the element found immediately after storing the elements of the collection is set to the `null` value 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)    From Collection<E> interface.
```

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.

See also array operations in the `Collection<E>` interface (§15.2, p. 798).

The actual element type of the elements in the `Object` array returned by the first `toArray()` method can be any subtype of `Object`. It may be necessary to cast the

`Object` reference of an element to the appropriate type, as in the following code:

Click here to view code image

```
System.out.println("(15) Convert list to array:");
Object[] objArray = strList.toArray();                      // Object[]
System.out.println("Object[] length: " + objArray.length); // 5
System.out.print("Length of each string in the Object array: ");
for (Object obj : objArray) {
  String str = (String) obj;                                // Cast required.

  System.out.print(str.length() + " ");
}
System.out.println();
```

The generic `toArray()` method returns an array of type `T`, when it is passed an array of type `T` as an argument. In the following code, the array of `String` that is returned has the same length as the size of the list of `String`, even though a `String` array of length 0 was passed as an argument:

Click here to view code image

```
String[] strArray = strList.toArray(new String[0]);         // String[]
System.out.println("String[] length: " + strArray.length);  // 5
System.out.print("Length of each string in the String array: ");
for (String str : strArray) {
  System.out.print(str.length() + " ");
}
System.out.println();
```

12.7 Creating List Views

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

Click here to view code image

```
@SafeVarargs <E> List<E> asList(E... elements)
```

From `Arrays` class.

Returns a *fixed-size list view* that is backed by the *array* corresponding to the variable arity parameter `elements`. The method is annotated with `@SafeVarargs` because of the variable arity parameter. The annotation suppresses the heap pollution warning in its declaration and also unchecked generic array creation warning at the call sites (§25.5, p. 1585).

Changes to the elements of the list view are reflected in the array, and vice versa. The list view is said to be *backed* by the array. The size of the list view is equal to the array length and *cannot* be changed. The iterator for a list view does not support the `remove()` method.

The code below illustrates use of the `asList()` method. The `list1` at (1) is backed by the `array1`. The `list2` is backed by an implicit array of `Integer` at (2). An array of a primitive type cannot be passed as an argument to this method, as evident by the compile-time error at (3). However, the `Collections.addAll()` method provides better performance when adding a few elements to an *existing* collection.

Click here to view code image

```
Integer[] array1 = new Integer[] {9, 1, 1};
List<Integer> list1 = Arrays.asList(array1);          // (1) A list of Integer
List<Integer> list2 = Arrays.asList(9, 1, 1);         // (2) Varargs

int[] array2 = new int[] {9, 1, 1};                   // An array of int
// List<Integer> intList3 = Arrays.asList(array2);    // (3) Compile-time error!
```

Various operations on the `list1` show how changes are reflected in the backing `array1`. Elements cannot be added to the list view (shown at (4)), and elements cannot be removed from the list view (shown at (9)). An `UnsupportedOperationException` is thrown in both cases. An element at a given position can be changed, as shown at (5). The change is reflected in the `list1` and the `array1`, as shown at (6) and (7), respectively. A sublist view is created from the `list1` at (8), and sorted at (10). The changes in the `sublist1` are reflected in the `list1` and the backing `array1`.

Click here to view code image

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

```
System.out.println(list1);                      // [1, 10, 1]
System.out.println(Arrays.toString(array1)); // [1, 10, 1]
```

The code below shows how duplicates can be eliminated from an array:

Click here to view code image

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

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

Comparing Unmodifiable Lists and List Views

There are subtle differences to be aware of between unmodifiable lists and list views.

- *Backing an array*
  The `Arrays.asList()` method returns a *fixed-size list view* that is backed by the array passed as an argument so that any changes made to the array are reflected in the view list as well. This is not true of the `List.of()` and `List.ofCopy()` methods, as they create *unmodifiable lists* which are *not backed* by any argument array that is passed either explicitly or implicitly as a variable arity parameter.
  In the code below, we see that the list view returned by the `Arrays.asList()` method reflects the change at (1) in its backing array, but not the unmodifiable list returned by the `List.of()` method at (2) when its argument array is modified.
  Click here to view code image

