# Chapter 14 Generics and Collections

**THE OCP EXAM TOPICS COVERED IN THIS CHAPTER INCLUDE THE FOLLOWING:**

- **Generics and Collections**
  - Use wrapper classes, autoboxing and autounboxing
  - Create and use generic classes, methods with diamond notation and wildcards
  - Describe the Collections Framework and use key collection interfaces
  - Use Comparator and Comparable interfaces
  - Create and use convenience methods for collections
- **Java Stream API**
  - Use lambda expressions and method references

You learned the basics of the Java Collections Framework in Chapter 5, "Core Java APIs," along with generics and the basics of sorting. We will review these topics while diving into them more deeply.

First, we will cover how to use method references. After a review of autoboxing and unboxing, we will explore more classes and APIs in the Java Collections Framework. The thread-safe collection types will be discussed in Chapter 18, "Concurrency."

Next, we will cover details about `Comparator` and `Comparable`. Finally, we will discuss how to create your own classes and methods that use generics so that the same class can be used with many types.

As you may remember from Chapter 6, "Lambdas and Functional Interfaces," we presented functional interfaces like Predicate, `Consumer`, `Function`, and `Supplier`. We will review all of these functional interfaces in Chapter 15, "Functional Programming," but since some will be used in this chapter, we provide Table 14.1 as a handy reference. The letters (`R`, `T`, and `U`) are generics that you can pass any type to when using these functional interfaces.

**TABLE 14.1** Functional interfaces used in this chapter

| Functional interfaces | Return type | Method name | # parameters |
|---|---|---|---|
| `Supplier<T>` | `T` | `get()` | 0 |
| `Consumer<T>` | `void` | `accept(T)` | 1 (`T`) |
| `BiConsumer<T, U>` | `void` | `accept(T,U)` | 2 (`T`, `U`) |
| `Predicate<T>` | `boolean` | `test(T)` | 1 (`T`) |
| `BiPredicate<T, U>` | `boolean` | `test(T, U)` | 2 (`T`, `U`) |
| `Function<T, R>` | `R` | `apply(T)` | 1 (`T`) |
| `BiFunction<T, U, R>` | `R` | `apply(T,U)` | 2 (`T`, `U`) |
| `UnaryOperator<T>` | `T` | `apply(T)` | 1 (`T`) |

For this chapter, you can just use these functional interfaces as is. In the next chapter, though, we'll be presenting and testing your knowledge of them.

## Using Method References

In Chapter 12, we went over lambdas and showed how they make code shorter. *Method references* are another way to make the code easier to read, such as simply mentioning the name of the method. Like lambdas, it takes time to get used to the new syntax.

In this section, we will show the syntax along with the four types of method references. We will also mix in lambdas with method references. If you'd like to review using lambdas, see Chapter 12. This will prepare you well for the next chapter, which uses both heavily.

Suppose we are coding a duckling who is trying to learn how to quack. First, we have a functional interface. As you'll recall from Chapter 12, this is an interface with exactly one abstract method.

```
@FunctionalInterface
public interface LearnToSpeak {
   void speak(String sound);
}
```

Next, we discover that our duckling is lucky. There is a helper class that the duckling can work with. We've omitted the details of teaching the duckling how to quack and left the part that calls the functional interface.

```
public class DuckHelper {
   public static void teacher(String name, LearnToSpeak trainer) {

      // exercise patience

      trainer.speak(name);
   }
}
```

Finally, it is time to put it all together and meet our little `Duckling`. This code implements the functional interface using a lambda:

```
public class Duckling {
   public static void makeSound(String sound) {
      LearnToSpeak learner = s -> System.out.println(s);
```

```
            DuckHelper.teacher(sound, learner);
        }
    }
```

Not bad. There's a bit of redundancy, though. The lambda declares one parameter named `s`. However, it does nothing other than pass that parameter to another method. A method reference lets us remove that redundancy and instead write this:

```
LearnToSpeak learner = System.out::println;
```

The `::` operator tells Java to call the `println()` method later. It will take a little while to get used to the syntax. Once you do, you may find your code is shorter and less distracting without writing as many lambdas.

---


Remember that `::` is like a lambda, and it is used for deferred execution with a functional interface.

---

A method reference and a lambda behave the same way at runtime. You can pretend the compiler turns your method references into lambdas for you.

There are four formats for method references:

- Static methods
- Instance methods on a particular instance
- Instance methods on a parameter to be determined at runtime
- Constructors

Let's take a brief look at each of these in turn. In each example, we will show the method reference and its lambda equivalent. We are going to use some built-in functional interfaces in these examples. We will remind

you what they do right before each example. In Chapter 15, we will cover many such interfaces.

### Calling Static Methods

The `Collections` class has a `static` method that can be used for sorting. Per Table 14.1, the `Consumer` functional interface takes one parameter and does not return anything. Here we will assign a method reference and a lambda to this functional interface:

```
14: Consumer<List<Integer>> methodRef = Collections::sort;
15: Consumer<List<Integer>> lambda = x -> Collections.sort(x);
```

On line 14, we reference a method with one parameter, and Java knows that it's like a lambda with one parameter. Additionally, Java knows to pass that parameter to the method.

Wait a minute. You might be aware that the `sort()` method is overloaded. How does Java know that we want to call the version with only one parameter? With both lambdas and method references, Java is inferring information from the context. In this case, we said that we were declaring a `Consumer`, which takes only one parameter. Java looks for a method that matches that description. If it can't find it or it finds multiple ones that could match multiple methods, then the compiler will report an error. The latter is sometimes called an *ambiguous* type error.

### Calling Instance Methods on a Particular Object

The `String` class has a `startsWith()` method that takes one parameter and returns a `boolean`. Conveniently, a `Predicate` is a functional interface that takes one parameter and returns a `boolean`. Let's look at how to use method references with this code:

```
18: var str = "abc";
19: Predicate<String> methodRef = str::startsWith;
20: Predicate<String> lambda = s -> str.startsWith(s);
```

Line 19 shows that we want to call `str.startsWith()` and pass a single parameter to be supplied at runtime. This would be a nice way of filtering the data in a list. In fact, we will do that later in the chapter.

