Chapter 9
Collections and Generics

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ **Morking with Arrays and Collections**

- Create arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements.

In this chapter, we introduce the Java Collections Framework classes and interfaces you need to know for the exam. The thread-safe collections are discussed in Chapter 13, "Concurrency."

As you may remember from Chapter 8, "Lambdas and Functional Interfaces," we covered lambdas, method references, and built-in functional interfaces. Many of these are used throughout this chapter.

Next, we cover details about `Comparator`, `Comparable`, and sorting. We also introduce the new sequenced collections interfaces. Finally, we discuss how to create your own classes and methods that use generics so that the same class can be used with many types.

Using Common Collection APIs

A *collection* is a group of objects contained in a single object. The *Java Collections Framework* is a set of classes in `java.util` for storing collections. There are four main interfaces in the Java Collections Framework.

- **`List`:** A *list* is an ordered collection of elements that allows duplicate entries. Elements in a list can be accessed by an `int` index.
- **`Set`:** A *set* is a collection that does not allow duplicate entries.
- **`Queue`:** A *queue* is a collection that orders its elements in a specific order for processing. A `Deque` is a subinterface of `Queue` that allows access at both ends.
- **`Map`:** A *map* is a collection that maps keys to values, with no duplicate keys allowed. The elements in a map are key/value pairs.

Figure 9.1 shows the `Collection` interface, its subinterfaces, and some classes that implement the interfaces that you should know for the exam. The interfaces are shown in rectangles, with the classes in rounded boxes.

Notice that `Map` doesn't implement the `Collection` interface. It is considered part of the Java Collections Framework even though it isn't technically a `Collection`. It is a collection (note the lowercase), though, in that it contains a group of objects. The reason maps are treated differently is that they need different methods due to being key/value pairs.

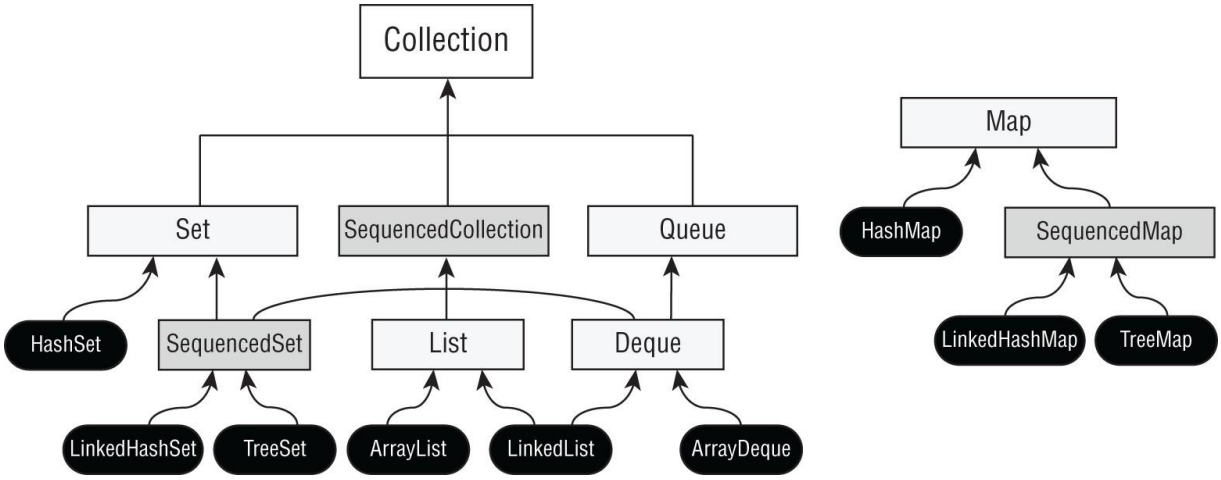


FIGURE 9.1 Java Collections Framework

Notice anything new in Figure 9.1? Brand new to Java 21 are sequenced collections! We'll discuss `SequencedSet`, `SequencedCollection`, and `SequencedMap` later in this chapter.

In this section, we discuss the common methods that the Collections API provides to the implementing classes. Many of these methods are *convenience methods* that could be implemented in other ways but make your code easier to write and read. This is why they are convenient.

In this section, we use `ArrayList` and `HashSet` as our implementation classes, but they can apply to any class that inherits the `Collection` interface. We cover the specific properties of each `Collection` class in the next section.

Understanding Generic Types

In the previous chapter, we showed you numerous functional interfaces that use generics. But what are generics? In Java, *generics* is just a way of saying parameterized type. For example, a `List<Integer>` is a list of numbers, while `Set<String>` is a set of strings.

Without generics, we'd have to write a lot of code like the following:

```
List numbers = new ArrayList(List.of(1,2,3));
Integer element = (Integer)numbers.get(0);  // Required cast to
compile
numbers.add("Welcome to the zoo!");         // Unrelated types allowed
```

With generics we can do better. The following change not only does away with the required cast from the previous code but also helps prevent unrelated objects from being added to the collection:

```
List<Integer> numbers = new ArrayList<Integer>(List.of(1,2,3));
Integer element = numbers.get(0);    // No cast required
numbers.add("Welcome to the zoo!");  // DOES NOT COMPILE
```

Getting a compiler error is good. You'll know right away that something is wrong rather than hoping to discover it later. Generics are convenient because the code for `List`, `Set`, and other collections does not change based on the generic type. You can even use your own class as the type, such as `List<Visitor>`.

We'll be using generics throughout this chapter, and even show you how to define your own generic classes toward the end of this chapter.

Shortening Generics Code

In the previous section, you saw generics that declare the type on both the left and right sides like the following:

```
List<Integer> list = new ArrayList<Integer>();
```

You might even have generics that contain other generics, such as this:

```
Map<Long,List<Integer>> mapOfLists = new HashMap<Long,List<Integer>>
();
```

That's a lot of duplicate code to write! In this section, we offer two ways of shortening this code.

Applying the Diamond Operator

The *diamond operator* (`<>`) is a shorthand notation that allows you to omit the generic type from the right side of a statement when the type can be inferred. It is

called the diamond operator because <> looks like a diamond. Compare the previous declarations with these new, much shorter versions:

```
List<Integer> list = new ArrayList<>();
Map<Long,List<Integer>> mapOfLists = new HashMap<>();
```

To the compiler, both these declarations and our previous ones are equivalent. Note the diamond operator cannot be used as the type in a variable declaration. It can be used only on the right side of an assignment operation. For example, neither of the following compiles:

```
List<> list = new ArrayList<Integer>();      // DOES NOT COMPILE

class InvalidUse {
   void use(List<> data) {}                  // DOES NOT COMPILE
}
```

Applying *var*

As you've seen earlier in this book, you can also use `var` to shorten expressions with generics.

```
var list = new ArrayList<Integer>();
var mapOfLists = new HashMap<Long,List<Integer>>();
```

Notice how the generic type is back on the right side. That's because `var` infers the type from the right side of the declaration, whereas the diamond operator infers it from the left side.

Which style you use, `var` or diamond operator, is up to you. Just don't specify the generic type on both sides of the `=`, because that's redundant!

Using Both Shorteners

What happens if you use both `var` and the diamond operator?

```
var map = new HashMap<>();
```

Believe it or not, this does compile! If you try to have them both infer, there isn't enough information and you get `Object` as the generic type. This is equivalent to the following:

```
HashMap<Object, Object> map = new HashMap<Object, Object>();
```

Adding Data

The `add()` method inserts a new element into the `Collection` and returns whether it was successful. The method signature is as follows:

```
public boolean add(E element)
```

Remember that the Collections Framework uses generics. You will see `E` appear frequently. It means the generic type that was used to create the collection. For some `Collection` types, `add()` always returns `true`. For other types, there is logic as to whether the `add()` call was successful. The following shows how to use this method:

```
3: Collection<String> list = new ArrayList<>();
4:  System.out.println(list.add("Sparrow"));  //  true
5:  System.out.println(list.add("Sparrow"));  //  true
6:
7: Collection<String> set = new HashSet<>();
8: System.out.println(set.add("Sparrow")); // true
9:  System.out.println(set.add("Sparrow"));  //  false
```

A `List` allows duplicates, making the return value `true` each time. A `Set` does not allow duplicates. On line 9, we tried to add a duplicate so that Java returns `false` from the `add()` method.

Removing Data

The `remove()` method removes a single matching value in the `Collection` and returns whether it was successful. The method signature is as follows:

```
public boolean remove(Object object)
```

This time, the `boolean` return value tells us whether a match was removed. The following shows how to use this method:

```
3: Collection<String> birds = new ArrayList<>();
4: birds.add("hawk");                              // [hawk]
5: birds.add("hawk");                              // [hawk, hawk]
6: System.out.println(birds.remove("cardinal")); // false
7: System.out.println(birds.remove("hawk"));     // true
8: System.out.println(birds);                     // [hawk]
```

Line 6 tries to remove an element that is not in `birds`. It returns `false` because no such element is found. Line 7 tries to remove an element that is in `birds`, so it returns `true`. Notice that it removes only one match.

Counting Elements

The `isEmpty()` and `size()` methods look at how many elements are in the `Collection`. The method signatures are as follows:

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

The following shows how to use these methods:

```
Collection<String> birds = new ArrayList<>();
System.out.println(birds.isEmpty()); // true
System.out.println(birds.size());    // 0
birds.add("hawk");                   // [hawk]
birds.add("hawk");                   // [hawk, hawk]
System.out.println(birds.isEmpty()); // false
System.out.println(birds.size());    // 2
```

At the beginning, `birds` has a size of `0` and is empty. It has a capacity that is greater than `0`. After we add elements, the size becomes positive, and it is no longer empty.

Clearing the Collection

The `clear()` method provides an easy way to discard all elements of the `Collection`. The method signature is as follows:

```
public void clear()
```

The following shows how to use this method:

```
Collection<String> birds = new ArrayList<>();
birds.add("hawk");                   // [hawk]
birds.add("hawk");                   // [hawk, hawk]
System.out.println(birds.isEmpty()); // false
System.out.println(birds.size());    // 2
birds.clear();                       // []
System.out.println(birds.isEmpty()); // true
System.out.println(birds.size());    // 0
```

After calling `clear()`, `birds` is back to being an empty `ArrayList` of size `0`.

Checking Contents

The `contains()` method checks whether a certain value is in the `Collection`. The method signature is as follows:

```
public boolean contains(Object object)
```

The following shows how to use this method:

```
Collection<String> birds = new ArrayList<>();
birds.add("hawk");                           // [hawk]
```

```
System.out.println(birds.contains("hawk"));  // true
System.out.println(birds.contains("robin")); // false
```

The `contains()` method calls `equals()` on elements of the `ArrayList` to see whether there are any matches.

Removing with Conditions

The `removeIf()` method removes all elements that match a condition. We can specify what should be deleted using a block of code or even a method reference.

The method signature looks like the following. (We explain what the `? super` means in the "Working with Generics" section later in this chapter.)

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

It uses a `Predicate`, which takes one parameter and returns a `boolean`. Let's take a look at an example:

```
4: Collection<String> list = new ArrayList<>();
5: list.add("Magician");
6: list.add("Assistant");
7: System.out.println(list);     // [Magician, Assistant]
8: list.removeIf(s -> s.startsWith("A"));
9: System.out.println(list);     // [Magician]
```

Line 8 shows how to remove all of the `String` values that begin with the letter `A`. This allows us to make the `Assistant` disappear. Let's try an example with a method reference:

```
11: Collection<String> set = new HashSet<>();
12: set.add("Wand");
13: set.add("");
14: set.removeIf(String::isEmpty); // s -> s.isEmpty()
15: System.out.println(set);       // [Wand]
```

On line 14, we remove any empty `String` objects from `set`. The comment on that line shows the lambda equivalent of the method reference. Line 15 shows that the `removeIf()` method successfully removed one element from `list`.

Iterating on a Collection

There's a `forEach()` method that you can call on a `Collection` instead of writing a loop. It uses a `Consumer` that takes a single parameter and doesn't return anything. The method signature is as follows:

```
public void forEach(Consumer<? super T> action)
```

Cats like to explore, so let's print out two of them using both method references and lambdas:

```
Collection<String> cats = List.of("Annie", "Ripley");
cats.forEach(System.out::println);
cats.forEach(c -> System.out.println(c));
```

The cats have discovered how to print their names. Now they have more time to play (as do we)!

Other Iteration Approaches

There are other ways to iterate through a `Collection`. For example, in Chapter 3, "Making Decisions," you saw how to loop through a list using an enhanced `for` loop.

```
for (String element: coll)
   System.out.println(element);
```

You may see another older approach used.

```
Iterator<String> iter = coll.iterator();
while(iter.hasNext()) {
   String name = iter.next();
   System.out.println(name);
}
```

Pay attention to the difference between these techniques. The `hasNext()` method checks whether there is a next value. In other words, it tells you whether `next()` will execute without throwing an exception. The `next()` method actually moves the `Iterator` to the next element.

Determining Equality

There is a custom implementation of `equals()` so you can compare two `Collections` to compare the type and contents. The implementation will vary. For example, `ArrayList` checks order, while `HashSet` does not.

```
boolean equals(Object object)
```

The following shows an example:

```
23: var list1 = List.of(1, 2);
24: var list2 = List.of(2, 1);
```

```
25: var set1 = Set.of(1, 2);
26: var set2 = Set.of(2, 1);
27:
28: System.out.println(list1.equals(list2)); // false
29: System.out.println(set1.equals(set2));   // true
30: System.out.println(list1.equals(set1));  // false
```

Line 28 prints `false` because the elements are in a different order, and a `List` cares about order. By contrast, line 29 prints `true` because a `Set` is not sensitive to order. Finally, line 30 prints `false` because the types are different.