```
Integer[] yrArray1 = {2020, 2021, 2022};
List<Integer> yrlist1 = Arrays.asList(yrArray1);
yrArray1[0] = 2019;                                    // Modify the array
out.println("yrArray1: " + Arrays.toString(yrArray1)); //     [2019, 2021, 2022]
out.println("yrlist1: " + yrlist1);                    // (1) [2019, 2021, 2022]

Integer[] yrArray2 = {2020, 2021, 2022};
List<Integer> yrlist2 = List.of(yrArray2);
yrArray2[0] = 2019;                                    // Modify the array
out.println("yrArray2: " + Arrays.toString(yrArray2)); //     [2019, 2021, 2022]
out.println("yrlist2: " + yrlist2);                    // (2) [2020, 2021, 2022]
```

- *Mutability*

  The list view returned by the `Arrays.asList()` method is *mutable*, but it cannot be structurally modified. In contrast, the unmodifiable list returned by the `List.of()` method is *immutable*.

  In the code below, only the list view returned by the `Arrays.asList()` method can be modified as shown at (1), but an attempt to modify the unmodifiable list returned by the `List.of()` method at (2) throws an exception.

  Click here to view code image

```
List<Integer> yrList3 = Arrays.asList(2020, 2021, 2022);
yrList3.set(2, 2023);                        // (1) OK
out.println(yrList3);                        // [2020, 2021, 2023]

List<Integer> yrlist4 = List.of(2020, 2021, 2022);
yrlist4.set(2, 2023);                        // (2) UnsupportedOperationException
```

  However, both lists will throw an exception if an attempt is made to change them *structurally*—that is, add or remove elements from the list:

  Click here to view code image

```
yrList3.add(2050);                           // UnsupportedOperationException
yrlist4.remove(0);                           // UnsupportedOperationException
```

- *The* `null` *value*

  The `Arrays.asList()` method allows `null` elements, whereas the `List.of()` and `List.ofCopy()` methods do not.

  Click here to view code image

```
List<Integer> yrList5 = Arrays.asList(2020, 2021, null);  // OK.
List<Integer> yrlist6 = List.of(2020, 2021, null);        // NullPointerExceptior
```

  The behavior of the `List.contains()` method when passed the `null` value is dependent on which method created the list.

  Click here to view code image