A method reference doesn't have to take any parameters. In this example, we use a `Supplier`, which takes zero parameters and returns a value:

```
var random = new Random();
Supplier<Integer> methodRef = random::nextInt;
Supplier<Integer> lambda = () -> random.nextInt();
```

Since the methods on `Random` are instance methods, we call the method reference on an instance of the `Random` class.

### Calling Instance Methods on a Parameter

This time, we are going to call an instance method that doesn't take any parameters. The trick is that we will do so without knowing the instance in advance.

```
23: Predicate<String> methodRef = String::isEmpty;
24: Predicate<String> lambda = s -> s.isEmpty();
```

Line 23 says the method that we want to call is declared in `String`. It looks like a `static` method, but it isn't. Instead, Java knows that `isEmpty()` is an instance method that does not take any parameters. Java uses the parameter supplied at runtime as the instance on which the method is called.

Compare lines 23 and 24 with lines 19 and 20 of our instance example. They look similar, although one references a local variable named `str`, while the other only references the functional interface parameters.

You can even combine the two types of instance method references. We are going to use a functional interface called a `BiPredicate`, which takes two parameters and returns a `boolean`.

```
26: BiPredicate<String, String> methodRef = String::startsWith;
27: BiPredicate<String, String> lambda = (s, p) -> s.startsWith(p);
```

Since the functional interface takes two parameters, Java has to figure out what they represent. The first one will always be the instance of the object for instance methods. Any others are to be method parameters.

Remember that line 26 may look like a `static` method, but it is really a method reference declaring that the instance of the object will be specified later. Line 27 shows some of the power of a method reference. We were able to replace two lambda parameters this time.

### Calling Constructors

A *constructor reference* is a special type of method reference that uses `new` instead of a method, and it instantiates an object. It is common for a constructor reference to use a `Supplier` as shown here:

```
30: Supplier<List<String>> methodRef = ArrayList::new;
31: Supplier<List<String>> lambda = () -> new ArrayList();
```

It expands like the method references you have seen so far. In the previous example, the lambda doesn't have any parameters.

Method references can be tricky. In our next example, we will use the `Function` functional interface, which takes one parameter and returns a result. Notice that line 32 in the following example has the same method reference as line 30 in the previous example:

```
32: Function<Integer, List<String>> methodRef = ArrayList::new;
33: Function<Integer, List<String>> lambda = x -> new ArrayList(x);
```

This means you can't always determine which method can be called by looking at the method reference. Instead, you have to look at the context to see what parameters are used and if there is a return type. In this example, Java sees that we are passing an `Integer` parameter and calls the constructor of `ArrayList` that takes a parameter.

### Reviewing Method References

Reading method references is helpful in understanding the code. Table 14.2 shows the four types of method references. If this table doesn't make sense, please reread the previous section. It can take a few tries before method references start to make sense.

**TABLE 14.2** Method references

| Type | Before colon | After colon | Example |
|---|---|---|---|
| Static methods | Class name | Method name | `Collections::sort` |
| Instance methods on a particular object | Instance variable name | Method name | `str::startsWith` |
| Instance methods on a parameter | Class name | Method name | `String::isEmpty` |
| Constructor | Class name | `new` | `ArrayList::new` |

NUMBER OF PARAMETERS IN A METHOD REFERENCE

We mentioned that a method reference can look the same even when it will behave differently based on the surrounding context. For example, given the following method:

```
public class Penguin {
   public static Integer countBabies(Penguin… cuties) {
      return cuties.length;
   }
}
```

we show three ways that `Penguin::countBabies` can be interpreted. This method allows you to pass zero or more values and creates an array with those values.

```
10: Supplier<Integer> methodRef1 = Penguin::countBabies;
11: Supplier<Integer> lambda1 = () -> Penguin.countBabies();
12:
13: Function<Penguin, Integer> methodRef2 = Penguin::countBabies;
14: Function<Penguin, Integer> lambda2 = (x) -> Penguin.countBabies(x);
15:
16: BiFunction<Penguin, Penguin, Integer> methodRef3 = Penguin::countBabi
17: BiFunction<Penguin, Penguin, Integer> lambda3 =
18:    (x, y) -> Penguin.countBabies(x, y);
```

Lines 10 and 11 do not take any parameters because the functional interface is a `Supplier`. Lines 13 and 14 take one parameter. Lines 16 and 17 take two parameters. All six lines return an `Integer` from the method reference or lambda.

There's nothing special about zero, one, and two parameters. If we had a functional interface with 100 parameters of type `Penguin` and the final one of `Integer`, we could still implement it with `Penguin::countBabies`.

## Using Wrapper Classes

As you read in Chapter 5, each Java primitive has a corresponding wrapper class, shown in Table 14.3. With *autoboxing*, the compiler automatically converts a primitive to the corresponding wrapper. Unsurprisingly, *unboxing* is the process in which the compiler automatically converts a wrapper class back to a primitive.

**TABLE 14.3** Wrapper classes

| Primitive type | Wrapper class | Example of initializing |
|---|---|---|
| `boolean` | `Boolean` | `Boolean.valueOf(true)` |
| `byte` | `Byte` | `Byte.valueOf((byte) 1)` |
| `short` | `Short` | `Short.valueOf((short) 1)` |
| `int` | `Integer` | `Integer.valueOf(1)` |
| `long` | `Long` | `Long.valueOf(1)` |
| `float` | `Float` | `Float.valueOf((float) 1.0)` |
| `double` | `Double` | `Double.valueOf(1.0)` |
| `char` | `Character` | `Character.valueOf('c')` |

Can you spot the autoboxing and unboxing in this example?

```
12: Integer pounds = 120;
13: Character letter = "robot".charAt(0);
14: char r = letter;
```

Line 12 is an example of autoboxing as the `int` primitive is autoboxed into an `Integer` object. Line 13 demonstrates that autoboxing can involve methods. The `charAt()` method returns a primitive `char`. It is then autoboxed into the wrapper object `Character`. Finally, line 14

shows an example of unboxing. The `Character` object is unboxed into a primitive `char`.