Unboxing *nulls*

Java protects us from many problems with `Collections`. However, it is still possible to write a `NullPointerException`.

```
3: var heights = new ArrayList<Integer>();
4: heights.add(null);
5: int h = heights.get(0);  // NullPointerException
```

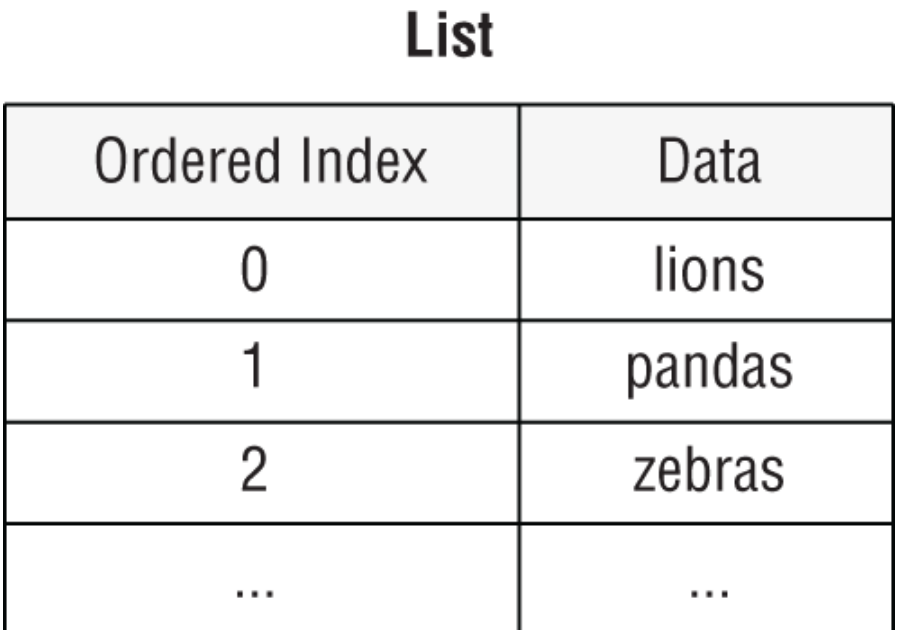
On line 4, we add a `null` to the list. This is legal because a `null` reference can be assigned to any reference variable. On line 5, we try to unbox that `null` to an `int` primitive. This is a problem. Java tries to get the `int` value of `null`. Since calling any method on `null` gives a `NullPointerException`, that is just what we get. Be careful when you see `null` in relation to autoboxing.

Using the *List* Interface

Now that you're familiar with some common `Collection` interface methods, let's move on to specific interfaces. You use a list when you want an ordered collection that can contain duplicate entries. For example, a list of names may contain duplicates, as two animals can have the same name. Items can be retrieved and inserted at specific positions in the list based on an `int` index, much like an array. Unlike an array, though, many `List` implementations can change in size after they are declared.

Lists are commonly used because there are many situations in programming where you need to keep track of a list of objects. For example, you might make a list of what you want to see at the zoo: first, see the lions, because they go to sleep early; second, see the pandas, because there is a long line later in the day; and so forth.

Figure 9.2 shows how you can envision a `List`. Each element of the `List` has an index, and the indexes begin with zero.



List

Ordered Index	Data
0	lions
1	pandas
2	zebras
...	...

FIGURE 9.2 Example of a `List`

Sometimes you don't care about the order of elements in a list. `List` is like the "go to" data type. When we make a shopping list before going to the store, the order of the list happens to be the order in which we thought of the items. We probably aren't attached to that particular order, but it isn't hurting anything.

While the classes implementing the `List` interface have many methods, you need to know only the most common ones. Conveniently, these methods are the same for all of the implementations that might show up on the exam.

The main thing all `List` implementations have in common is that they are ordered and allow duplicates. Beyond that, they each offer different functionality. We look at the implementations that you need to know and the available methods.

Pay special attention to which names are classes and which are interfaces. The exam may ask you which is the best class or which is the best interface for a scenario.

Comparing List Implementations

Reviewing Figure 9.1, you need to know about two classes that implement the `List` interface: `ArrayList` and `LinkedList`. An `ArrayList` is like a resizable array. When elements are added, the `ArrayList` automatically grows. When you aren't sure which collection to use, use an `ArrayList`.

The main benefit of an `ArrayList` is that you can look up any element in constant time. Adding or removing an element is slower than accessing an element. This makes an `ArrayList` a good choice when you are reading more often than (or the same amount as) writing to the `ArrayList`.

A `LinkedList` is special because it implements both `List` and `Deque`. It has all the methods of a `List`. It also has additional methods to facilitate adding or removing from the beginning and/or end of the list.

The main benefits of a `LinkedList` are that you can access, add to, and remove from the beginning and end of the list in constant time. The trade-off is that dealing with an arbitrary index takes linear time. This makes a `LinkedList` a good choice when you'll be using it as `Deque`.

Creating a *List* with a Factory

When you create a `List` of type `ArrayList` or `LinkedList`, you know the type. There are a few special methods where you get a `List` back but don't know the type. These methods let you create a `List` including data in one line using a factory method. This is convenient, especially when testing. Some of these methods return an immutable object. As we saw in Chapter 6, "Class Design," an immutable object cannot be changed or modified. Table 9.1 summarizes these three methods to create a list.

TABLE 9.1 Factory methods to create a `List`

Method	Description	Can add elements?	Can replace elements?	Can delete elements?
`Arrays.asList(varargs)`	Returns fixed size list backed by an array	No	Yes	No
`List.of(varargs)`	Returns immutable list	No	No	No
`List.copyOf(collection)`	Returns immutable list with copy of original collection's values	No	No	No

Let's take a look at an example of these three methods:

```
16: String[] array = new String[] {"a", "b", "c"};
17: List<String> asList = Arrays.asList(array); // [a, b, c]
18: List<String> of = List.of(array);           // [a, b, c]
19: List<String> copy = List.copyOf(asList);    // [a, b, c]
20:
21: array[0] = "z";
22:
23: System.out.println(asList);                 // [z, b, c]
24: System.out.println(of);                     // [a, b, c]
25: System.out.println(copy);                   // [a, b, c]
26:
27: asList.set(0, "x");
28: System.out.println(Arrays.toString(array)); // [x, b, c]
```

Line 17 creates a `List` that is backed by an array. Line 21 changes the array, and line 23 reflects that change. Lines 27 and 28 show the other direction where changing the `List` updates the underlying array. Lines 18 and 19 each create an immutable `List`.

When run independently, the following shows both types are immutable by throwing an exception when trying to set a value.

```
of.set(0, "y");     // UnsupportedOperationException
copy.set(0, "y");   // UnsupportedOperationException
```

Similarly, each of the following lines throws an exception when adding or removing a value:

```
asList.add("z");    // UnsupportedOperationException
of.remove(0);       // UnsupportedOperationException
```

```
copy.remove(0);     // UnsupportedOperationException
```

Creating a *List* with a Constructor

Most `Collections` have two constructors that you need to know for the exam. The following shows them for `LinkedList`:

```
var linked1 = new LinkedList<String>();
var linked2 = new LinkedList<String>(linked1);
```

The first says to create an empty `LinkedList` containing all the defaults. The second tells Java that we want to make a copy of another `LinkedList`. Granted, `linked1` is empty in this example, so it isn't particularly interesting.

`ArrayList` has an extra constructor you need to know. We now show the three constructors.

```
var list1 = new ArrayList<String>();
var list2 = new ArrayList<String>(list1);
var list3 = new ArrayList<String>(10);
```

The first two are the common constructors you need to know for all `Collection`s. The final example says to create an `ArrayList` containing a specific number of slots, but again not to assign any. You can think of this as the size of the underlying array.

Working with *List* Methods

The methods in the `List` interface are for working with indexes. In addition to the inherited `Collection` methods, you should also know the methods in Table 9.2 for the exam.

TABLE 9.2 `List` methods

Method	Description
`boolean` **`add`**`(E element)`	Adds element to end (available on all `Collection` APIs).
`void` **`add`**`(int index, E element)`	Adds element at index and moves the rest toward the end.
`E` **`get`**`(int index)`	Returns element at index.
`int` **`indexOf`**`(Object o)`	Returns the index of the first matching element or -1 if not found.
`int` **`lastIndexOf`**`(Object o)`	Returns the index of the last matching element or -1 if not found.
`E` **`remove`**`(int index)`	Removes element at index and moves the rest toward the front.
`default void` **`replaceAll`**`(UnaryOperator<E> op)`	Replaces each element in list with the result of operator.
`E` **`set`**`(int index, E e)`	Replaces element at index and returns original. Throws `IndexOutOfBoundsException` if index is invalid.
`default void` **`sort`**`(Comparator<? super E> c)`	Sorts list. We cover this later in the chapter in the "Sorting Data" section.

The following statements demonstrate most of these methods for working with a `List`:

```
3:  List<String> list = new ArrayList<>();
4:  list.add("SD");                    // [SD]
5:   list.add(0, "NY");                // [NY,SD]
6:   list.set(1, "FL");                // [NY,FL]
7:   System.out.println(list.get(0)); // NY
8:   list.remove("NY");                // [FL]
9:   list.remove(0);                   // []
10: list.set(0, "?");                  // IndexOutOfBoundsException
```

On line 3, `list` starts out empty. Line 4 adds an element to the end of the list. Line 5 adds an element at index `0` that bumps the original index `0` to index `1`. Notice how

the `ArrayList` is now automatically one larger. Line 6 replaces the element at index `1` with a new value.

Line 7 uses the `get()` method to print the element at a specific index. Line 8 removes the element matching `NY`. Finally, line 9 removes the element at index `0`, and `list` is empty again.

Line 10 throws an `IndexOutOfBoundsException` because there are no elements in the `List`. Since there are no elements to replace, even index `0` isn't allowed. If line 10 were moved up between lines 4 and 5, the call would succeed.

The output would be the same if you tried these examples with `LinkedList`. Although the code would be less efficient, it wouldn't be noticeable until you had very large lists.

Now let's take a look at the `replaceAll()` method. It uses a `UnaryOperator` that takes one parameter and returns a value of the same type.

```
var numbers = Arrays.asList(1, 2, 3);
numbers.replaceAll(x -> x*2);
System.out.println(numbers);   // [2, 4, 6]
```

This lambda doubles the value of each element in the list. The `replaceAll()` method calls the lambda on each element of the list and replaces the value at that index.

Overloaded *remove()* Methods

We've now seen two overloaded `remove()` methods. The one from `Collection` removes an object that matches the parameter. By contrast, the one from `List` removes an element at a specified index.

This gets tricky when you have an `Integer` type. What do you think the following prints?

```
31: var list = new LinkedList<Integer>();
32: list.add(3);
33: list.add(2);
34: list.add(1);
35: list.remove(2);
36: list.remove(Integer.valueOf(2));
37: System.out.println(list);
```

The correct answer is `[3]`. Let's look at how we got there. At the end of line 34, we have `[3, 2, 1]`. Line 35 passes a primitive, which means we are requesting deletion of the element at index `2`. This leaves us with `[3, 2]`. Then line 36 passes an `Integer` object, which means we are deleting the value 2. That brings us to `[3]`.

The `remove()` method that takes an element will return `false` if the element is not found. Contrast this with the `remove()` method that takes an `int`, which throws an exception if the element is not found:

```
var list = new LinkedList<Integer>();
list.remove(Integer.valueOf(100));  // Returns false
list.remove(100);                   // IndexOutOfBoundsException
```

Searching a List

From Table 9.2, the `List` interface includes two methods for searching for elements, `indexOf()` and `lastIndexOf()`. They work similarly to the methods of the same name in the `String` class:

```
var list = List.of("peacock", "chicken", "peacock", "turkey");

System.out.println(list.indexOf("peacock"));      // 0
System.out.println(list.lastIndexOf("peacock"));  // 2
System.out.println(list.indexOf("penguin"));      // -1
```

Later in this chapter, we'll show you how to perform a more efficient search by first sorting the list and then using the `Collections.binarySearch()` method.

Converting from *List* to an Array

Since an array can be passed as a vararg, Table 9.1 covered how to convert an array to a `List`. You should also know how to do the reverse. Let's start with turning a `List` into an array.

```
13: List<String> list = new ArrayList<>();
14: list.add("hawk");
15: list.add("robin");
16: Object[] objectArray = list.toArray();
17: String[] stringArray = list.toArray(new String[0]);
18: list.clear();
19: System.out.println(objectArray.length);     // 2
20: System.out.println(stringArray.length);     // 2
```

Line 16 shows that a `List` knows how to convert itself to an array. The only problem is that it defaults to an array of class `Object`. This isn't usually what you want. Line 17 specifies the type of the array and does what we want. The advantage of specifying a size of `0` for the parameter is that Java will create a new array of the proper size for the return value. If you like, you can suggest a larger array to be used instead. If the `List` fits in that array, it will be returned. Otherwise, a new array will be created.

Also, notice that line 18 clears the original `List`. This does not affect either array. The array is a newly created object with no relationship to the original `List`. It is simply a copy.

Using the *Set* Interface

You use a `Set` when you don't want to allow duplicate entries. For example, you might want to keep track of the unique animals that you want to see at the zoo. You aren't concerned with the order in which you see these animals, but there isn't time to see them more than once. You just want to make sure you see the ones that are important to you and remove them from the set of outstanding animals to see after you see them.

Figure 9.3 shows how you can envision a `Set`. The main thing that all `Set` implementations have in common is that they do not allow duplicates. We look at each implementation that you need to know for the exam and how to write code using `Set`.