```
boolean flag1 = Arrays.asList(2021, 2022).contains(null); // OK.
boolean flag2 = List.of(2021, 2022).contains(null);       // NullPointerExceptior
```

12.8 Arrays versus `ArrayList`s

Table 12.1 summarizes the differences between arrays and `ArrayList`s.

Table 12.1 *Summary of Arrays versus* ArrayLists

	Arrays	ArrayList
Construct support	Built into the language.	Provided by the generic class ArrayList<E>.
Initial length/size specification	Length is specified in the array construction expression directly or indirectly by the initialization block.	Cannot specify the size at construction time. However, initial capacity can be specified.
Length/size	The length of an array is static (fixed) once it is created. Each array has a public final int field called length. (The String and the StringBuilder class provide the method length() for this purpose.)	Both size and capacity can change dynamically. ArrayList<E> provides the method size() to obtain the current size of the list.
Element type	Primitive and reference types.	Only reference types.
Operations on elements	An element in the array is designated by the array name and an index using the [] operator, and can be used as a simple variable.	The ArrayList<E> class provides various methods to add, insert, replace, retrieve, and remove elements from a list.
Iterator	Arrays do not provide an iterator, apart from using the for(:) loop for traversal.	The ArrayList<E> class provides customized iterators for lists, in addition to the for(:) loop for iterating over the elements (§15.2, p. 791).
Generics	Cannot create arrays of generic types using the new operator. Runtime check required for storage at runtime.	ArrayList<E> is a generic type. Can create parameterized ArrayLists of reference types using the new operator. No runtime check required for storage at runtime, as type-safety is checked at compile time.

Subtype relationship	Subtype relationship between two reference types implies subtype relationship between arrays of the two types—that is, element subtype relationship implies array subtype relationship.	Subtype relationship between two reference types does not imply covariance relationship between `ArrayLists` of the two types—that is, element subtype relationship does not imply list subtype relationship.

Sorting	`java.util.Arrays.sort(array)`
	`java.util.Arrays.sort(array, comparator)`
	(§15.12, p. 864)
		`java.util.Collections.sort(list)`
		`java.util.Collections.sort(list, comparator)`
		`java.util.List.sort(comparator)`
		(§15.11, p. 856)

Text representation	`java.util.Arrays.toString(array)`	`list.toString()`

Example 12.1 is a collection of code snippets used throughout this chapter to illustrate the various methods of the `ArrayList<E>` class.

Example 12.1 *Using an* `ArrayList`

Click here to view code image

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

import static java.lang.System.out;

public class ArrayListMethods {

  public static void main(String[] args) {

    String[] wordArray = { "level", "Ada", "kayak", "Bob", "Rotator", "Bob" };

    out.println("(1) Create an empty list of strings:");
    List<String> strList = new ArrayList<>();
    printListWithIndex(strList);

    out.println("\n(2) Add elements to list:");
    for (String str : wordArray) {
      strList.add(str);
      printListWithIndex(strList);
    }
```

```
        out.println("Insert an element at index 2 in the list:");
        strList.add(2, "Java");
        printListWithIndex(strList);

        out.println("\n(3) Replace the element at index 1:");
        String oldElement = strList.set(1, "Naan");
        out.println("Element that was replaced: " + oldElement);
        printListWithIndex(strList);

        out.println("\n(4) Remove the element at index 0:");
        out.println("Element removed: " + strList.remove(0));
        printListWithIndex(strList);

        out.println("\n(5) Remove the first occurrence of \"Java\":");
        out.println("Element removed: " + strList.remove("Java"));
        printListWithIndex(strList);

        out.println("\n(6) Determine the size of the list:");
        out.println("The size of the list is " + strList.size());

        out.println("\n(7) Determine if the list is empty:");
        boolean result = strList.isEmpty();
        out.println("The list " + (result ? "is" : "is not") + " empty.");

        out.println("\n(8) Get the element at specific index:");
        out.println("First element: " + strList.get(0));
        out.println("Last element: " + strList.get(strList.size() - 1));

        out.println("\n(9) Compare two lists:");
        List<String> strList2 = new ArrayList<>(strList);
        boolean trueOrFalse = strList.equals(strList2);
        out.println("The lists strList and strList2 are"
            + (trueOrFalse ? "" : " not") + " equal.");
        strList2.add(null);
        printListWithIndex(strList2);
        trueOrFalse = strList.equals(strList2);
        out.println("The lists strList and strList2 are"
            + (trueOrFalse ? "" : " not") + " equal.");

        out.println("\n(10) Sublists as views:");
        out.println("Underlying list: " + strList); // [Naan, kayak, Bob, Rotator, Bob]
        List<String> strList3 = strList.subList(1, 4);
        out.println("Sublist before remove: " + strList3);    // [kayak, Bob, Rotator]
        out.println("Remove: " + strList3.get(0));  // "kayak"
        strList3.remove(0);                         // Remove element at index 0
        out.println("Sublist after remove: " + strList3);     // [Bob, Rotator]
        out.println("Underlying list: " + strList); // [Naan, Bob, Rotator, Bob]

        out.println("\n(11) Membership test:");
```

```
    boolean found = strList.contains("Naan");
    String msg = found ? "contains" : "does not contain";
    out.println("The list " + msg + " the string \"Naan\".");

    out.println("\n(12) Find the index of an element:");
    int pos = strList.indexOf("Bob");
    out.println("The index of string \"Bob\" is: " + pos);
    pos = strList.indexOf("BOB");
    out.println("The index of string \"BOB\" is: " + pos);
    pos = strList.lastIndexOf("Bob");
    out.println("The last index of string \"Bob\" is: " + pos);
    printListWithIndex(strList);

    out.println("\n(13) Iterating over the list using the for(;;) loop:");
    for (int i = 0; i < strList.size(); i++) {
      out.print(i + ":" + strList.get(i) + " ");
    }
    out.println();

    out.println("\n(14) Iterating over the list using the for(:) loop:");
    for (String str : strList) {
      out.print(str +  " ");
      // strList.remove(str);        // Throws ConcurrentModificationException.
    }
    out.println();

    out.println("\n(15) Convert list to array:");
    Object[] objArray = strList.toArray();
    out.println("Object[] length: " + objArray.length);
    out.print("Length of each string in the Object array: ");
    for (Object obj : objArray) {
      String str = (String) obj; // Cast required.
      out.print(str.length() + " ");
    }
    out.println();
    String[] strArray = strList.toArray(new String[0]);
    out.println("String[] length: " + strArray.length);
    out.print("Length of each string in the String array: ");
    for (String str : strArray) {
      out.print(str.length() + " ");
    }
  }

  /**
   * Print the elements of a list, together with their index:
   * [0:value0, 1:value1, ...]
   * @param list   List to print with index
   */
  public static <E> void printListWithIndex(List<E> list) {             // (16)
```

```
    List<String> newList = new ArrayList<>();
    for (int i = 0; i < list.size(); i++) {
      newList.add(i + ":" + list.get(i));
    }
    out.println(newList);
  }
}
```