There are two tricks in the space of autoboxing and unboxing. The first has to do with `null` values. This innocuous-looking code throws an exception:

```
15: var heights = new ArrayList<Integer>();
16: heights.add(null);
17: int h = heights.get(0); // NullPointerException
```

On line 16, we add a `null` to the list. This is legal because a `null` reference can be assigned to any reference variable. On line 17, 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.

---

**WRAPPER CLASSES AND *NULL***

Speaking of `null`, one advantage of a wrapper class over a primitive is that it can hold a `null` value. While `null` values aren't particularly useful for numeric calculations, they are quite useful in data-based services. For example, if you are storing a user's location data using (latitude, longitude), it would be a bad idea to store a missing point as (0,0) since that refers to an actual location off the cost of Africa where the user could theoretically be.

---

Also, be careful when autoboxing into `Integer`. What do you think this code outputs?

```
23: List<Integer> numbers = new ArrayList<Integer>();
24: numbers.add(1);
25: numbers.add(Integer.valueOf(3));
26: numbers.add(Integer.valueOf(5));
27: numbers.remove(1);
```

```
28: numbers.remove(Integer.valueOf(5));
29: System.out.println(numbers);
```

It actually outputs `[1]`. Let's walk through why that is. On lines 24 through 26, we add three `Integer` objects to `numbers`. The one on line 24 relies on autoboxing to do so, but it gets added just fine. At this point, `numbers` contains `[1, 3, 5]`.

Line 27 contains the second trick. The `remove()` method is overloaded. One signature takes an `int` as the index of the element to remove. The other takes an `Object` that should be removed. On line 27, Java sees a matching signature for `int`, so it doesn't need to autobox the call to the method. Now `numbers` contains `[1, 5]`. Line 28 calls the other `remove()` method, and it removes the matching object, which leaves us with just `[1]`.

## Using the Diamond Operator

In the past, we would write code using generics like the following:

```
List<Integer> list = new ArrayList<Integer>();
Map<String,Integer> map = new HashMap<String,Integer>();
```

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

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

That's a lot of duplicate code to write! We'll cover expressions later in this chapter where the generic types might not be the same, but often the generic types for both sides of the expression are identical.

Luckily, the diamond operator, `<>`, was added to the language. 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<String,Integer> map = new HashMap<>();
Map<Long,List<Integer>> mapOfLists = new HashMap<>();
```

The first is the variable declaration and fully specifies the generic type. The second is an expression that infers the type from the assignment operator, using the diamond operator. To the compiler, both these declarations and our previous ones are equivalent. To us, though, the latter is a lot shorter and easier to read.

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, none of the following compiles:

```
List<> list = new ArrayList<Integer>();      // DOES NOT COMPILE
Map<> map = new HashMap<String, Integer>();  // DOES NOT COMPILE
class InvalidUse {
   void use(List<> data) {}                  // DOES NOT COMPILE
}
```

Since `var` is new to Java, let's look at the impact of using `var` with the diamond operator. Do you think these two statements compile and are equivalent?

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

While they both compile, they are not equivalent. The first one creates an `ArrayList<Integer>` just like the prior set of examples. The second one creates an `ArrayList<Object>`. Since there is no generic type specified, it cannot be inferred. Java happily assumes you wanted `Object` in this scenario.

## Using Lists, Sets, Maps, and Queues

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 collec-

tions. 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 typical queue processes its elements in a first-in, first-out order, but other orderings are possible.
- `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 14.1 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.

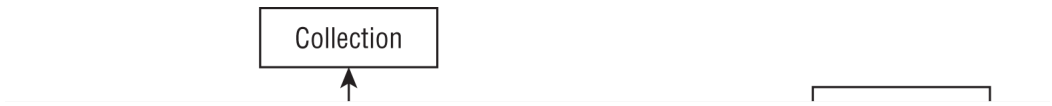



**FIGURE 14.1** The `Collection` interface is the root of all collections except maps.

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 why maps are treated differently is that they need different methods due to being key/value pairs.

We will first discuss the methods `Collection` provides to all implementing classes. Then we will cover the different types of collections, including when to use each one and the concrete subclasses. Then we will compare the different types.

### Common Collections Methods

The `Collection` interface contains useful methods for working with lists, sets, and queues. In the following sections, we will discuss the most common ones. We will cover streams in the next chapter. 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'll cover the specific properties of each `Collection` class in the next section.

### *add()*

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

```
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.

### *remove()*

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

```
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.

Since calling `remove()` on a `List` with an `int` uses the index, an index that doesn't exist will throw an exception. For example, `birds.remove(100);` throws an `IndexOutOfBoundsException`. Remember that there are overloaded `remove()` methods. One takes the element to remove. The other takes the index of the element to remove. The latter is being called here.

---

**DELETING WHILE LOOPING**

Java does not allow removing elements from a list while using the enhanced `for` loop.

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

for (String bird : birds) // ConcurrentModificationException
   birds.remove(bird);
```

Wait a minute. Concurrent modification? We don't get to concurrency until Chapter 18. That's right. It is possible to get a `ConcurrentModificationException` without threads. This is Java's way of complaining that you are trying to modify the list while looping through it. In Chapter 18, we'll return to this example and show how to fix it with the `CopyOnWriteArrayList` class.

---

### *isEmpty() and size()*

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

```
boolean isEmpty()
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.

### *clear()*

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

```
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`.

### *contains()*

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

```
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.

***removeIf()***

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 will explain what the `? super` means in the "Working with Generics" section later in this chapter.)

```
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.

How would you replace line 8 with a method reference? Trick question—you can't. The `removeIf()` method takes a `Predicate`. We can pass only one value with this method reference. Since `startsWith` takes a literal, it needs to be specified "the long way."

Let's try one more example that does use 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 the 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 the list.