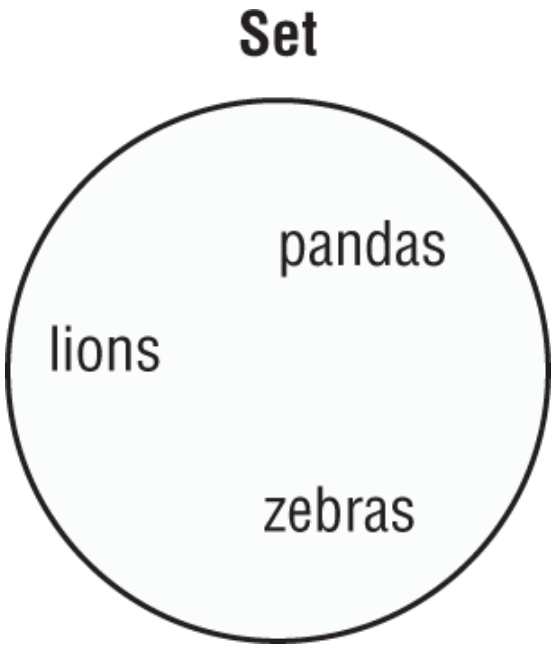


FIGURE 9.C Example of a `Set`

Comparing *Set* Implementations

Reviewing Figure 9.1 again, you need to know about three classes that implement the `Set` interface: `HashSet`, `LinkedHashSet`, `TreeSet`. A `HashSet` stores its elements in a *hash table*, which means the keys are a hash and the values are an `Object`. This means the `HashSet` uses the `hashCode()` method of the objects to retrieve them more efficiently. Remember that a valid `hashCode()` doesn't mean every object will get a unique value, but the method is often written so that hash values are spread out over a large range to reduce collisions.

A `LinkedHashSet` is basically a `HashSet` with an imaginary `LinkedList` running across its elements. This allows you to iterate over the set in a well-defined encounter order, which is often the order the elements were inserted. That said, `LinkedHashSet` also includes methods to add/remove elements from the front or back of the set, allowing you to change the ordering as needed.

Finally, a `TreeSet` stores its elements in a sorted tree structure. The main benefit is that the set is always in sorted order. The trade-off is that adding or removing an element could take longer than with a `HashSet`, especially as the tree grows larger.

Figure 9.4 shows how you can envision these three classes being stored. `HashSet` is more complicated in reality, but this is fine for the purpose of the exam.

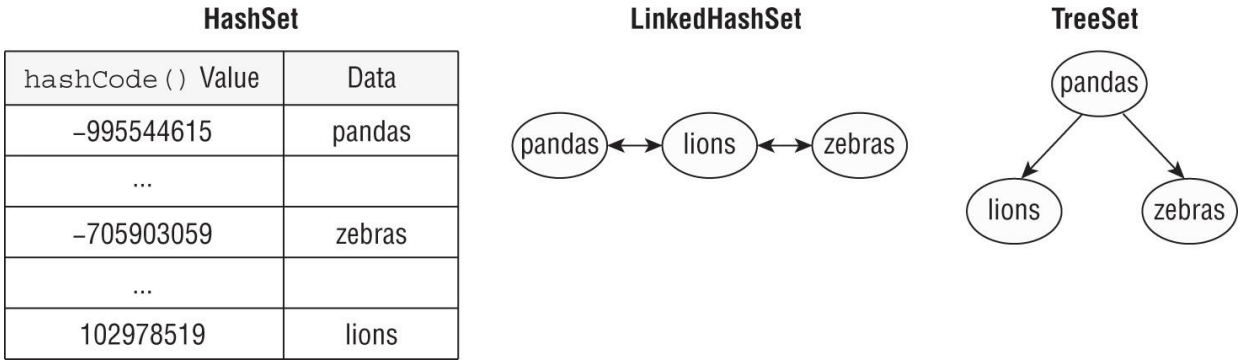


FIGURE 9.4 Examples of Sets

For the exam, you don't need to know how to create a hash or tree set class (the implementation can be complex). Phew! You just need to know how to use them!

Working with *Set* Methods

Like a `List`, you can create an immutable `Set` in one line or make a copy of an existing one.

```
Set<Character> letters = Set.of('c', 'a', 't');
Set<Character> copy = Set.copyOf(letters);
```

Those are the only extra methods you need to know for the `Set` interface for the exam! You do have to know how sets behave with respect to the traditional `Collection` methods. You also have to know the differences between the types of sets. Let's start with `HashSet`.

```
3: Set<Integer> set = new HashSet<>();
4: boolean b1 = set.add(66);  // true
5: boolean b2 = set.add(10);  // true
6: boolean b3 = set.add(66);  // false
7: boolean b4 = set.add(8);   // true
8: for (Integer value: set)
9:    System.out.print(value + ","); // 66,8,10,
```

The `add()` methods should be straightforward. They return `true` unless the `Integer` is already in the set. Line 6 returns `false`, because we already have 66 in the set, and a set must preserve uniqueness. Line 8 prints the elements of the set in an *arbitrary* order. In this case, it happens not to be sorted order or the order in which we added the elements.

Remember that the `equals()` method is used to determine equality. The `hashCode()` method is used to know which bucket to look in so that Java doesn't have to look through the whole set to find out whether an object is there. The best case is that hash codes are unique and Java has to call `equals()` on only one object. The worst

case is that all implementations return the same `hashCode()` and Java has to call `equals()` on every element of the set anyway.

Let's replace line 3 with a `LinkedHashSet` and see how the output changes.

```
3: Set<Integer> set = new LinkedHashSet<>();
```

This time, the code prints the elements in the order they were inserted.

```
66,10,8,
```

Finally, we can use a `TreeSet` on line 3.

```
3: Set<Integer> set = new TreeSet<>();
```

The elements are now printed out in their natural sorted order.

```
8,10,66,
```

Number wrapper types implement the `Comparable` interface in Java, which is used for sorting. Later in the chapter, you learn how to create your own `Comparable` objects.

Using the *Queue* and *Deque* Interfaces

You use a `Queue` when elements are added and removed in a specific order. You can think of a queue as a line. For example, when you want to enter a stadium and someone is waiting in line, you get in line behind that person. And if you are British, you get in the queue behind that person, making this really easy to remember! This is a *FIFO* (first-in, first-out) queue.

A `Deque` (double-ended queue), often pronounced "deck" extends `Queue` but is different from a regular queue in that you can insert and remove elements from both the front (head) and back (tail). Think, "Dr. Woodie Flowers, come right to the front! You are the only one who gets this special treatment. Everyone else will have to start at the back of the line."

You can envision a double-ended queue as shown in Figure 9.5.

FIGURE 9.5 Example of a `Deque`

Supposing we are using this as a FIFO queue. Rover is first, which means he was first to arrive. Bella is last, which means she was last to arrive and has the longest wait remaining. All queues have specific requirements for adding and removing the

next element. Beyond that, they each offer different functionality. We look at the implementations you need to know and the available methods.

Comparing *Deque* Implementations

Reviewing Figure 9.1 one more time (you should know it well by now!), `LinkedList` and `ArrayDeque` both implement the `Deque` interface, which inherits `Queue`. You saw `LinkedList` earlier in the `List` section. The main benefit of a `LinkedList` is that it implements both the `List` and `Deque` interfaces. The trade-off is that it isn't as efficient as a "pure" queue. You can use the `ArrayDeque` class if you don't need the `List` methods.

Working with *Queue* and *Deque* Methods

The `Queue` interface contains six methods, shown in Table 9.3. There are three capabilities, each with two versions of the methods: one that throws an exception, and one that uses the return type to convey the same information. We've bolded the ones that throw an exception when something goes wrong, like trying to read from an empty `Queue`.

TABLE 9.C `Queue` methods

Functionality	Methods
Add to back	**`boolean add(E e)`** `boolean offer(E e)`
Read from front	**`E element()`** `E peek()`
Get and remove from front	**`E remove()`** `E poll()`

Let's look at the following simple queue example:

```
4: Queue<Integer> queue = new LinkedList<>();
5: queue.add(10);
6: queue.add(4);
7: System.out.println(queue.remove());   // 10
8: System.out.println(queue.peek());     // 4
```

Lines 5 and 6 add elements to the queue. Line 7 asks the first element waiting the longest to come off the queue. Line 8 checks for the next entry in the queue while leaving it in place.

Next, we move on to the `Deque` interface. Since the `Deque` interface supports double-ended queues, it inherits all `Queue` methods and adds more so that it is clear if we are

working with the front or back of the queue. Table 9.4 shows the methods when using it as a double-ended queue.

TABLE 9.4 `Deque` methods

Functionality	Methods
Add to front	**`void addFirst(E e)`** `boolean offerFirst(E e)`
Add to back	**`void addLast(E e)`** `public boolean offerLast(E e)`
Read from front	**`E getFirst()`** `E peekFirst()`
Read from back	**`E getLast()`** `E peekLast()`
Get and remove from front	**`E removeFirst()`** `E pollFirst()`
Get and remove from back	**`E removeLast()`** `E pollLast()`

Let's try an example that works with both ends of the queue:

```
Deque<Integer> deque = new LinkedList<>();
```

This is more complicated, so we use Figure 9.6 to show what the queue looks like at each step of the code.

Lines 13 and 14 successfully add an element to the front and back of the queue, respectively. Some queues are limited in size, which would cause offering an element to the queue to fail. You won't encounter a scenario like that on the exam. Line 15 looks at the first element in the queue, but it does not remove it. Lines 16 and 17 remove the elements from the queue, one from each end. This results in an empty queue. Lines 18 and 19 try to look at the first element of the queue, which results in `null`.

In addition to FIFO queues, there are *LIFO* (last-in, first-out) queues, which are commonly referred to as *stacks*. Picture a stack of plates. You always add to or remove from the top of the stack to avoid a mess. Luckily, we can use the same double-ended queue implementations. Different methods are used for clarity, as shown in Table 9.5.

```
13: deque.offerFirst(10); // true
14: deque.offerLast(4);   // true
15: deque.peekFirst();    // 10
16: deque.pollFirst();    // 10
17: deque.pollLast();     // 4
18: deque.pollFirst();    // null
19: deque.peekFirst();    // null
```

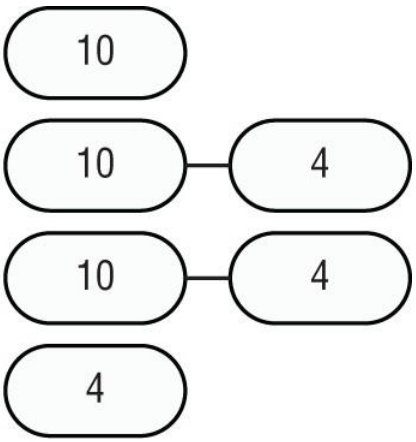


FIGURE 9.6 Working with a `Deque`

TABLE 9.5 Using a `Deque` as a stack

Functionality	Methods
Add to the front/top	`void push(E e)`
Remove from the front/top	`E pop()`
Get first element	`E peek()`

Let's try another one using the `Deque` as a stack:

```
Deque<Integer> stack = new ArrayDeque<>();
```

This time, Figure 9.7 shows what the stack looks like at each step of the code. Lines 13 and 14 successfully put an element on the front/top of the stack. The remaining code looks at the front as well.

```
13: stack.push(10);
14: stack.push(4);
15: stack.peek();     // 4
16: stack.pop();      // 4
17: stack.pop();      // 10
18: stack.peek();     // null
```

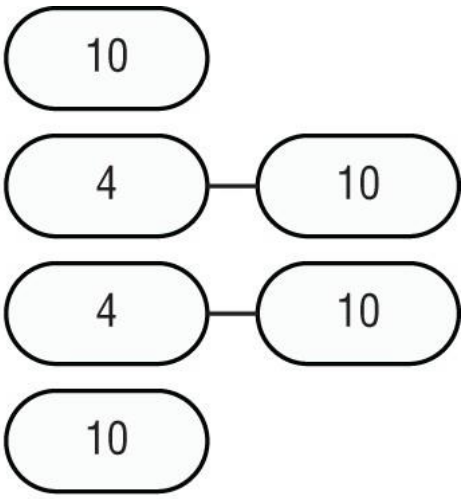


FIGURE 9.7 Working with a stack

When using a `Deque`, it is really important to determine if it is being used as a FIFO queue, a LIFO stack, or a double-ended queue. To review, a FIFO queue is like a line of people. You get on in the back and off in the front. A LIFO stack is like a stack of plates. You put the plate on the top and take it off the top. A double-ended queue uses both ends.

Using the *Map* Interface

You use a `Map` when you want to identify values by a key. For example, when you use the contact list in your phone, you look up "George" rather than looking through each phone number in turn.

You can envision a `Map` as shown in Figure 9.8. You don't need to know the names of the specific interfaces that the different maps implement, but you do need to know that `LinkedHashMap` is ordered and `TreeMap` is sorted.

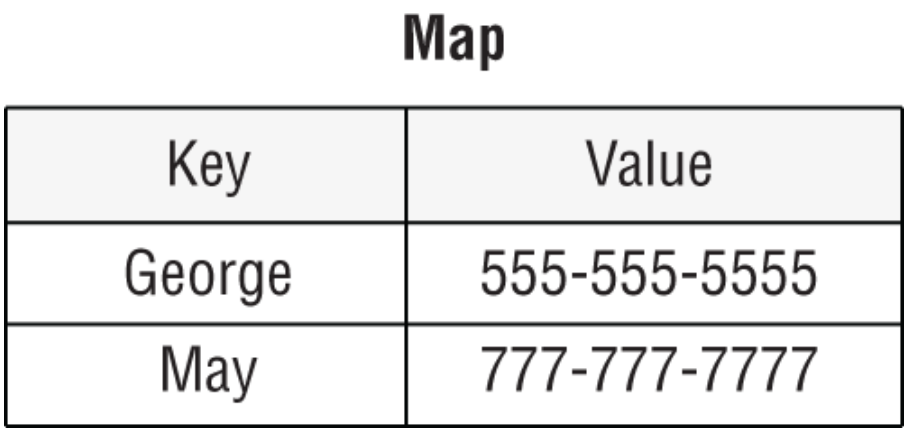


FIGURE 9.8 Example of a `Map`

The main thing that all `Map` classes have in common is that they have keys and values. Beyond that, they each offer different functionality. We look at the implementations you need to know and the available methods.