Output from the program:

Click here to view code image

```
(1) Create an empty list of strings:
[]

(2) Add elements to list:
[0:level]
[0:level, 1:Ada]
[0:level, 1:Ada, 2:kayak]
[0:level, 1:Ada, 2:kayak, 3:Bob]
[0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator]
[0:level, 1:Ada, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]
Insert an element at index 2 in the list:
[0:level, 1:Ada, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]

(3) Replace the element at index 1:
Element that was replaced: Ada
[0:level, 1:Naan, 2:Java, 3:kayak, 4:Bob, 5:Rotator, 6:Bob]

(4) Remove the element at index 0:
Element removed: level
[0:Naan, 1:Java, 2:kayak, 3:Bob, 4:Rotator, 5:Bob]

(5) Remove the first occurrence of "Java":
Element removed: true
[0:Naan, 1:kayak, 2:Bob, 3:Rotator, 4:Bob]

(6) Determine the size of the list:
The size of the list is 5

(7) Determine if the list is empty:
The list is not empty.

(8) Get the element at specific index:
First element: Naan
Last element: Bob

(9) Compare two lists:
```

```
The lists strList and strList2 are equal.
[0:Naan, 1:kayak, 2:Bob, 3:Rotator, 4:Bob, 5:null]
The lists strList and strList2 are not equal.

(10) Sublists as views:
Underlying list: [Naan, kayak, Bob, Rotator, Bob]
Sublist before remove: [kayak, Bob, Rotator]
Remove: kayak
Sublist after remove: [Bob, Rotator]
Underlying list: [Naan, Bob, Rotator, Bob]

(11) Membership test:
The list contains the string "Naan".

(12) Find the index of an element:
The index of string "Bob" is: 1
The index of string "BOB" is: -1
The last index of string "Bob" is: 3
[0:Naan, 1:Bob, 2:Rotator, 3:Bob]

(13) Iterating over the list using the for(;;) loop:
0:Naan 1:Bob 2:Rotator 3:Bob

(14) Iterating over the list using the for(:) loop:
Naan Bob Rotator Bob

(15) Convert list to array:
Object[] length: 4
Length of each string in the Object array: 4 3 7 3
String[] length: 4
Length of each string in the String array: 4 3 7 3
```