#### *forEach()*

Looping through a `Collection` is common. On the 1Z0-815 exam, you wrote lots of loops. There's also a `forEach()` method that you can call on a `Collection`. It uses a `Consumer` that takes a single parameter and doesn't return anything. The method signature is as follows:

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

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

```
Collection<String> cats = Arrays.asList("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)!

### Using the List Interface

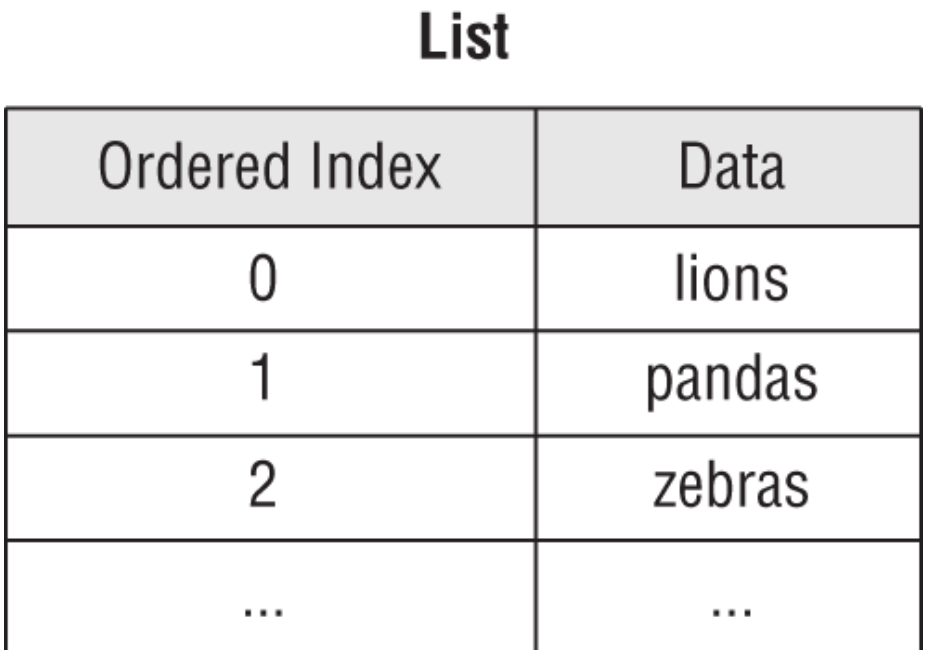
Now that you're familiar with some common `Collection` interface methods, let's move on to specific classes. You use a list when you want an ordered collection that can contain duplicate entries. Items can be retrieved and inserted at specific positions in the list based on an `int` index much

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 14.2 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 14.2** Example of a `List`

Sometimes, you don't actually 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 that all `List` implementations have in common is that they are ordered and allow duplicates. Beyond that, they each offer different functionality. We will 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

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 `Queue`. 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, 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 `Queue`. As you saw in Figure 14.1, a `LinkedList` implements both the `List` and `Queue` interface.

### 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 12, an immutable object cannot be changed or modified. Table 14.4 summarizes these three lists.

**TABLE 14.4** Factory methods to create a List

| Method | Description | Can add elements? | Can replace element? | 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]
29:
30: copy.add("y");  // throws UnsupportedOperationException
```

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 create an immutable `List`. Line 30 shows it is immutable by throwing an exception when trying to add a value. All three lists would throw an exception when adding or removing a value. The `of` and `copy` lists would also throw one on trying to update a reference.

### Working with *List* Methods

The methods in the `List` interface are for working with indexes. In addition to the inherited `Collection` methods, the method signatures that you need to know are in Table 14.5.

**TABLE 14.5** 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 |
| `E remove(int index)` | Removes element at index and moves the rest toward the front |
| `void replaceAll(UnaryOperator<E> op)` | Replaces each element in the list with the result of the operator |
| `E set(int index, E e)` | Replaces element at index and returns original. Throws `IndexOutOfBoundsException` if the index is larger than the maximum one set |

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 have succeeded.

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 have very large lists.

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

```
List<Integer> 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.

ITERATING THROUGH A LIST

There are many ways to iterate through a list. For example, in Chapter 4, "Making Decisions," you saw how to loop through a list using an enhanced `for` loop.

```
for (String string: list) {
   System.out.println(string);
}
```

You may see another approach used.

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

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.

## 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 14.3 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 will look at each implementation that you need to know for the exam and how to write code using `Set`.

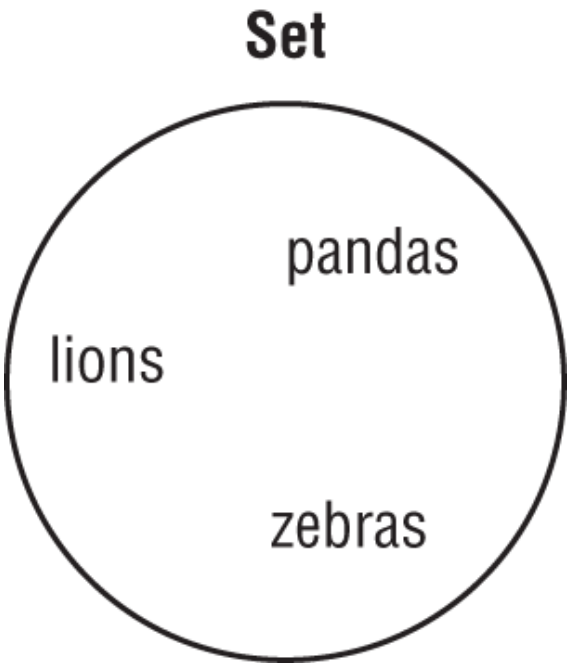



**FIGURE 14.3** Example of a `Set`

### Comparing *Set* Implementations

A `HashSet` stores its elements in a *hash table*, which means the keys are a hash and the values are an `Object`. This means that it uses the `hashCode()` method of the objects to retrieve them more efficiently.

The main benefit is that adding elements and checking whether an element is in the set both have constant time. The trade-off is that you lose the order in which you inserted the elements. Most of the time, you aren't concerned with this in a set anyway, making `HashSet` the most common set.

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 and checking whether an element exists take longer than with a `HashSet`, especially as the tree grows larger.

Figure 14.4 shows how you can envision `HashSet` and `TreeSet` being stored. `HashSet` is more complicated in reality, but this is fine for the purpose of the exam.

### Working with *Set* Methods

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

```
Set<Character> letters = Set.of('z', 'o', 'o');
Set<Character> copy = Set.copyOf(letters);
```

**HashSet**

| `hashCode()` Value | Data |
|---|---|
| –995544615 | pandas |
| ... | |
| –705903059 | zebras |
| ... | |
| 102978519 | lions |

**TreeSet**

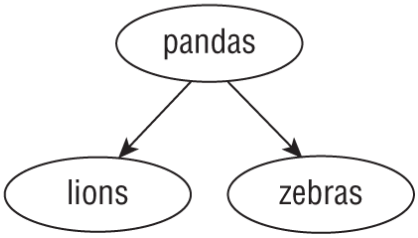



**FIGURE 14.4** Examples of a `HashSet` and `TreeSet`

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: set.forEach(System.out::println);
```

This code prints three lines:

```
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 al-

ready 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.

Now let's look at the same example with `TreeSet`.