Map.of() and *Map.copyOf()*

Just like `List` and `Set`, there is a factory method to create a `Map`. You pass up to 10 pairs of keys and values.

```
Map.of("key1", "value1", "key2", "value2");
```

Unlike `List` and `Set`, this is less than ideal. Passing keys and values is harder to read because you have to keep track of which parameter is which. Luckily, there is a better way. `Map` also provides a method that lets you supply key/value pairs.

```
Map.ofEntries(
   Map.entry("key1", "value1"),
   Map.entry("key2",  "value2"));
```

Now we can't forget to pass a value. If we leave out a parameter, the `entry()` method won't compile. Conveniently, `Map.copyOf(map)` works just like the `List` and `Set` interface `copyOf()` methods.

Comparing *Map* Implementations

From Figure 9.1, `HashMap`, `LinkedHashMap`, and `TreeMap` are the three classes that implement the `Map` interface. A `HashMap` stores the keys in a hash table. This means

that it uses the `hashCode()` method of the keys to retrieve their values more efficiently.

Like `LinkedHashSet`, the `LinkedHashMap` supports iterating over the elements in a well-defined order. This is generally the insertion order, although it also includes methods to add/remove elements at the front or back of the map.

Finally, a `TreeMap` stores the keys in a sorted tree structure. The main benefit is that the keys are always in sorted order. Like a `TreeSet`, the trade-off is that adding and checking whether a key is present takes longer as the tree grows larger.

Working with *Map* Methods

Given that `Map` doesn't extend `Collection`, more methods are specified on the `Map` interface. Since there are both keys and values, we need generic type parameters for both. The class uses `K` for key and `V` for value. The methods you need to know for the exam are in Table 9.6. Some of the method signatures are simplified to make them easier to understand.

TABLE 9.6 `Map` methods

Method	Description
`void` **`clear`**`()`	Removes all keys and values from map.
`boolean` **`containsKey`**`(Object key)`	Returns whether key is in map.
`boolean` **`containsValue`**`(Object value)`	Returns whether value is in map.
`Set<Map.Entry<K,V>>` **`entrySet`**`()`	Returns `Set` of key/value pairs.
`void` **`forEach`**`(BiConsumer<K, V> action)`	Loops through each key/value pair.
`V` **`get`**`(Object key)`	Returns value mapped by key or `null` if none is mapped.
`V` **`getOrDefault`**`(Object key, V defaultValue)`	Returns value mapped by key or default value if none is mapped.
`boolean` **`isEmpty`**`()`	Returns whether map is empty.
`Set<K>` **`keySet`**`()`	Returns set of all keys.
`V` **`merge`**`(K key, V value, BiFunction<V, V, V> func)`	Sets value if key not set. Runs function if key is set, to determine new value. Removes if value is `null`.
`V` **`put`**`(K key, V value)`	Adds or replaces key/value pair. Returns previous value or `null`.
`V` **`putIfAbsent`**`(K key, V value)`	Adds value if key not present and returns null. Otherwise, returns existing value.
`V` **`remove`**`(Object key)`	Removes and returns value mapped to key. Returns `null` if none.
`V` **`replace`**`(K key, V value)`	Replaces value for given key if key is set. Returns original value or `null` if none.
`void` **`replaceAll`**`(BiFunction<K, V, V> func)`	Replaces each value with results of function.
`int` **`size`**`()`	Returns number of entries (key/value pairs) in map.
`Collection<V>` **`values`**`()`	Returns `Collection` of all values.

While Table 9.6 is a pretty long list of methods, don't worry; many of the names are straightforward. Also, many exist as a convenience. For example, `containsKey()` can be replaced with a `get()` call that checks if the result is `null`. Which one you use is up to you.

Calling Basic Methods

Let's start by comparing the behavior of each of the `Map` classes. Consider the following method:

```
void addElementsAndPrint(Map<String, String> map) {
   map.put("koala", "bamboo");
   map.put("lion", "meat");
   map.put("giraffe", "leaf");
   String food = map.get("koala"); // bamboo
   for (String key: map.keySet())
      System.out.print(key + ",");
}
```

Here we use the `put()` method to add key/value pairs to the map and `get()` to get a value given a key. We also use the `keySet()` method to get all the keys. We can then apply this method to each of our three `Map` classes.

```
addElementsAndPrint(new   HashMap<>());    //   koala,giraffe,lion,
addElementsAndPrint(new LinkedHashMap<>()); // koala,lion,giraffe,
addElementsAndPrint(new TreeMap<>());         // giraffe,koala,lion,
```

Like we saw with the `Set` classes, `HashMap` prints the elements in an arbitrary ordering using the `hashCode()` of the key. `LinkedHashMap` prints the elements in the order in which they were inserted. Finally, `TreeMap` prints the elements based on the order of the keys.

Using our `HashMap` instance, we can try some `boolean` checks.

```
System.out.println(map.containsKey("lion"));    // true
System.out.println(map.containsValue("lion")); // false
System.out.println(map.size());     // 3
map.clear();
System.out.println(map.size());     // 0
System.out.println(map.isEmpty()); // true
```

The first two lines show that keys and values are checked separately. We can see that there are three key/value pairs in our map. Then we clear out the contents of the map and see that there are zero elements and it is empty.

Do you see why this doesn't compile?

```
System.out.println(map.contains("lion"));       // DOES NOT COMPILE
```

It doesn't compile because the `contains()` method is on the `Collection` interface but not the `Map` interface.

In the following sections, we show `Map` methods you might not be as familiar with.

Iterating through a *Map*

You saw the `forEach()` method earlier in the chapter. Note that it works a little differently on a `Map`. This time, the lambda used by the `forEach()` method has two parameters: the key and the value. Let's look at an example, shown here:

```
Map<Integer, Character> map = new HashMap<>();
map.put(1, 'a');
map.put(2, 'b');
map.put(3, 'c');
map.forEach((k, v) -> System.out.println(v));
```

The lambda has both the key and value as the parameters. It happens to print out the value but could do anything with the key and/or value. Interestingly, since we don't care about the key, this particular code could have been written with the `values()` method and a method reference instead.

```
map.values().forEach(System.out::println);
```

Another way of iterating over the data in a map is using `entrySet()`, which returns a set of `Map.Entry<K, V>` objects. If this type seems a little strange, don't worry! This is just a fancy way of storing the key/value pair in an object. It provides methods to retrieve the key and value of each pair.

```
map.entrySet().forEach(e ->
   System.out.println(e.getKey() + " " + e.getValue()));
```

In this case, each element `e` is of type `Map.Entry<Integer, Character>`.

Getting Values Safely

The `get()` method returns `null` if the requested key is not in the map. Sometimes you prefer to have a different value returned. Luckily, the `getOrDefault()` method makes this easy. Let's compare the two methods.

```
3: Map<Character, String> map = new HashMap<>();
4: map.put('x', "spot");
5: System.out.println("X marks the " + map.get('x'));
6: System.out.println("X marks the " + map.getOrDefault('x', ""));
7: System.out.println("Y marks the " + map.get('y'));
8: System.out.println("Y marks the " + map.getOrDefault('y', ""));
```

This code prints the following:

```
X marks the spot
X marks the spot
Y marks the null
Y marks the
```

As you can see, lines 5 and 6 have the same output because `get()` and `getOrDefault()` behave the same way when the key is present. They return the value mapped by that key. Lines 7 and 8 give different output, showing that `get()` returns `null` when the key is not present. By contrast, `getOrDefault()` returns the empty string we passed as a parameter.

Replacing Values

These methods are similar to the `List` version, except a key is involved:

```
21: Map<Integer, Integer> map = new HashMap<>();
22: map.put(1, 2);
23: map.put(2, 4);
24: Integer original = map.replace(2, 10); // 4
25: System.out.println(map);    // {1=2, 2=10}
26: map.replaceAll((k, v) -> k + v);
27: System.out.println(map);    // {1=3, 2=12}
```

Line 24 replaces the value for key `2` and returns the original value. Line 26 calls a function and sets the value of each element of the map to the result of that function. In our case, we added the key and value together.

Note that `replace()` and `replaceAll()` do not modify the `Map` if it does not contain the key. Contrast this with `put()`, which will always attempt to set a value.

Putting If Absent

The `putIfAbsent()` method sets a value in the map but skips it if the value is already set to a non-`null` value.

```
Map<String, String> favorites = new HashMap<>();
favorites.put("Jenny", "Bus Tour");
favorites.put("Tom", null);
favorites.putIfAbsent("Jenny", "Tram");
favorites.putIfAbsent("Sam", "Tram");
favorites.putIfAbsent("Tom", "Tram");
System.out.println(favorites); // {Tom=Tram, Jenny=Bus Tour, Sam=Tram}
```

As you can see, `Jenny`'s value is not updated because one was already present. `Sam` wasn't there at all, so he was added. `Tom` was present as a key but had a `null` value. Therefore, he was updated as well.

Merging Data

The `merge()` method adds logic of what to choose. Suppose we want to choose the ride with the longest name. We can write code to express this by passing a mapping function to the `merge()` method.

```
11: BiFunction<String, String, String> mapper = (v1, v2)
12:    -> v1.length()> v2.length() ? v1: v2;
13:
14: Map<String, String> favorites = new HashMap<>();
15: favorites.put("Jenny", "Bus Tour");
16: favorites.put("Tom", "Tram");
17:
18: String jenny = favorites.merge("Jenny", "Skyride", mapper);
19: String tom = favorites.merge("Tom", "Skyride", mapper);
20:
21: System.out.println(favorites); // {Tom=Skyride, Jenny=Bus Tour}
22: System.out.println(jenny);     // Bus Tour
23: System.out.println(tom);       // Skyride
```

The code on lines 11 and 12 takes two parameters and returns a value. Our implementation returns the one with the longest name. Line 18 calls this mapping function, and it sees that `Bus Tour` is longer than `Skyride`, so it leaves the value as `Bus Tour`. Line 19 calls this mapping function again. This time, `Tram` is shorter than `Skyride`, so the map is updated. Line 21 prints out the new map contents. Lines 22 and 23 show that the result is returned from `merge()`.

The `merge()` method also has logic for what happens if `null` values or missing keys are involved. In this case, it doesn't call the `BiFunction` at all, and it simply uses the new value.

```
BiFunction<String, String, String> mapper =
   (v1, v2) -> v1.length()> v2.length() ? v1 : v2;
Map<String, String> favorites = new HashMap<>();
favorites.put("Sam", null);
favorites.merge("Tom", "Skyride", mapper);
favorites.merge("Sam", "Skyride", mapper);
System.out.println(favorites);   // {Tom=Skyride, Sam=Skyride}
```

Notice that the mapping function isn't called. If it were, we'd have a `NullPointerException`. The mapping function is used only when there are two actual values to decide between.

The final thing to know about `merge()` is what happens when the mapping function is called and returns `null`. The key is removed from the map when this happens.

```
BiFunction<String, String, String> mapper = (v1, v2) -> null;
Map<String, String> favorites = new HashMap<>();
favorites.put("Jenny", "Bus Tour");
favorites.put("Tom", "Bus Tour");
```

```
favorites.merge("Jenny", "Skyride", mapper);
favorites.merge("Sam", "Skyride", mapper);
System.out.println(favorites);   // {Tom=Bus Tour, Sam=Skyride}
```

Tom was left alone since there was no `merge()` call for that key. Sam was added since that key was not in the original list. Jenny was removed because the mapping function returned `null`.

Table 9.7 shows all of these scenarios as a reference.

TABLE 9.7 Behavior of the `merge()` method

If the requested key	And mapping function returns	Then:
Has a `null` value in map	N/A (mapping function not called)	Update key's value in map with value parameter.
Has a non-`null` value in map	`null`	Remove key from map.
Has a non-`null` value in map	A non-`null` value	Set value to mapping function result.
Is not in map	N/A (mapping function not called)	Add key with value parameter to map directly without calling mapping function.

Sorting Data

We discussed "order" for the `TreeSet` and `TreeMap` classes. For numbers, order is obvious—it is numerical order. For `String` objects, order is defined according to the Unicode character mapping.

Remember 7Up from Chapter 4, "Core APIs"? When working with a `String`, numbers sort before letters, and uppercase letters sort before lowercase letters.

We use `Collections.sort()` in many of these examples. It returns `void` because the method parameter is what gets sorted.

You can also sort objects that you create yourself. Java provides an interface called `Comparable`. If your class implements `Comparable`, it can be used in data structures that require comparison. In fact, you've seen many `Comparable` classes in this book include `String`, `StringBuilder`, `BigDecimal`, `BigInteger` and the primitive wrapper classes. There is also a class called `Comparator`, which is used to specify that you want to use a different order than the object itself provides.

`Comparable` and `Comparator` are similar enough to be tricky. The exam likes to see if it can trick you into mixing up the two. Don't be confused! In this section, we discuss `Comparable` first. Then, as we go through `Comparator`, we point out all of the differences.

Creating a *Comparable* Class

The `Comparable` interface has only one method. In fact, this is the entire interface:

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

The generic `T` lets you implement this method and specify the type of your object. This lets you avoid a cast when implementing `compareTo()`. Any object can be `Comparable`. For example, we have a bunch of ducks and want to sort them by name. First, we create a record that inherits `Comparable<Duck>`, and then we implement the `compareTo()` method.

```
public record Duck(String name) implements Comparable<Duck> {
   public int compareTo(Duck d) {
      return name.compareTo(d.name);  // Sorts ascendingly by name
   }
}
```

Next, we can sort the ducks as follows:

```
11: var ducks = new ArrayList<Duck>();
12: ducks.add(new Duck("Quack"));
13: ducks.add(new Duck("Puddles"));
14: Collections.sort(ducks);  // sort by name
15: System.out.print(ducks); // [Duck[name=Puddles],
Duck[name=Quack]]
```

If we didn't implement the `Comparable` interface, all we have is a method named `compareTo()`, and line 14 would not compile. We could also implement `Comparable<Object>` or some other class for `T`, but this wouldn't be as useful for sorting a group of `Duck` objects.

Finally, the `Duck` class implements `compareTo()`. Since `Duck` is comparing objects of type `String` and the `String` class already has a `compareTo()` method, it can just

delegate.