Review Questions

12.1 Which statement is true about the following program?

Click here to view code image

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

public class RQ12A10 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada"); strList.add("Ada");
    strList.add("Bob"); strList.add("Bob"); strList.add("Adda");
```

```
    for (int i = 0; i < strList.size(); /* empty */) {
      if (strList.get(i).length() <= 3) {
        strList.remove(i);
      } else {
        ++i;
      }
    }
    System.out.println(strList);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will throw an `IndexOutOfBoundsException` at runtime.

c. The program will throw a `ConcurrentModificationException` at runtime.

d. The program will not terminate when run.

e. The program will print `[Anna, Adda]`.

f. The program will print `[Anna, Ada, Bob, Adda]`.

12.2 Which of the following statements are true about the following program?

Click here to view code image

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

public class RQ12A15 {
  public static void main(String[] args) {
    doIt1(); doIt2();
  }

  public static void doIt1() {
    List<StringBuilder> sbListOne = new ArrayList<>();
    sbListOne.add(new StringBuilder("Anna"));
    sbListOne.add(new StringBuilder("Ada"));
    sbListOne.add(new StringBuilder("Bob"));
    List<StringBuilder> sbListTwo = new ArrayList<>(sbListOne);
    sbListOne.add(null);
    sbListTwo.get(1).reverse();
    System.out.println(sbListOne);                         // (1)
  }
```

```
  public static void doIt2() {
    List<String> listOne = new ArrayList<>();
    listOne.add("Anna"); listOne.add("Ada"); listOne.add("Bob");
    List<String> listTwo = new ArrayList<>(listOne);
    String strTemp = listOne.get(0);
    listOne.set(0, listOne.get(listOne.size()-1));
    listOne.set(listOne.size()-1, strTemp);
    System.out.println(listTwo);                                  // (2)
  }
}
```

Select the two correct answers.

a. (1) will print `[Anna, Ada, Bob, null]`.

b. (1) will print `[Anna, adA, Bob, null]`.

c. (2) will print `[Anna, Ada, Bob]`.

d. (2) will print `[Bob, Ada, Anna]`.

e. The program will throw an `IndexOutOfBoundsException` at runtime.

12.3 Which statement is true about the following program?

Click here to view code image

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

public class RQ12A20 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada"); strList.add(null);
    strList.add("Bob"); strList.add("Bob"); strList.add("Adda");
    for (int i = 0; i < strList.size(); ++i) {
      if (strList.get(i).equals("Bob")) {
        System.out.print(i);
      }
    }
    System.out.println();
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will throw an `IndexOutOfBoundsException` at runtime.

c. The program will throw a `NullPointerException` at runtime.

d. The program will print `34`.

12.4 Which statement is true about the following program?

Click here to view code image

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

public class RQ12A30 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada");
    strList.add("Bob"); strList.add("Bob");
    for (int i = 0; i < strList.size(); ++i) {
      if (strList.get(i).equals("Bob")) {
        strList.remove(i);
      }
    }
    System.out.println(strList);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will throw an `IndexOutOfBoundsException` at runtime.

c. The program will throw a `NullPointerException` at runtime.

d. The program will throw a `ConcurrentModificationException` at runtime.

e. The program will not terminate when run.

f. The program will print `[Anna, Ada, Bob]`.

g. The program will print `[Anna, Ada]`.

12.5 Which statement is true about the following program?

Click here to view code image

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

public class RQ12A40 {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<>();
    strList.add("Anna"); strList.add("Ada"); strList.add(null);
    strList.add("Bob"); strList.add("Bob"); strList.add("Adda");
    while (strList.remove("Bob"));
    System.out.println(strList);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will throw a `NullPointerException` at runtime.

c. The program will not terminate when run.

d. The program will print `[Anna, Ada, Adda]`.

e. The program will print `[Anna, Ada, Bob, Adda]`.

f. The program will print `[Anna, Ada, null, Adda]`.

g. The program will print `[Anna, Ada, null, Bob, Adda]`.

12.6 What will be the result of running the following program?

Click here to view code image