```
3: Set<Integer> set = new TreeSet<>();
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: set.forEach(System.out::println);
```

This time, the code prints the following:

```
8
10
66
```

The elements are printed out in their natural sorted order. Numbers implement the `Comparable` interface in Java, which is used for sorting. Later in the chapter, you will learn how to create your own `Comparable` objects.

### Using the Queue Interface

You use a queue when elements are added and removed in a specific order. Queues are typically used for sorting elements prior to processing them. For example, when you want to buy a ticket 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!

Unless stated otherwise, a queue is assumed to be *FIFO* (first-in, first-out). Some queue implementations change this to use a different order. You can envision a FIFO queue as shown in Figure 14.5. The other format is *LIFO* (last-in, first-out), which is commonly referred to as a *stack*. In Java, though, both can be implemented with the `Queue` interface.


**FIGURE 14.5** Example of a `Queue`

Since this is a FIFO queue, Rover is first, which means he was the first one 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 will look at the implementations that you need to know and the available methods.

### Comparing *Queue* Implementations

You saw `LinkedList` earlier in the `List` section. In addition to being a list, it is a double-ended queue. A double-ended queue is different from a regular queue in that you can insert and remove elements from both the front and back of the queue. Think, "Mr. 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."

The main benefit of a `LinkedList` is that it implements both the `List` and `Queue` interfaces. The trade-off is that it isn't as efficient as a "pure" queue. You can use the `ArrayDeque` class (short for double-ended queue) if you need a more efficient queue. However, `ArrayDeque` is not in scope for the exam.

### Working with *Queue* Methods

The `Queue` interface contains many methods. Luckily, there are only six methods that you need to focus on. These methods are shown in Table 14.6.

**TABLE 14.6** Queue methods

| Method | Description | Throws exception on failure |
|---|---|---|
| `boolean add(E e)` | Adds an element to the back of the queue and returns `true` or throws an exception | Yes |
| `E element()` | Returns next element or throws an exception if empty queue | Yes |
| `boolean offer(E e)` | Adds an element to the back of the queue and returns whether successful | No |
| `E remove()` | Removes and returns next element or throws an exception if empty queue | Yes |
| `E poll()` | Removes and returns next element or returns `null` if empty queue | No |
| `E peek()` | Returns next element or returns `null` if empty queue | No |

As you can see, there are basically two sets of methods. One set throws an exception when something goes wrong. The other uses a different return value when something goes wrong. The `offer()` / `poll()` / `peek()` methods are more common. This is the standard language people use when working with queues.

Let's look at an example that uses some of these methods.

```
12: Queue<Integer> queue = new LinkedList<>();
13: System.out.println(queue.offer(10)); // true
14: System.out.println(queue.offer(4));  // true
15: System.out.println(queue.peek());    // 10
16: System.out.println(queue.poll());    // 10
17: System.out.println(queue.poll());    // 4
18: System.out.println(queue.peek());    // null
```

Figure 14.6 shows what the queue looks like at each step of the code. Lines 13 and 14 successfully add an element to the end of the queue. 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 actually remove the elements from the queue, which results in an empty queue. Line 18 tries to look at the first element of a queue, which results in `null`.

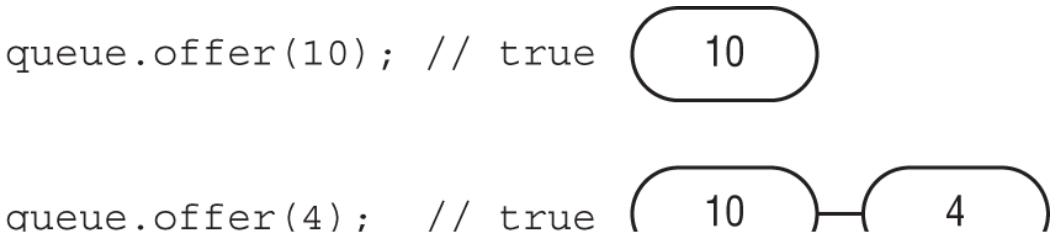



**FIGURE 14.6** Working with a queue

## 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 14.7. You don't need to know the names of the specific interfaces that the different maps implement, but you do need to know that `TreeMap` is sorted.

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

**FIGURE 14.7** Example of a `Map`

---

#### *MAP.OF() AND MAP.COPYOF()*

Just like `List` and `Set`, there is a helper method to create a `Map`. You pass any number of pairs of keys and values.

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

Unlike `List` and `Set`, this is less than ideal. Suppose you miscount and leave out a value.

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

This code compiles but throws an error at runtime. Passing keys and values is also 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("key1", "value1"));
```

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

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.

The main benefit is that adding elements and retrieving the element by key both have constant time. The trade-off is that you lose the order in which you inserted the elements. Most of the time, you aren't concerned with this in a map anyway. If you were, you could use `LinkedHashMap`, but that's not in scope for the exam.

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`, there are more methods 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 14.7. Some of the method signatures are simplified to make them easier to understand.