You might have noticed we use a record here. From Chapter 7, "Beyond Classes," a record provides a lot of useful boilerplate code like constructors and meaningful implementations of `toString()`. Just remember, both records and classes can implement `Comparable`.

Designing a *compareTo()* method

In the previous example, we relied on the built-in `String compareTo()` method, but often you need to create your own. When writing a `compareTo()` method, the most important part is the return value. The following rules should apply to the return type of your `compareTo()` method:

- The number `0` is returned when the current object is equivalent to the argument to `compareTo()`.
- A negative number (less than `0`) is returned when the current object is smaller than the argument to `compareTo()`.
- A positive number (greater than `0`) is returned when the current object is larger than the argument to `compareTo()`.

Let's look at an implementation of `compareTo()` that compares numbers instead of `String` objects:

```
2: public record ZooDuck(int id, String name) implements
Comparable<ZooDuck> {
3:    public int compareTo(ZooDuck d) {
4:       return id - d.id;  // Sorts ascendingly by id
5:    }
6: }
```

Line 4 shows one way to compare two `int` values. We could have used `Integer.compare(id, d.id)`, but we wanted to show you how to create your own. Be sure you can recognize both approaches. Remember that `id - d.id` sorts in ascending order, and `d.id - id` sorts in descending order.

Let's try this new method out in some code:

```
21: var d1 = new ZooDuck (5, "Daffy");
22: var d2 = new ZooDuck(7, "Donald");
23: System.out.println(d1.compareTo(d2));   // -2
24: System.out.println(d1.compareTo(d1));   // 0
25: System.out.println(d2.compareTo(d1));   // 2
```

Line 23 compares a smaller `id` to a larger one, and therefore it prints a negative number. Line 24 compares animals with the same `id`, and therefore it prints `0`. Line 25 compares a larger `id` to a smaller one, and therefore it returns a positive number.

Casting the *compareTo()* Argument

When dealing with legacy code or code that does not use generics, the `compareTo()` method requires a cast since it is passed an `Object`. We can accomplish this using pattern-matching that you saw in Chapter 3.

```
public record LegacyDuck(String name) implements Comparable {
   public int compareTo(Object obj) {
      if(obj instanceof LegacyDuck d)
         return name.compareTo(d.name);
      throw new UnsupportedOperationException("Not a duck");
   } }
```

Since we don't specify a generic type for `Comparable`, Java assumes that we want an `Object`.

Checking for *null*

When working with `Comparable` and `Comparator` in this chapter, we tend to assume the data has values, but this is not always the case. When writing your own compare methods, you should check the data before comparing it if it is not validated ahead of time.

```
public record MissingDuck(String name) implements
Comparable<MissingDuck>  {
   public int compareTo(MissingDuck quack) {
      if (quack == null)
         throw new IllegalArgumentException("Poorly formed duck!");
      if (this.name == null && quack.name == null)
         return 0;
      else if (this.name == null) return -1;
      else if (quack.name == null) return 1;
      else return name.compareTo(quack.name);
   } }
```

This method throws an exception if it is passed a `null MissingDuck` object. What about the ordering? If the `name` of a duck is `null`, it's sorted first.

Keeping *compareTo()* and *equals()* Consistent

If you write a class that implements `Comparable`, you introduce new business logic for determining equality. The `compareTo()` method returns `0` if two objects are equal, while your `equals()` method returns `true` if two objects are equal. A *natural ordering* that uses `compareTo()` is said to be *consistent with equals* if, and only if, `x.equals(y)` is `true` whenever `x.compareTo(y)` equals `0`.

Similarly, `x.equals(y)` must be `false` whenever `x.compareTo(y)` is not `0`. You are strongly encouraged to make your `Comparable` classes consistent with equals because not all collection classes behave predictably if the `compareTo()` and `equals()` methods are not consistent.

For example, the following `Product` class defines a `compareTo()` method that is not consistent with equals:

```
public class Product implements Comparable<Product> {
   private int id;
   private String name;

   public int hashCode() { return id; }

   public boolean equals(Object obj) {
      if (obj instanceof Product other)
         return this.id == other.id;
      return false;
   }

   public int compareTo(Product obj) {
      return this.name.compareTo(obj.name);
   } }
```

This class checks equality with `id`, but orders by `name`. Assuming names are not unique, this means we could have many pairs of elements in which `compareTo()` returns `0`, but `equals()` returns `false`.

One way to fix this is to update the methods to rely on the same attributes. If you still need to sort things by `name`, you can use a `Comparator` defined outside the class, as shown in the next section.

Comparing Data with a *Comparator*

Sometimes you want to sort an object that did not implement `Comparable`, or you want to sort objects in different ways at different times. Suppose that we add `weight` to our `Duck` class.

```
public record Duck(String name, int weight) implements
Comparable<Duck> {
   public int compareTo(Duck d) {
      return name.compareTo(d.name);
   }
```

```
   public String toString() { return name; }
}
```

We also override `toString()` so our next set of output is shorter. We now have the following:

```
11: Comparator<Duck> byWeight = new Comparator<>() {
12:    public int compare(Duck d1, Duck d2) {
13:       return d1.weight() - d2.weight();
14:    }
15: };
16: var ducks = new ArrayList<Duck>();
17: ducks.add(new Duck("Quack", 7));
18: ducks.add(new Duck("Puddles", 10));
19: Collections.sort(ducks);
20: System.out.println(ducks);  // [Puddles, Quack]
21: Collections.sort(ducks, byWeight);
22: System.out.println(ducks);  // [Quack, Puddles]
```

The `Duck` class itself can define only one `compareTo()` method. In this case, `name` was chosen. If we want to sort by something else, we have to define that sort order outside the `compareTo()` method using a separate class or lambda expression.

Lines 11–15 show how to define a `Comparator` using an anonymous class. On lines 19–22, we sort with the class's internal `Comparator` and then with the separate `Comparator` to see the difference in output.

`Comparator` is a functional interface since there is only one abstract method to implement. This means that we can rewrite the `Comparator` on lines 11–15 using a lambda expression, as shown here:

```
Comparator<Duck> byWeight = (d1, d2) -> d1.weight()-d2.weight();
```

Alternatively, we can use a method reference and a helper method to specify that we want to sort by weight.

```
Comparator<Duck> byWeight = Comparator.comparing(Duck::weight);
```

In this example, `Comparator.comparing()` is a `static` interface method that creates a `Comparator` given a lambda expression or method reference. Convenient, isn't it?

Is *Comparable* a Functional Interface?

We said that `Comparator` is a functional interface because it has a single abstract method. `Comparable` is also a functional interface since it also has a single abstract method. However, using a lambda for `Comparable` would be silly. The point of `Comparable` is to implement it inside the object being compared.

Comparing *Comparable* and *Comparator*

There are several differences between `Comparable` and `Comparator`. We've listed them for you in Table 9.8.

TABLE 9.8 Comparison of `Comparable` and `Comparator`

Difference	`Comparable`	`Comparator`
Package name	`java.lang`	`java.util`
Interface must be implemented by class comparing?	Yes	No
Method name in interface	`compareTo()`	`compare()`
Number of parameters	1	2
Common to declare using a lambda	No	Yes

Memorize this table—really. The exam will try to trick you by mixing up the two and seeing if you can catch it. Do you see why this doesn't compile?

```
var byWeight = new Comparator<Duck>() { // DOES NOT COMPILE
   public int compareTo(Duck d1, Duck d2) {
      return  d1.getWeight()-d2.getWeight();
   }
};
```

The method name is wrong. A `Comparator` must implement a method named `compare()`. Pay special attention to method names and the number of parameters when you see `Comparator` and `Comparable` in questions.

Comparing Multiple Fields

When writing a `Comparator` that compares multiple instance variables, the code gets a little messy. Suppose that we have a `Squirrel` record, as shown here:

```
public record Squirrel(int weight, String species) {}
```

We want to write a `Comparator` to sort by species name. If two squirrels are from the same species, we want to sort the one that weighs the least first. We could do this with code that looks like this:

```
public class MultiFieldComparator implements Comparator<Squirrel> {
   public int compare(Squirrel s1, Squirrel s2) {
      int result = s1.species().compareTo(s2.species());
      if (result != 0) return result;
      else return s1.weight() - s2.weight();
   } }
```

This works assuming no species' names are `null`. It checks one field. If they don't match, we are finished sorting. If they do match, it looks at the next field. This isn't easy to read, though. It is also easy to get wrong. Changing `!=` to `==` breaks the sort completely.

Alternatively, we can use method references and build the `Comparator`. This code represents logic for the same comparison:

```
Comparator<Squirrel> c = Comparator.comparing(Squirrel::species)
   .thenComparingInt(Squirrel::weight);
```

This time, we chain the methods. First, we create a `Comparator` on species ascending. Then, if there is a tie, we sort by weight. We can also sort in descending order. Some methods on `Comparator`, like `thenComparingInt()`, are `default` methods.

Suppose we want to sort in descending order by species.

```
var c = Comparator.comparing(Squirrel::species).reversed();
```

Table 9.9 shows the helper methods you should know for building a `Comparator`. We've omitted the parameter types to keep you focused on the methods. They use many of the functional interfaces you learned about in the previous chapter.

TABLE 9.9 Helper static methods for building a `Comparator`

Method	Description
`comparing(function)`	Compare by results of function that returns any `Object` (or primitive autoboxed into `Object`) .
`comparingDouble(function)`	Compare by results of function that returns `double`.
`comparingInt(function)`	Compare by results of function that returns `int`.
`comparingLong(function)`	Compare by results of function that returns `long`.
`naturalOrder()`	Sort using order specified by the `Comparable` implementation on the object itself.
`reverseOrder()`	Sort using reverse of order specified by `Comparable` implementation on the object itself.

Table 9.10 shows the methods that you can chain to a `Comparator` to further specify its behavior.

TABLE 9.1O Helper default methods for building a `Comparator`

Method	Description
`reversed()`	Reverse order of chained `Comparator`.
`thenComparing(function)`	If previous `Comparator` returns `0`, use this comparator function that returns `Object` or can be autoboxed into one. Otherwise, return result from previous Comparator.
`thenComparingDouble(function)`	If previous `Comparator` returns `0`, use this comparator function that returns `double`. Otherwise, return result from previous `Comparator`.
`thenComparingInt(function)`	If previous `Comparator` returns `0`, use this comparator function that returns `int`. Otherwise, return result from previous `Comparator`.
`thenComparingLong(function)`	If previous `Comparator` returns `0`, use this comparator function that returns `long`. Otherwise, return result from previous `Comparator`.

You've probably noticed by now that we often ignore `null` values in checking equality and comparing objects. This works fine for the exam. In the real world, though, things aren't so neat. You will have to decide how to handle `null` values or prevent them from being in your object.

Sorting and Searching

Now that you've learned all about `Comparable` and `Comparator`, we can finally do something useful with them, like sorting. The `Collections.sort()` method uses the `compareTo()` method to sort. It expects the objects to be sorted to be `Comparable`.

```
2:  public class SortRabbits {
3:     static record Rabbit(int id) {}
4:     public static void main(String[] args) {
5:        List<Rabbit> rabbits = new ArrayList<>();
6:        rabbits.add(new Rabbit(3));
7:        rabbits.add(new  Rabbit(1));
8:        Collections.sort(rabbits);  // DOES NOT COMPILE
9:     }
10: }
```

Java knows that the `Rabbit` record is not `Comparable`. It knows sorting will fail, so it doesn't even let the code compile. You can fix this by passing a `Comparator` to `sort()`. Remember that a `Comparator` is useful when you want to specify sort order without using a `compareTo()` method.

```
8:        Comparator<Rabbit> c = (r1, r2) -> r1.id - r2.id;
9:        Collections.sort(rabbits, c);
10:       System.out.println(rabbits); // [Rabbit[id=1],
Rabbit[id=3]]
```

Suppose you want to sort the rabbits in descending order. You could change the `Comparator` to `r2.id - r1.id`. Alternatively, you could reverse the contents of the list afterward:

```
8:        Comparator<Rabbit> c = (r1, r2) -> r1.id - r2.id;
9:        Collections.sort(rabbits, c);
10:       Collections.reverse(rabbits);
11:       System.out.println(rabbits); // [Rabbit[id=3],
Rabbit[id=1]]
```

The `sort()` and `binarySearch()` methods allow you to pass in a `Comparator` object when you don't want to use the natural order.

Reviewing *binarySearch()*

The `binarySearch()` method requires a sorted `List`.

```
11: List<Integer> list = Arrays.asList(6,9,1,8);
12: Collections.sort(list); // [1, 6, 8, 9]
13:  System.out.println(Collections.binarySearch(list,  6));  //  1
14:  System.out.println(Collections.binarySearch(list,  3));  //  -2
```

Line 12 sorts the `List` so we can call binary search properly. Line 13 prints the index at which a match is found. Line 14 prints one less than the negated index of where the requested value would need to be inserted. The number `3` would need to be inserted at index `1` (after the number `1` but before the number `6`). Negating that gives us `-1`, and subtracting `1` gives us `-2`.

There is a trick in working with `binarySearch()`. What do you think the following outputs?

```
3: var names = Arrays.asList("Fluffy", "Hoppy");
4: Comparator<String> c = Comparator.reverseOrder();
5: var index = Collections.binarySearch(names, "Hoppy", c);
6: System.out.println(index);
```

The answer happens to be `-1`. Before you panic, you don't need to know that the answer is `-1`. You do need to know that the answer is not defined. Line 3 creates a list, `[Fluffy, Hoppy]`. This list happens to be sorted in ascending order. Line 4 creates a `Comparator` that reverses the natural order. Line 5 requests a binary search in descending order. Since the list is not in that order, we don't meet the precondition for doing a search.

While the result of calling `binarySearch()` on an improperly sorted list is undefined, sometimes you can get lucky. For example, search starts in the middle of an odd-numbered list. If you happen to ask for the middle element, the index returned will be what you expect.

Earlier in the chapter, we talked about collections that require classes to implement `Comparable`. Unlike sorting, they don't check that you have implemented `Comparable` at compile time.