```
import java.util.*;
public class Test12_01 {
  public static void main(String[] args) {

    String[] data1 = {"A","B","B","A"};
    List<String> data2 = new ArrayList<>();
    for (String s : data1) {
      data2.add(s);
    }
    data2.set(1, "X");
    data2.add(1, "X");
    data2.remove(2);
```

```
    System.out.println(data2);
  }
}
```

Select the one correct answer.

a. `[X, B, B, A]`

b. `[A, X, B, A]`

c. `[A, X, A]`

d. `[A, X, X]`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

12.7 What will be the result of running the following program?

Click here to view code image

```
import java.util.*;
public class Test12_02 {
  public static void main(String[] args) {
    String[] data1 = {"A","B","B","A"};
    List<String> data2 = Arrays.asList(data1);
    data2.set(1, "X");
    data2.set(2, "X");
    System.out.println(data2);
  }
}
```

Select the one correct answer.

a. `[A, X, X, A]`

b. `[X, X, B, A]`

c. `[A, X, X, B, B, A]`

d. `[X, X, A, B, B, A]`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

12.8 What will be the result of running the following program?

Click here to view code image

```
public class Song {
  private String name;
  public Song(String name) {
    this.name = name;
  }
  public void update() {
    name = name.toUpperCase();
  }
  public String toString() {
    return name;
  }
}
```

Click here to view code image

```
import java.util.*;
public class Test12_04 {
  public static void main(String[] args) {
    Song[] playArray1 = {new Song("a"), new Song("b")};
    List<Song> playlist = Arrays.asList(playArray1);
    Song[] playArray2 = playlist.toArray(new Song[]{});
    playArray1[1].update();
    System.out.print(playArray1[1]);
    System.out.print(playlist.get(1));
    System.out.print(playArray2[1]);
  }
}
```

Select the one correct answer.

a. `Bbb`

b. `BBb`

c. `BBB`

d. `bbb`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

12.9 What will be the result of running the following program?

Click here to view code image

```
public class MySong {
  private String name;
  public MySong(String name) {
    this.name = name;
  }
  public String toString() {
    return name;
  }
}
```

Click here to view code image

```
import java.util.*;
public class Test12_05 {
  public static void main(String[] args) {
    MySong[] playArray1 = {new MySong("A"), new MySong("B")};
    List<MySong> playlist = List.of(playArray1);
    MySong[] playlist2 = playlist.toArray(new MySong[]{});
    playArray1[0] = new MySong("C");
    System.out.print(playArray1[0]);
    System.out.print(playlist.get(0));
    System.out.print(playlist2[0]);
  }
}
```

Select the one correct answer.

a. `CCA`

b. `CAA`

c. `CCC`

d. `AAA`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

12.10 What will be the result of running the following program?

Click here to view code image

```
import java.util.*;
public class Test12_06 {
  public static void main(String[] args) {
    List<String> data1 = List.of("A","B","C");
    String[] data2 = data1.toArray(new String[]{"X","Y","Z"});
    data2[1] = data1.get(0).toLowerCase();
    for (String s: data2) {
      System.out.print(s);
    }
  }
}
```

Select the one correct answer.

a. `AaC`

b. `AbC`

c. `XaZ`

d. `XbZ`

e. The program will throw an exception at runtime.

f. The program will fail to compile.

12.11 What will be the result of running the following program?

Click here to view code image

```
import java.util.*;
public class Test12_07 {
  public static void main(String[] args) {

    List<Character> text = new ArrayList<>(3);
    for (char a = 'a'; a <= 'e'; a++) {
      text.add(a);
    }
    System.out.println(text);
  }
}
```

Select the one correct answer.

a. `[a, b, c]`

b. `[a, b, c, d, e]`

c. `[a, b, c, d]`

d. The program will throw an exception at runtime.

e. The program will fail to compile.