**TABLE 14.7** Map methods

| Method | Description |
|---|---|
| `void clear()` | Removes all keys and values from the 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 a `Set` of key/value pairs. |
| `void forEach(BiConsumer(K key, V value))` | Loop through each key/value pair. |
| `V get(Object key)` | Returns the value mapped by key or `null` if none is mapped. |
| `V getOrDefault(Object key, V defaultValue)` | Returns the value mapped by the key or the default value if none is mapped. |
| `boolean isEmpty()` | Returns whether the map is empty. |
| `Set<K> keySet()` | Returns set of all keys. |
| `V merge(K key, V value, Function(<V, V, V> func))` | Sets value if key not set. Runs the function if the key is set to determine the new value. Removes if `null`. |
| `V put(K key, V value)` | Adds or replaces key/value pair. Returns previous value or `null`. |

| Method | Description |
| --- | --- |
| `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 the value for a given key if the key is set. Returns the original value or `null` if none. |
| `void replaceAll(BiFunction<K, V, V> func)` | Replaces each value with the results of the function. |
| `int size()` | Returns the number of entries (key/value pairs) in the map. |
| `Collection<V> values()` | Returns `Collection` of all values. |

### Basic Methods

Let's start out by comparing the same code with two `Map` types. First up is `HashMap`.

```
Map<String, String> map = new HashMap<>();
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 + ","); // koala,giraffe,lion,
```

Here we set 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.

Java uses the `hashCode()` of the key to determine the order. The order here happens to not be sorted order, or the order in which we typed the values. Now let's look at `TreeMap`.

```
Map<String, String> map = new TreeMap<>();
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 + ","); // giraffe,koala,lion,
```

`TreeMap` sorts the keys as we would expect. If we were to have called `values()` instead of `keySet()`, the order of the values would correspond to the order of the keys.

With our same map, we can try some `boolean` checks.

```
System.out.println(map.contains("lion")); // DOES NOT COMPILE
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 line is a little tricky. The `contains()` method is on the `Collection` interface but not the `Map` interface. The next 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 there are zero elements and it is empty.

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

***forEach() and entrySet()***

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, if you 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 going through all the data in a map is to get the key/value pairs in a `Set`. Java has a `static` interface inside `Map` called `Entry`. It provides methods to get the key and value of each pair.

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

### *getOrDefault()*

The `get()` method returns null if the requested key is not in 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.

***replace() and replaceAll()***

These methods are similar to the `Collection` 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.

***putIfAbsent()***

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 added as well.

***merge()***

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 take 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 not longer than `Skyride` , so the map is updated. Line 21 prints out the new map contents. Lines 22 and 23 show that the result gets 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 14.8 shows all of these scenarios as a reference.

**TABLE 14.8** 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 key 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. |

## Comparing Collection Types

We conclude this section with a review of all the collection classes. Make sure that you can fill in Table 14.9 to compare the four collections types from memory.

**TABLE 14.9** 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 |
| `Map` | Yes (for values) | No | Yes | No |
| `Queue` | Yes | Yes (retrieved in defined order) | No | Yes |
| `Set` | No | No | No | No |

Additionally, make sure you can fill in Table 14.10 to describe the types on the exam.

**TABLE 14.10** Collection attributes

| Type | Java Collections Framework interface | Sorted? | Calls `hashCode()`? | Calls `compareTo()`? |
|---|---|---|---|---|
| `ArrayList` | `List` | No | No | No |
| `HashMap` | `Map` | No | Yes | No |
| `HashSet` | `Set` | No | Yes | No |
| `LinkedList` | `List`, `Queue` | No | No | No |
| `TreeMap` | `Map` | Yes | No | Yes |
| `TreeSet` | `Set` | Yes | No | Yes |

Next, the exam expects you to know which data structures allow `null` values. The data structures that involve sorting do not allow `null` values.

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 that you might come across in older code. All three were early Java data structures you could use with threads. In Chapter 18, you'll learn about modern alternatives if you need a concurrent collection.

- `Vector` : Implements `List` . If you don't need concurrency, use `ArrayList` instead.
- `Hashtable` : Implements `Map` . If you don't need concurrency, use `HashMap` instead.
- `Stack` : Implements `Queue` . If you don't need concurrency, use a `LinkedList` instead.

## 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. As far as the exam is concerned, that means numbers sort before letters, and uppercase letters sort before lowercase letters.


Remember that numbers sort before letters, and uppercase letters sort before lowercase letters.

We will be using `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 these data structures that require comparison. 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 will discuss `Comparable` first. Then, as we go through `Comparator`, we will 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 update the class declaration to inherit `Comparable<Duck>`, and then we implement the `compareTo()` method.

```
import java.util.*;
public class Duck implements Comparable<Duck> {
   private String name;
   public Duck(String name) {
      this.name = name;
   }
   public String toString() { // use readable output
      return name;
   }
   public int compareTo(Duck d) {
      return name.compareTo(d.name); // sorts ascendingly by name
   }
   public static void main(String[] args) {
      var ducks = new ArrayList<Duck>();
      ducks.add(new Duck("Quack"));
      ducks.add(new Duck("Puddles"));
```

```
        Collections.sort(ducks);   // sort by name
        System.out.println(ducks); // [Puddles, Quack]
    }}
```

Without implementing that interface, all we have is a method named `compareTo()`, but it wouldn't be a `Comparable` object. 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 with each other.

---


The `Duck` class overrides the `toString()` method from `Object`, which we described in Chapter 12. This override provides useful output when printing out ducks. Without this override, the output would be something like `[Duck@70dea4e, Duck@5c647e05]` —hardly useful in seeing which duck's name comes first.

---

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.

We still need to know what the `compareTo()` method returns so that we can write our own. There are three rules to know.

- 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.

```
1:  public class Animal implements Comparable<Animal> {
2:     private int id;
3:     public int compareTo(Animal a) {
4:        return id - a.id; // sorts ascending by id
5:     }
6:     public static void main(String[] args) {
7:        var a1 = new Animal();
8:        var a2 = new Animal();
9:        a1.id = 5;
10:       a2.id = 7;
11:       System.out.println(a1.compareTo(a2)); // -2
12:       System.out.println(a1.compareTo(a1)); // 0
13:       System.out.println(a2.compareTo(a1)); // 2
14:    } }
```

Lines 7 and 8 create two `Animal` objects. Lines 9 and 10 set their `id` values. This is not a good way to set instance variables. It would be better to use a constructor or setter method. Since the exam shows nontraditional code to make sure that you understand the rules, we throw in some nontraditional code as well.

Lines 3-5 show one way to compare two `int` values. We could have used `Integer.compare(id, a.id)` instead. Be sure to be able to recognize both approaches.

---


Remember that `id - a.id` sorts in ascending order, and `a.id - id` sorts in descending order.

---

Lines 11 through 13 confirm that we've implemented `compareTo()` correctly. Line 11 compares a smaller `id` to a larger one, and therefore it prints a negative number. Line 12 compares animals with the same `id`, and therefore it prints 0. Line 13 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`.