Going back to our `Rabbit` that does not implement `Comparable`, we try to add it to a `TreeSet`:

```
2:  public class UseTreeSet {
3:      record Rabbit(int id) {}
4:      public static void main(String[] args) {
5:         Set<Duck> ducks = new TreeSet<>();
6:         ducks.add(new  Duck("Puddles"));
7:
8:         Set<Rabbit> rabbits = new TreeSet<>();
9:         rabbits.add(new Rabbit(1));  // ClassCastException
10: } }
```

Line 6 is fine. `Duck` does implement `Comparable`. `TreeSet` is able to sort it into the proper position in the set. Line 9 is a problem. When `TreeSet` tries to sort it, Java discovers the fact that `Rabbit` does not implement `Comparable`. Java throws an exception that looks like this:

```
Exception in thread "main" java.lang.ClassCastException:
   class UseTreeSet$Rabbit cannot be cast to class
java.lang.Comparable
```

It may seem weird for this exception to be thrown when the first object is added to the set. After all, there is nothing to compare yet. Java works this way for consistency.

Just like searching and sorting, you can tell collections that require sorting that you want to use a specific `Comparator`. For example:

```
8: Set<Rabbit> rabbits = new TreeSet<>((r1, r2) -> r1.id - r2.id);
9: rabbits.add(new Rabbit(1));
```

Now Java knows that you want to sort by `id`, and all is well. A `Comparator` is a helpful object. It lets you separate sort order from the object to be sorted. Notice that line 9 in both of the previous examples is the same. It's the declaration of the `TreeSet` that has changed.

Sorting a *List*

While you can call `Collections.sort(list)`, you can also sort directly on the list object.

```
3: List<String> bunnies = new ArrayList<>();
4: bunnies.add("long ear");
5: bunnies.add("floppy");
6: bunnies.add("hoppy");
7: System.out.println(bunnies);     // [long ear, floppy, hoppy]
8: bunnies.sort((b1, b2) -> b1.compareTo(b2));
9: System.out.println(bunnies);     // [floppy, hoppy, long ear]
```

On line 8, we sort the list alphabetically. The `sort()` method takes a `Comparator` that provides the sort order. Remember that `Comparator` takes two parameters and returns an `int`. If you need a review of what the return value of a `compare()` operation means, check the `Comparator` section in this chapter or the "Comparing" section in Chapter 4. This is really important to memorize!

Introducing Sequenced Collections

New to Java 21 are *sequenced collections*, which includes the three interfaces from Figure 9.1.

- `SequencedCollection`
- `SequencedSet`
- `SequencedMap`

A sequenced collection is a collection in which the encounter order is well-defined. By *encounter order*, it means all of the elements can be read in a repeatable way. While the elements of the collection may be sorted, it is not required.

Working with *SequencedCollection*

Let's start with the simplest example of a sequenced collection, one you've been working with throughout this book. An `ArrayList` is a `SequencedCollection`, as its first and last elements are well-defined, as is the ordering of all elements in between.

Table 9.11 includes various methods available on a `SequencedCollection`.

TABLE 9.11 `SequencedCollection` Methods

Method	Description
`addFirst(E e)`	Adds element as the first element in the collection
`addLast(E e)`	Adds element as the last element in the collection
`getFirst()`	Retrieves the first element in the collection
`getLast()`	Retrieves the last element in the collection
`removeFirst()`	Removes the first element in the collection
`removeLast()`	Removes the last element in the collection
`reversed()`	Returns a reverse-ordered view of the collection

For `ArrayList`, it should be pretty obvious how most of these methods are implemented. For example, to add or retrieve the first element in the list, you could

call the `add(0,e)` and `get(0)`, respectively. The purpose of the `SequencedCollection` interface isn't necessarily to add new functionality, but to make it easier to work with related types. For example, let's say we have the following method that welcomes the next visitor to the zoo:

```
public void welcomeNext(SequencedCollection<String> visitors) {
   System.out.println("Welcome to the Zoo! " + visitors.getFirst());
   visitors.removeFirst();
}
```

We can now apply various sequenced collections to this method:

```
var visitArrayList = new ArrayList<String>(List.of("Huey", "Dewey",
"Louie"));
var visitLinkedList = new LinkedList<String>(List.of("Moe", "Larry",
"Shemp"));
var visitTreeSet = new TreeSet<String>(Set.of("Alvin", "Simon",
"Theodore"));

welcomeNext(visitArrayList);  // Welcome to the Zoo! Huey
welcomeNext(visitLinkedList); // Welcome to the Zoo! Moe
welcomeNext(visitTreeSet);    // Welcome to the Zoo! Alvin
```

Sequenced collections grant us the ability to work with lots of different types that all have a well-defined encounter order. Using some of the other methods from Table 9.11, we can even rearrange the elements.

```
public void moveToEnd(SequencedCollection<String> visitors) {
   visitors.addLast(visitors.removeFirst());
}
```

What happens if we call this new method on another group of collections?

```
var visitArrayList = new ArrayList<String>(
   List.of("Bluey", "Bingo", "Socks"));
var visitLinkedList = new LinkedList<String>(List.of("Garfield",
"Odie"));
var visitTreeSet = new TreeSet<String>(Set.of("Tom", "Jerry"));

moveToEnd(visitArrayList);
welcomeNext(visitArrayList);  // Welcome to the Zoo! Bingo

moveToEnd(visitLinkedList);
welcomeNext(visitLinkedList); // Welcome to the Zoo! Odie

moveToEnd(visitTreeSet);      //
java.lang.UnsupportedOperationException
welcomeNext(visitTreeSet);
```

Uh-oh, why didn't the last example work? Just because a method implements `SequencedCollection` doesn't mean the class supports all of the methods in Table

9.11. In this example, the `addLast()` call fails at runtime because you can't insert an item at the end of a sorted structure. Doing so could violate the comparator within the `TreeSet`.

For the exam, you don't need to know which collections support which methods found in Table 9.11, but you should know the difference between a sequenced collection and a sorted collection.

A `SequencedSet` is a subtype of `SequencedCollection`; therefore, it inherits all its methods. It only applies to `SequencedCollection` classes that also implement `Set`, such as `LinkedHashSet` and `TreeSet`.

Working with *SequencedMap*

As you can probably guess, a `SequencedMap` is a `Map` with a defined encounter order. We define common methods in Table 9.12.

TABLE 9.12 Common `SequencedMap` Methods

Method	Description
`firstEntry()`	Retrieves the first key/value pair in the map
`lastEntry()`	Retrieves the last key/value pair in the map
`pollFirstEntry()`	Removes and retrieves the first key/value pair in the map
`pollLastEntry()`	Removes and retrieves the last key/value pair in the map
`putFirst(K k, V v)`	Adds the key/value pair as the first element in the map
`putLast(K k, V v)`	Adds the key/value pair as the last element in the map
`reversed()`	Returns a reverse-ordered view of the map

Let's define a method for working with `SequencedMap`.

```
public void welcomeNext(SequencedMap<String, String> visitors) {
   System.out.println("Welcome to the Zoo! " +
visitors.pollFirstEntry());
}
```

What do you think the following snippet prints?

```
var visitHashMap = new HashMap<String,String>(
   Map.of("1", "Yakko", "2", "Wakko", "3", "Dot"));
welcomeNext(visitHashMap);
```

Trick question! It actually doesn't compile. Like we explained with `HashSet` earlier, a `HashMap` does not have an ordering, so it cannot be used as a `SequencedMap`. What about this example?

```
var visitTreeMap = new TreeMap<String,String>(
   Map.of("Pink", "Blossom", "Green", "Buttercup", "Blue",
"Bubbles"));
welcomeNext(visitTreeMap);
```

If you guessed `Welcome to the Zoo! Blue=Bubbles`, then you were paying attention when we covered `TreeMap`. A `TreeMap` sorts things by the natural order of its keys, not the order in which they were added to the map. Since `Blue` is the first key in sorted order, it is the first pair printed.

Most of the collections that you have worked with throughout this book are sequenced. Two notable exceptions are `HashSet` and `HashMap`.

Reviewing Collection Types

We conclude this part of the chapter by reviewing rules that apply to various collection types, as well as an overview of all the types covered in this chapter.

Using Unmodifiable Wrapper Views

An *unmodifiable view* is a wrapper object around a collection that cannot be modified through the view itself. While the view object cannot be modified, the underlying data can still be modified.

There are four methods you should be familiar with for the exam that create unmodifiable views of a collection:

```
Collection<String> coll =
Collections.unmodifiableCollection(List.of("brown"));
List<String> list       =
Collections.unmodifiableList(List.of("orange"));
Set<String> set         =
Collections.unmodifiableSet(Set.of("green"));
```

```
Map<String,Integer> map = Collections.unmodifiableMap(Map.of("red",
1));
```

Let's consider some code that uses them:

```
10: Map<String, Integer> map = new TreeMap<>();
11: map.put("blue", 41);
12: map.put("red", 90);
13: List<String> list = Arrays.asList("green", "yellow");
14: Set<String> set = new HashSet<>(list);
15:
16: Map<String, Integer> mapView = Collections.unmodifiableMap(map);
18: Collection<String> collView  =
Collections.unmodifiableCollection(list);
19: List<String> listView         = Collections.unmodifiableList(list);
20: Set<String> setView           = Collections.unmodifiableSet(set);
```

As you might expect, trying to modify an unmodifiable view throws an exception. When run independently, each of the following compiles and throws an `UnsupportedOperationException` at runtime.

```
collView.add("pink");
setView.remove("green");
mapView.put("blue",  42);
```

However, since it is a view, nothing prevents you from changing the original values. For example:

```
24: System.out.println(mapView);   // {blue=41, red=90}
25: System.out.println(collView);  // [green, yellow]
26: System.out.println(listView);  // [green, yellow]
27: System.out.println(setView);   // [green, yellow]
28:
29: map.put("blue", 105);
30: list.set(1, "purple");
31:
32: System.out.println(mapView);   // {blue=105, red=90}
33: System.out.println(collView);  // [green, purple]
34: System.out.println(listView);  // [green, purple]
35: System.out.println(setView);   // [green, yellow]
```

On line 29, notice that the value of `blue` is changed to `105` in the original `TreeMap` and it shows up as changed in `mapView` on line 32. The `list` variable created on line 13 refers to a fixed sized backed array. Which means both `collView` and `listView` represent a view of a `List` that refers to a backed array. Since the value is set on line 30, it remains the same size, and the change properly shows up in our views.

However, `setView` has not changed value. The constructor on line 14 makes a new set that is disconnected from the original data structure. This means line 30 has no effect on `set`.

What happens if we try adding elements to these collections?

```
36: set.add("orange");
37: System.out.println(setView);   // [green, yellow, orange]
38:
39: list.add("orange");            // UnsupportedOperationException
```

Line 36 successfully modifies the underlying `HashSet`, with the changes reflected in the view on line 37. Line 39 throws an exception at runtime. Remember, the `list` was created with `Arrays.asList()`. As we saw earlier in the chapter, you can replace elements in such objects but you cannot add/remove elements. For the exam, remember to check the type of the underlying object to determine if things can be added, removed, or modified.

Comparing Collection Types

Make sure that you can fill in Table 9.13 to compare the four collection types from memory.

TABLE 9.1C Java Collections Framework types

Type	Can contain duplicate elements?	Elements always ordered?	Has keys and values?	Must add/remove in specific order?
`List`	Yes	Yes (by index)	No	No
`Queue`	Yes	Yes (retrieved in defined order)	No	Yes
`Set`	No	No	No	No
`Map`	Yes (for values)	No	Yes	No

Additionally, make sure you can fill in Table 9.14 to describe the types on the exam.

TABLE 9.14 Collection classes

Type	Java Collections Framework interfaces	Ordered?	Sorted?	Calls `hashCode`?	Calls `compareTo`?
`ArrayDeque`	`Deque` `SequencedCollection`	Yes	No	No	No
`ArrayList`	`List` `SequencedCollection`	Yes	No	No	No
`HashMap`	`Map`	No	No	Yes	No
`HashSet`	`Set`	No	No	Yes	No

`LinkedList`	`Deque`
	`List`
	`SequencedCollection`
		Yes	No	No	No

`LinkedHashSet`	`Set` `SequencedSet`	Yes	No	No	No
`LinkedHashMap`	`Map` `SequencedMap`	Yes	No	No	No
`TreeMap`	`Map` `SequencedMap`	Yes	Yes	No	Yes

`TreeSet`	`Set`
	`SequencedCollection`
	`SequencedSet`
		Yes	Yes	No	Yes

The exam expects you to know that data structures that involve sorting require you to tread carefully with `null`. For sorted sets, that means `null` is not permitted; and for sorted maps, this means `null` keys are not permitted.

Finally, the exam expects you to be able to choose the right collection type given a description of a problem. We recommend first identifying which type of collection the question is asking about. Figure out whether you are looking for a list, map, queue, or set. This lets you eliminate a number of answers. Then you can figure out which of the remaining choices is the best answer.

Real World Scenario

Older Collections

There are a few collections that are no longer on the exam but that you might come across in older code. All three were early Java data structures you could use with threads.

- `Vector`: Implements `List`
- `Hashtable`: Implements `Map`
- `Stack`: Implements `List`

These classes are rarely used anymore, as there are much better concurrent alternatives that we cover in Chapter 13.

Working with Generics

We conclude this chapter with one of the most useful, and at times most confusing, features in the Java language: generics. In this section, we present more advanced topics including creating generic classes and methods.

Creating Generic Classes

You can introduce generics into your own classes. The syntax for introducing a generic is to declare a *formal type parameter* in angle brackets. For example, the following class named `Crate` has a generic type variable declared after the name of the class:

```
public class Crate<T> {
   private T contents;
   public T lookInCrate() {
      return contents;
   }
   public void packCrate(T contents) {
      this.contents = contents;
   }
}
```

The generic type `T` is available anywhere within the `Crate` class. When you instantiate the class, you tell the compiler what `T` should be for that particular

instance.