```
public class LegacyDuck implements Comparable {
   private String name;
   public int compareTo(Object obj) {
      LegacyDuck d = (LegacyDuck) obj; // cast because no generics
      return name.compareTo(d.name);
   }
}
```

Since we don't specify a generic type for `Comparable`, Java assumes that we want an `Object`, which means that we have to cast to `LegacyDuck` before accessing instance variables on it.

### 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 is not validated ahead of time.

```
public class MissingDuck implements Comparable<MissingDuck> {
   private String name;
   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`, then 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)) return false;
      var other = (Product) obj;
      return this.id == other.id;
   }
   public int compareTo(Product obj) {
      return this.name.compareTo(obj.name);
   } }
```

You might be sorting `Product` objects by name, but names are not unique. Therefore, the return value of `compareTo()` might not be `0` when comparing two equal `Product` objects, so this `compareTo()` method is not consistent with equals. One way to fix that is to use a `Comparator` to define the sort elsewhere.

Now that you know how to implement `Comparable` objects, you get to look at `Comparator`s and focus on the differences.

### 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. We now have the following:

```
1:  import java.util.ArrayList;
2:  import java.util.Collections;
3:  import java.util.Comparator;
4:
5:  public class Duck implements Comparable<Duck> {
6:     private String name;
7:     private int weight;
8:
9:     // Assume getters/setters/constructors provided
10:
11:    public String toString() { return name; }
12:
13:    public int compareTo(Duck d) {
14:       return name.compareTo(d.name);
15:    }
16:
17:    public static void main(String[] args) {
18:       Comparator<Duck> byWeight = new Comparator<Duck>() {
19:          public int compare(Duck d1, Duck d2) {
20:             return d1.getWeight()-d2.getWeight();
21:          }
22:       };
23:       var ducks = new ArrayList<Duck>();
24:       ducks.add(new Duck("Quack", 7));
25:       ducks.add(new Duck("Puddles", 10));
26:       Collections.sort(ducks);
27:       System.out.println(ducks); // [Puddles, Quack]
28:       Collections.sort(ducks, byWeight);
29:       System.out.println(ducks); // [Quack, Puddles]
30:    }
31: }
```

First, notice that this program imports `java.util.Comparator` on line 3. We don't always show imports since you can assume they are present if

not shown. Here, we do show the import to call attention to the fact that `Comparable` and `Comparator` are in different packages, namely, `java.lang` versus `java.util`, respectively. That means `Comparable` can be used without an `import` statement, while `Comparator` cannot.

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 18-22 of the `main()` method show how to define a `Comparator` using an inner class. On lines 26-29, we sort without the comparator and then with the 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 18-22 using a lambda expression, as shown here:

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

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

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

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 a good number of differences between `Comparable` and `Comparator`. We've listed them for you in Table 14.11.

**TABLE 14.11** 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 one 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` class, as shown here:

```
public class Squirrel {
   private int weight;
   private String species;
   // Assume getters/setters/constructors provided
}
```

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.getSpecies().compareTo(s2.getSpecies());
      if (result != 0) return result;
      return s1.getWeight()-s2.getWeight();
   }}
```

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 that 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::getSpecies)
   .thenComparingInt(Squirrel::getWeight);
```

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. As discussed in Chapter 12, `default` methods were introduced in Java 8 as a way of expanding APIs.

Suppose we want to sort in descending order by species.

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

Table 14.12 shows the helper methods you should know for building `Comparator`. We've omitted the parameter types to keep you focused on the methods. They use many of the functional interfaces you'll be learning about in the next chapter.

**TABLE 14.12** Helper static methods for building a Comparator

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

Table 14.13 shows the methods that you can chain to a `Comparator` to further specify its behavior.

**TABLE 14.13** Helper default methods for building a Comparator

| Method | Description |
|---|---|
| `reversed()` | Reverse the order of the chained `Comparator`. |
| `thenComparing(function)` | If the previous `Comparator` returns `0`, use this comparator that returns an `Object` or can be autoboxed into one. |
| `thenComparingDouble(function)` | If the previous `Comparator` returns `0`, use this comparator that returns a `double`. Otherwise, return the value from the previous `Comparator`. |
| `thenComparingInt(function)` | If the previous `Comparator` returns `0`, use this comparator that returns an `int`. Otherwise, return the value from the previous `Comparator`. |
| `thenComparingLong(function)` | If the previous `Comparator` returns `0`, use this comparator that returns a `long`. Otherwise, return the value from the 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 it, 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 class Rabbit{ int id; }
4:    public static void main(String[] args) {
5:       List<Rabbit> rabbits = new ArrayList<>();
6:       rabbits.add(new Rabbit());
7:       Collections.sort(rabbits); // DOES NOT COMPILE
8:    } }
```

Java knows that the `Rabbit` class 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.

```
2: public class SortRabbits {
3:    static class Rabbit{ int id; }
4:    public static void main(String[] args) {
5:       List<Rabbit> rabbits = new ArrayList<>();
6:       rabbits.add(new Rabbit());
7:       Comparator<Rabbit> c = (r1, r2) -> r1.id - r2.id;
8:       Collections.sort(rabbits, c);
9:    } }
```

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 correct answer is `-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 in ascending order, we don't meet the precondition for doing a search.