Naming Conventions for Generics

A type parameter can be named anything you want. The convention is to use single uppercase letters to make it obvious that they aren't real class names. The following are common letters to use:

- `E` for an element
- `K` for a map key
- `V` for a map value
- `N` for a number
- `T` for a generic data type
- `S`, `U`, `V`, and so forth for multiple generic types

For example, suppose an `Elephant` class exists, and we are moving our elephant to a new and larger enclosure in our zoo. (The San Diego Zoo did this in 2009. It was interesting seeing the large metal crate.)

```
Elephant elephant = new Elephant();
Crate<Elephant> crateForElephant = new Crate<>();
crateForElephant.packCrate(elephant);
Elephant inNewHome = crateForElephant.lookInCrate();
```

To be fair, we didn't pack the crate so much as the elephant walked into it. However, you can see that the `Crate` class is able to deal with an `Elephant` without knowing anything about it.

This probably doesn't seem particularly impressive. We could have just typed in `Elephant` instead of `T` when coding `Crate`. What if we wanted to create a `Crate` for another animal?

```
Crate<Zebra> crateForZebra = new Crate<>();
```

Now we couldn't have simply hard-coded `Elephant` in the `Crate` class since a `Zebra` is not an `Elephant`. However, we could have created an `Animal` superclass or interface and used that in `Crate`.

Generic classes become useful when the classes used as the type parameter can have absolutely nothing to do with each other. For example, we need to ship our 120-

pound robot to another city.

```
Robot joeBot = new Robot();
Crate<Robot> robotCrate = new Crate<>();
robotCrate.packCrate(joeBot);
// ship to Houston
Robot atDestination = robotCrate.lookInCrate();
```

Now it is starting to get interesting. The `Crate` class works with any type of class. Before generics, we would have needed `Crate` to use the `Object` class for its instance variable, which would have put the burden on the caller to cast the object it receives on emptying the crate.

In addition to `Crate` not needing to know about the objects that go into it, those objects don't need to know about `Crate`. We aren't requiring the objects to implement an interface named `Crateable` or the like. A class can be put in the `Crate` without any changes at all.

Don't worry if you can't think of a use for generic classes of your own. Unless you are writing a library for others to reuse, generics hardly show up in the class definitions you write. You've already seen them frequently in the code you call, such as functional interfaces and collections.

Generic classes aren't limited to having a single type parameter. This class shows two generic parameters.

```
public class SizeLimitedCrate<T, U> {
   private T contents;
   private U sizeLimit;
   public SizeLimitedCrate(T contents, U sizeLimit) {
      this.contents = contents;
      this.sizeLimit = sizeLimit;
   } }
```

`T` represents the type that we are putting in the crate. `U` represents the unit that we are using to measure the maximum size for the crate. To use this generic class, we can write the following:

```
Elephant elephant = new Elephant();
Integer numPounds = 15_000;
```

```
SizeLimitedCrate<Elephant, Integer> c1
   = new SizeLimitedCrate<>(elephant, numPounds);
```

Here we specify that the type is `Elephant`, and the unit is `Integer`. We also throw in a reminder that numeric literals can contain underscores.

Understanding Type Erasure

Specifying a generic type allows the compiler to enforce proper use of the generic type. For example, specifying the generic type of `Crate` as `Robot` is like replacing the `T` in the `Crate` class with `Robot`. However, this is just for compile time.

Behind the scenes, the compiler replaces all references to `T` in `Crate` with `Object`. In other words, after the code compiles, your generics are just `Object` types. The `Crate` class looks like the following at runtime:

```
public class Crate {
   private Object contents;
   public Object lookInCrate() {
      return contents;
   }
   public void packCrate(Object contents) {
      this.contents = contents;
   }
}
```

This means there is only one class file. There aren't different copies for different parameterized types. (Some other languages work that way.) This process of removing the generics syntax from your code is referred to as *type erasure*. Type erasure allows your code to be compatible with older versions of Java that do not contain generics.

The compiler adds the relevant casts for your code to work with this type of erased class. For example, you type the following:

```
Robot r = crate.lookInCrate();
```

The compiler turns it into the following:

```
Robot r = (Robot) crate.lookInCrate();
```

In the following sections, we look at the implications of generics and type erasure for method declarations.

Overloading a Generic Method

Only one of these two methods is allowed in a class because type erasure will reduce both sets of arguments to `(List input)`.

```
public class LongTailAnimal {
   protected void chew(List<Object> input) {}
   protected void chew(List<Double> input) {}  // DOES NOT COMPILE
}
```

For the same reason, you also can't overload a generic method from a parent class.

```
public class LongTailAnimal {
   protected void chew(List<Object> input) {}
}

public class Anteater extends LongTailAnimal {
   protected void chew(List<Double> input) {}  // DOES NOT COMPILE
}
```

Both of these examples fail to compile because of type erasure. In the compiled form, the generic type is dropped, and it appears as an invalid overloaded method. Now, let's look at another version of the same subclass:

```
public class Anteater extends LongTailAnimal {
   protected void chew(List<Object> input) {}
   protected void chew(ArrayList<Double> input) {}
}
```

The first `chew()` method compiles because it uses the same generic type in the overridden method as the one defined in the parent class. The second `chew()` method compiles as well. However, it is an overloaded method because one of the method arguments is a `List` and the other is an `ArrayList`. When working with generic methods, it's important to consider the underlying type.

Returning Generic Types

When you're working with overridden methods that return generics, the return values must be covariant. In terms of generics, this means the return type of the class or interface declared in the overriding method must be a subtype of the class defined in the parent class. The generic parameter type must match its parent's type exactly.

Given the following declaration for the `Mammal` class, which of the two subclasses, `Monkey` and `Goat`, compile?

```
public class Mammal {
   public List<CharSequence> play() { … }
   public CharSequence sleep() { … }
}

public class Monkey extends Mammal {
   public ArrayList<CharSequence> play() { … }
```

```
}

public class Goat extends Mammal {
   public List<String> play() { … }  // DOES NOT COMPILE
   public String sleep() { … }
}
```

The `Monkey` class compiles because `ArrayList` is a subtype of `List`. The `play()` method in the `Goat` class does not compile, though. For the return types to be covariant, the generic type parameter must match. Even though `String` is a subtype of `CharSequence`, it does not exactly match the generic type defined in the `Mammal` class. Therefore, this is considered an invalid override.

Notice that the `sleep()` method in the `Goat` class does compile since `String` is a subtype of `CharSequence`. This example shows that covariance applies to the return type, just not the generic parameter type.

For the exam, it might be helpful for you to apply type erasure to questions involving generics to ensure that they compile properly. Once you've determined which methods are overridden and which are being overloaded, work backward, making sure the generic types match for overridden methods. And remember, generic methods cannot be overloaded by changing the generic parameter type only.

Implementing Generic Interfaces

Just like a class, an interface can declare a formal type parameter. For example, the following `Shippable` interface uses a generic type as the argument to its `ship()` method:

```
public interface Shippable<T> {
   void ship(T t);
}
```

There are three ways a class can approach implementing this interface. The first is to specify the generic type in the class. The following concrete class says that it deals only with robots. This lets it declare the `ship()` method with a `Robot` parameter.

```
class ShippableRobotCrate implements Shippable<Robot> {
   public void ship(Robot t) { }
}
```

The next way is to create a generic class. The following concrete class allows the caller to specify the type of the generic:

```
class ShippableAbstractCrate<U> implements Shippable<U> {
   public void ship(U t) { }
}
```

In this example, the type parameter could have been named anything, including `T`. We used `U` in the example to avoid confusion about what `T` refers to. The exam won't mind trying to confuse you by using the same type parameter name.

The final way is to not use generics at all. This is the old way of writing code. It generates a compiler warning about `Shippable` being a *raw type*, but it does compile. Here the `ship()` method has an `Object` parameter since the generic type is not defined:

```
class ShippableCrate implements Shippable {
   public void ship(Object t) { }
}
```

Real World Scenario

What You Can't Do with Generic Types

There are some limitations on what you can do with a generic type. These aren't on the exam, but it will be helpful to refer to this scenario when you are writing practice programs and run into one of these situations.

Most of the limitations are due to type erasure. Oracle refers to a type whose information is fully available at runtime as a *reifiable type*. Reifiable types can do anything that Java allows. Non-reifiable types have some limitations.

Here are the things that you can't do with generics (and by "can't," we mean without resorting to contortions like passing in a class object):

- **Call a constructor:** Writing `new T()` is not allowed because at runtime, it would be `new Object()`.
- **Create an array of that generic type:** This one is the most annoying, but it makes sense because you'd be creating an array of `Object` values.
- **Call `instanceof`:** This is not allowed because at runtime `List<Integer>` and `List<String>` look the same to Java, thanks to type erasure.
- **Use a primitive type as a generic type parameter:** This isn't a big deal because you can use the wrapper class instead. If you want a type of `int`, just use `Integer`.
- **Create a `static` variable as a generic type parameter:** This is not allowed because the type is linked to the instance of the class.
- **Catch an exception of type T**: Even if `T` extends `Exception`, it cannot be used in a catch block since the precise type is not known.

Writing Generic Methods

Up until this point, you've seen formal type parameters declared on the class or interface level. It is also possible to declare them on the method level. This is often useful for `static` methods since they aren't part of an instance that can declare the type. However, it is also allowed on non-`static` methods.

In this example, both methods use a generic parameter:

```
public class Handler {
   public static <T> void prepare(T t) {
      System.out.println("Preparing " + t);
   }
   public static <T> Crate<T> ship(T t) {
      System.out.println("Shipping " + t);
      return new Crate<T>();
   }
}
```

The method parameter is the generic type `T`. Before the return type, we declare the formal type parameter of `<T>`. In the `ship()` method, we show how you can use the generic parameter in the return type, `Crate<T>`, for the method.

Unless a method is obtaining the generic formal type parameter from the class/interface, it is specified immediately before the return type of the method. This can lead to some interesting-looking code!

```
2: public class More {
3:    public static <T> void sink(T t) { }
4:    public static <T> T identity(T t) { return t; }
5:    public static T noGood(T t) { return t; } // DOES NOT COMPILE
6: }
```

Line 3 shows the formal parameter type immediately before the return type of `void`. Line 4 shows the return type being the formal parameter type. It looks weird, but it is correct. Line 5 omits the formal parameter type and therefore does not compile.

Real World Scenario

Optional Syntax for Invoking a Generic Method

You can call a generic method normally, and the compiler will try to figure out which one you want. Alternatively, you can specify the type explicitly to make it obvious what the type is.

```
Box.<String>ship("package");
Box.<String[]>ship(args);
```

It is up to you whether this makes things clearer. You should at least be aware that this syntax exists.

When you have a method declare a generic parameter type, it is independent of the class generics. Take a look at this class that declares a generic `T` at both levels:

```
1: public class TrickyCrate<T> {
2:   public <T> T tricky(T t) {
3:      return t;
4:    }
5: }
```

See if you can figure out the type of `T` on lines 1 and 2 when we call the code as follows:

```
10: public static String crateName() {
11:    TrickyCrate<Robot> crate = new TrickyCrate<>();
12:    return crate.tricky("bot");
13: }
```

Clearly, "T is for tricky." Let's see what is happening. On line 1, `T` is `Robot` because that is what gets referenced when constructing a `Crate`. On line 2, `T` is `String` because that is what is passed to the method. When you see code like this, take a deep breath and write down what is happening so you don't get confused.

Creating a Generic Record

Generics can also be used with records. This record takes a single generic type parameter:

```
public record CrateRecord<T>(T contents) {
   @Override
   public T contents() {
      if (contents == null)
         throw new IllegalStateException("missing contents");
      return contents;
   }
}
```

This works the same way as classes. You can create a record of the robot!

```
Robot robot = new Robot();
CrateRecord<Robot> record = new CrateRecord<>(robot);
```

This is convenient. Now we have an immutable, generic record!

Bounding Generic Types

By now, you might have noticed that generics don't seem particularly useful since they are treated as `Object`s and, therefore, don't have many methods available. Bounded wildcards solve this by restricting what types can be used in a generic. A *bounded parameter type* is a generic type that specifies a bound for the generic. Be

warned that this is the hardest section in the chapter, so don't feel bad if you have to read it more than once.

A *wildcard generic type* is an unknown generic type represented with a question mark (`?`). You can use generic wildcards in three ways, as shown in Table 9.15. This section looks at each of these three wildcard types.

TABLE 9.15 Types of bounds

Type of bound	Syntax	Example
Unbounded wildcard	`?`	`List<?> a = new ArrayList<String>();`
Wildcard with upper bound	`? extends type`	`List<? extends Exception> a = new ArrayList<RuntimeException>();`
Wildcard with lower bound	`? super type`	`List<? super Exception> a = new ArrayList<Object>();`

Creating Unbounded Wildcards

An unbounded wildcard represents any data type. You use `?` when you want to specify that any type is OK with you. Let's suppose that we want to write a method that looks through a list of any type.

```
public static void printList(List<Object> list) {
   for (Object x: list)
      System.out.println(x);
}
public static void main(String[] args) {
   List<String> keywords = new ArrayList<>();
   keywords.add("java");
   printList(keywords); // DOES NOT COMPILE
}
```

Wait. What's wrong? A `String` is a subclass of an `Object`. This is true. However, `List<String>` cannot be assigned to `List<Object>`. We know, it doesn't sound logical. Java is trying to protect us from ourselves with this one. Imagine if we could write code like this:

```
4: List<Integer> numbers = new ArrayList<>();
5: numbers.add(Integer.valueOf(42));
6: List<Object> objects = numbers; // DOES NOT COMPILE
7: objects.add("forty two");
8: System.out.println(numbers.get(1));
```

On line 4, the compiler promises us that only `Integer` objects will appear in `numbers`. If line 6 compiled, line 7 would break that promise by putting a `String` in

there since `numbers` and `objects` are references to the same object. Good thing the compiler prevents this.

Going back to printing a list, we cannot assign a `List<String>` to a `List<Object>`. That's fine; we don't need a `List<Object>`. What we really need is a `List` of "whatever." That's what `List<?>` is. The following code does what we expect:

```
public static void printList(List<?> list) {
   for (Object x: list)
      System.out.println(x);
}
public static void main(String[] args) {
   List<String> keywords = new ArrayList<>();
   keywords.add("java");
   printList(keywords);
}
```

The `printList()` method takes any type of list as a parameter. The `keywords` variable is of type `List<String>`. We have a match! `List<String>` is a list of anything. "Anything" just happens to be a `String` here.

Finally, let's look at the impact of `var`. Do you think these two statements are equivalent?

```
List<?> x1 = new ArrayList<>();
var x2 = new ArrayList<>();
```

They are not. There are two key differences. First, `x1` is of type `List`, while `x2` is of type `ArrayList`. Additionally, we can only assign `x2` to a `List<Object>`. These two variables do have one thing in common. Both return type `Object` when calling the `get()` method.

Creating Upper-Bounded Wildcards

Let's try to write a method that adds up the total of a list of numbers. We've established that a generic type can't just use a subclass.

```
ArrayList<Number> list = new ArrayList<Integer>(); // DOES NOT COMPILE
```

Instead, we need to use a wildcard:

```
List<? extends Number> list = new ArrayList<Integer>();
```

The upper-bounded wildcard says that any class that `extends Number` or `Number` itself can be used as the formal parameter type:

```
public static long total(List<? extends Number> list) {
   long count = 0;
   for (Number number: list)
```

```
      count += number.longValue();
   return count;
}
```

Remember how we kept saying that type erasure makes Java think that a generic type is an `Object`? That is still happening here. Java converts the previous code to something equivalent to the following:

```
public static long total(List list) {
   long count = 0;
   for (Object obj: list) {
      Number number = (Number) obj;
      count += number.longValue();
   }
   return count;
}
```

Something interesting happens when we work with upper bounds or unbounded wildcards. The list becomes logically immutable and therefore cannot be modified. Technically, you can remove elements from the list, but the exam won't ask about this.

```
2: static class Sparrow extends Bird { }
3: static class Bird { }
4:
5: public static void main(String[] args) {
6:    List<? extends Bird> birds = new ArrayList<Bird>();
7:    birds.add(new Sparrow()); // DOES NOT COMPILE
8:    birds.add(new Bird());    // DOES NOT COMPILE
9: }
```

The problem stems from the fact that Java doesn't know what type `List<? extends Bird>` really is. It could be `List<Bird>` or `List<Sparrow>` or some other generic type that hasn't even been written yet. Line 7 doesn't compile because we can't add a `Sparrow` to `List<? extends Bird>`, and line 8 doesn't compile because we can't add a `Bird` to `List<Sparrow>`. From Java's point of view, both scenarios are equally possible, so neither is allowed.

Now let's try an example with an interface. We have an interface and two classes that implement it.

```
interface Flyer { void fly(); }
class HangGlider implements Flyer { public void fly() {} }
class Goose implements Flyer { public void fly() {} }
```

We also have two methods that use it. One just lists the interface, and the other uses an upper bound.

```
private void anyFlyer(List<Flyer> flyer) {}
private void groupOfFlyers(List<? extends Flyer> flyer) {}
```

Note that we used the keyword `extends` rather than `implements`. Upper bounds are like anonymous classes in that they use `extends` regardless of whether we are working with a class or an interface.

You already learned that a variable of type `List<Flyer>` can be passed to either method. A variable of type `List<Goose>` can be passed only to the one with the upper bound. This shows a benefit of generics. Random flyers don't fly together. We want our `groupOfFlyers()` method to be called only with the same type. Geese fly together but don't fly with hang gliders.

Creating Lower-Bounded Wildcards

Let's try to write a method that adds a string `"quack"` to two lists.

```
List<String> strings = new ArrayList<String>();
strings.add("tweet");

List<Object> objects = new ArrayList<Object>(strings);
addSound(strings);
addSound(objects);
```

The problem is that we want to pass a `List<String>` and a `List<Object>` to the same method. First, make sure you understand why the first three examples in Table 9.16 do *not* solve this problem.

TABLE 9.16 Why we need a lower bound

`static void addSound( list) { list.add("quack"); }`	**Method compiles**	**Can pass a `List<String>`**	**Can pass a `List<Object>`**
`List<?>`	No	Yes	Yes
`List<? extends Object>`	No	Yes	Yes
`List<Object>`	Yes	No (with generics, must pass exact match)	Yes
`List<? super String>`	Yes	Yes	Yes

To solve this problem, we need to use a lower bound.

```
public static void addSound(List<? super String> list) {
   list.add("quack");
}
```

With a lower bound, we are telling Java that the list will be a list of `String` objects or a list of some objects that are a superclass of `String`. Either way, it is safe to add

a `String` to that list.

Just like generic classes, you probably won't use this in your code unless you are writing code for others to reuse. Even then, it would be rare. But it's on the exam, so now is the time to learn it!

Understanding Generic Supertypes

When you have subclasses and superclasses, lower bounds can get tricky.

```
    3: List<? super IOException> exceptions = new
ArrayList<Exception>();
    4: exceptions.add(new Exception()); // DOES NOT COMPILE
    5: exceptions.add(new IOException());
    6: exceptions.add(new FileNotFoundException());
```

Line 3 references a `List` that could be `List<IOException>` or `List<Exception>` or `List<Object>`. Line 4 does not compile because we could have a `List<IOException>`, and an `Exception` object wouldn't fit in there.

Line 5 is fine. `IOException` can be added to any of those types. Line 6 is also fine. `FileNotFoundException` can also be added to any of those three types. This is tricky because `FileNotFoundException` is a subclass of `IOException`, and the keyword says `super`. Java says, "Well, `FileNotFoundException` also happens to be an `IOException`, so everything is fine."

Putting It All Together

At this point, you know everything that you need to know to ace the exam questions on generics. It is possible to put these concepts together to write some *really* confusing code, which the exam likes to do.

This section is going to be difficult to read. It contains the hardest questions that you could possibly be asked about generics. The exam questions will probably be easier to read than these. We want you to encounter the really tough ones here so that you are ready for the exam. In other words, don't panic. Take it slow, and reread the code a few times. You'll get it.

Combining Generic Declarations

Let's try an example. First, we declare three classes that the example will use:

```
class A {}
class B extends A {}
```

```
class C extends B {}
```

Ready? Can you figure out why these do or don't compile? Also, try to figure out what they do.

```
6: List<?> list1 = new ArrayList<A>();
7: List<? extends A> list2 = new ArrayList<A>();
8: List<? super A> list3 = new ArrayList<A>();
```

Line 6 creates an `ArrayList` that can hold instances of class `A`. It is stored in a variable with an unbounded wildcard. Any generic type can be referenced from an unbounded wildcard, making this OK.

Line 7 tries to store a list in a variable declaration with an upper-bounded wildcard. This is OK. You can have `ArrayList<A>`, `ArrayList<B>`, or `ArrayList<C>` stored in that reference. Line 8 is also OK. This time, you have a lower-bounded wildcard. The lowest type you can reference is `A`. Since that is what you have, it compiles.

Did you get those right? Let's try a few more.

```
9:  List<? extends B> list4 = new ArrayList<A>(); // DOES NOT COMPILE
10: List<? super B> list5 = new ArrayList<A>();
11: List<?> list6 = new ArrayList<? extends A>(); // DOES NOT COMPILE
```

Line 9 has an upper-bounded wildcard that allows `ArrayList<B>` or `ArrayList<C>` to be referenced. Since you have `ArrayList<A>` that is trying to be referenced, the code does not compile. Line 10 has a lower-bounded wildcard, which allows a reference to `ArrayList<A>`, `ArrayList<A>`, or `ArrayList<Object>`.

Finally, line 11 allows a reference to any generic type since it is an unbounded wildcard. The problem is that you need to know what that type will be when instantiating the `ArrayList`. It wouldn't be useful anyway, because you can't add any elements to that `ArrayList`.

Passing Generic Arguments

Now on to the methods. Same question: try to figure out why they don't compile or what they do. We will present the methods one at a time because there is more to think about.

```
<T> T first(List<? extends T> list) {
   return list.get(0);
}
```

The first method, `first()`, is a perfectly normal use of generics. It uses a method-specific type parameter, `T`. It takes a parameter of `List<T>`, or some subclass of `T`, and it returns a single object of that `T` type. For example, you could call it with a

`List<String>` parameter and have it return a `String`. Or you could call it with a `List<Number>` parameter and have it return a `Number`. Or—well, you get the idea.

Given that, you should be able to see what is wrong with this one:

```
<T> <? extends T> second(List<? extends T> list) { // DOES NOT COMPILE
   return list.get(0);
}
```

The next method, `second()`, does not compile because the return type isn't actually a type. You are writing the method. You know what type it is supposed to return. You don't get to specify this as a wildcard.

Now be careful—this one is extra tricky:

```
<B extends A> B third(List<B> list) {
   return new B(); // DOES NOT COMPILE
}
```

This method, `third()`, does not compile. `<B extends A>` says that you want to use `B` as a type parameter just for this method and that it needs to extend the `A` class. Coincidentally, `B` is also the name of a class. Well, it isn't a coincidence. It's an evil trick. Within the scope of the method, B can represent class `A`, `B`, or `C`, because all extend the `A` class. Since B no longer refers to the `B` class in the method, you can't instantiate it.

After that, it would be nice to get something straightforward.

```
void fourth(List<? super B> list) {}
```

We finally get a method, `fourth()`, that is a normal use of generics. You can pass the type `List<B>`, `List<A>`, or `List<Object>`.

Finally, can you figure out why this example does not compile?

```
<X> void fifth(List<X super B> list) { } // DOES NOT COMPILE
```

This last method, `fifth()`, does not compile because it tries to mix a method-specific type parameter with a wildcard. A wildcard must have a `?` in it.

Phew. You made it through generics. It's the hardest topic in this chapter (and why we covered it last!). Remember that it's OK if you need to go over this material a few times to get your head around it.

Summary

The Java Collections Framework includes four main types of data structures: lists, sets, queues, and maps. The `Collection` interface is the parent interface of `List`,

`Set`, and `Queue`. Additionally, `Deque` extends `Queue`. The `Map` interface does not extend `Collection`. You need to recognize the following:

- **`List`:** An ordered collection of elements that allows duplicate entries.
  - **`ArrayList`:** Standard resizable list.
  - **`LinkedList`:** Can easily add/remove from beginning or end.
- **`Set`**: Does not allow duplicates.
  - **`HashSet`:** Uses `hashCode()` to find unordered elements.
  - **`LinkedHashSet`:** Well-defined encounter order.
  - **`TreeSet`:** Sorted. Does not allow `null` values.
- **`Queue/Deque`:** Orders elements for processing.
  - **`ArrayDeque`:** Double-ended queue.
  - **`LinkedList`:** Double-ended queue and list.
- **`Map`:** Maps unique keys to values.
  - **`HashMap`:** Uses `hashCode()` to find keys.
  - **`LinkedHashMap`:** Well-defined encounter order.
  - **`TreeMap`:** Sorted map. Does not allow `null` keys.

Java 21 now includes sequenced collections, for types with a defined encounter order.

- **`SequencedCollection`:** `ArrayDeque`, `ArrayList`, `LinkedList`, LinkedHashSet, and `TreeSet`.
- **`SequencedSet`:** `LinkedHashSet` and `TreeSet`.
- **`SequencedMap`:** `LinkedHashMap` and `TreeMap`.

The `Comparable` interface declares the `compareTo()` method. This method returns a negative number if the object is smaller than its argument, `0` if the two objects are equal, and a positive number otherwise. The `compareTo()` method is declared on the object that is being compared, and it takes one parameter. The `Comparator` interface defines the `compare()` method. A negative number is returned if the first argument is smaller, zero if they are equal, and a positive number otherwise. The `compare()` method can be declared in any code, and it takes two parameters. A `Comparator` is often implemented using a lambda.

Generics are type parameters for code. To create a class with a generic parameter, add `<T>` after the class name. You can use any name you want for the type parameter. Single uppercase letters are common choices. Generics allow you to specify wildcards. `<?>` is an unbounded wildcard that means any type. `<? extends Object>` is an upper bound that means any type that is `Object` or extends it. `<? extends MyInterface>` means any type that implements `MyInterface`. `<? super Number>` is a lower bound that means any type that is `Number` or a superclass. A compiler error results from code that attempts to add an item in a list with an unbounded or upper-bounded wildcard.

Exam Essentials

Pick the correct type of collection from a description. A `List` allows duplicates and orders the elements. A `Set` does not allow duplicates. A `Deque` orders its elements to facilitate retrievals from the front or back. A `Map` maps keys to values. Be familiar with the differences in implementations of these interfaces.

Mork with convenience methods. The Collections Framework contains many methods such as `contains()`, `forEach()`, and `removeIf()` that you need to know for the exam. There are too many to list in this paragraph for review, so please do review the tables in this chapter.

Understand how to use sequenced collections. Other than `HashSet` and `HashMap`, most collection classes now implement a sequenced collection interface (`SequencedCollection` or `SequencedMap` or `SequencedSet`). This includes methods that support iterating over the collection in a predictable encounter order. The methods make it easier to work with related types using a consistent interface.

Differentiate between *`Comparable`* and *`Comparator`*. Classes that implement `Comparable` are said to have a natural ordering and implement the `compareTo()` method. A class is allowed to have only one natural ordering. A `Comparator` takes two objects in the `compare()` method. Different ones can have different sort orders. A `Comparator` is often implemented using a lambda such as `(a, b) -> a.num - b.num`.

Identify valid and invalid uses of generics and wildcards. `<T>` represents a type parameter. Any name can be used, but a single uppercase letter is the convention. `<?>` is an unbounded wildcard. `<? extends X>` is an upper-bounded wildcard. `<? super X>` is a lower-bounded wildcard.