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

actually 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:     static class 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());  // 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 Duck 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());
```

Now Java knows that you want to sort by `id` and all is well. `Comparator`s are helpful objects. They let 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.

## Working with Generics

We conclude this chapter with one of the most useful, and at times most confusing, feature in the Java language: generics. Why do we need generics? Well, remember when we said that we had to hope the caller didn't put something in the list that we didn't expect? The following does just that:

```
14: static void printNames(List list) {
15:    for (int i = 0; i < list.size(); i++) {
16:       String name = (String) list.get(i); // ClassCastException
17:       System.out.println(name);
18:    }
19: }
20: public static void main(String[] args) {
21:    List names = new ArrayList();
22:    names.add(new StringBuilder("Webby"));
23:    printNames(names);
24: }
```

This code throws a `ClassCastException` . Line 22 adds a `StringBuilder` to `list` . This is legal because a nongeneric list can contain anything. However, line 16 is written to expect a specific class to be in there. It casts to a `String` , reflecting this assumption. Since the assumption is incorrect, the code throws a `ClassCastException` that `java.lang .StringBuilder` cannot be cast to `java.lang.String` .

Generics fix this by allowing you to write and use parameterized types. You specify that you want an `ArrayList` of `String` objects. Now the compiler has enough information to prevent you from causing this problem in the first place.

```
List<String> names = new ArrayList<String>();
names.add(new StringBuilder("Webby")); // 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.

### 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 emptyCrate() {
      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.emptyCrate();
```

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 yet. 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 St. Louis
Robot atDestination = robotCrate.emptyCrate();
```

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 of needing 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` either. 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. They do show up frequently in the code you call, such as the Java Collections Framework.

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 SizeLimiteCrate<>(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.

#### WHAT IS 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 actually just `Object` types. The `Crate` class looks like the following at runtime:

```
public class Crate {
   private Object contents;
   public Object emptyCrate() {
      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.emptyCrate();
```

The compiler turns it into the following:

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

### 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 so that it isn't confusing as to what `T` refers to. The exam won't mind trying to confuse you by using the same type parameter name.

### Raw Types

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 types whose information is fully available at runtime as *reifiable*. Reifiable types can do anything that Java allows. Nonreifiable 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):

- **Calling a constructor:** Writing `new T()` is not allowed because at runtime it would be `new Object()`.
- **Creating 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.
- **Calling `instanceof`:** This is not allowed because at runtime `List<Integer>` and `List<String>` look the same to Java thanks to type erasure.
- **Using 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`.
- **Creating a `static` variable as a generic type parameter:** This is not allowed because the type is linked to the instance of the class.

### 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 it 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);
```

As to whether this makes things clearer, it is up to you. 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 Crate<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 createName() {
11:    Crate<Robot> crate = new Crate<>();
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.

## Bounding Generic Types

By now, you might have noticed that generics don't seem particularly useful since they are treated as an `Object` and therefore don't have many methods available. Bounded wildcards solve this by restricting what types can be used in a wildcard. 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 14.14. This section looks at each of these three wildcard types.

**TABLE 14.14** Types of bounds

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

### Unbounded Wildcards

An unbounded wildcard represents any data type. You use `?` when you want to specify that any type is okay 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(new Integer(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 were to have 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 that 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 really 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.

**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 one of the benefits 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.

**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 that you understand

why the first three examples in Table 14.15 do *not* solve this problem.

**TABLE 14.15** Why we need a lower bound

| `public static void addSound(_____list) {list.add("quack");}` | **Method compiles** | **Can pass a** `List<String>` | **Can pass a** `List<Object>` |
|---|---|---|---|
| `List<?>` | No (unbounded generics are immutable) | Yes | Yes |
| `List<? extends Object>` | No (upper-bounded generics are immutable) | 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!

---

**UNDERSTAND 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`. What happens is that 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 okay.

Line 7 tries to store a list in a variable declaration with an upper-bounded wildcard. This is okay. You can have `ArrayList<A>`, `ArrayList<B>`, or `ArrayList<C>` stored in that reference. Line 8 is also okay. 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<B>`, 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. 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()`, which is a normal use of generics. You can pass the types `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. That's the hardest topic in this chapter (and why we covered it last!). Remember that it's okay if you need to go over this material a few times to get your head around it.

## Summary

A method reference is a compact syntax for writing lambdas that refer to methods. There are four types: `static` methods, instance methods on a

particular object, instance methods on a parameter, and constructor references.

Each primitive class has a corresponding wrapper class. For example, `long`'s wrapper class is `Long`. Java can automatically convert between primitive and wrapper classes when needed. This is called autoboxing and unboxing. Java will use autoboxing only if it doesn't find a matching method signature with the primitive. For example, `remove(int n)` will be called rather than `remove(Object o)` when called with an `int`.

The diamond operator (`<>`) is used to tell Java that the generic type matches the declaration without specifying it again. The diamond operator can be used for local variables or instance variables as well as one-line declarations.

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`. 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
  - **`TreeSet`:** Sorted. Does not allow `null` values
- **`Queue`:** Orders elements for processing
  - **`LinkedList`:** Can easily add/remove from beginning or end
- **`Map`:** Maps unique keys to values
  - **`HashMap`:** Uses `hashCode()` to find keys
  - **`TreeMap`:** Sorted map. Does not allow `null` keys

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. `Comparator` is often implemented using a lambda.

The `Arrays` and `Collections` classes have methods for `sort()` and `binarySearch()`. Both take an optional `Comparator` parameter. It is necessary to use the same sort order for both sorting and searching, so the result is not undefined.

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

**Translate method references to the "long form" lambda.** Be able to convert method references into regular lambda expressions and vice versa. For example, `System.out::print` and `x -> System.out.print(x)` are equivalent. Remember that the order of method parameters is inferred for both based on usage.

**Use autoboxing and unboxing.** Autoboxing converts a primitive into an `Object`. For example, `int` is autoboxed into `Integer`. Unboxing converts an `Object` into a primitive. For example, `Character` is autoboxed into `char`.

**Pick the correct type of collection from a description.** A `List` allows duplicates and orders the elements. A `Set` does not allow duplicates. A `Queue` orders its elements to facilitate retrievals. A `Map` maps keys to val-

ues. Be familiar with the differences of implementations of these interfaces.

**Work 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.

**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 `Comparator`s can have different sort orders. A `Comparator` is often implemented using a lambda such as `(a, b) -> a.num - b.num`.

**Write code using the diamond operator.** The diamond operator (`<>`) is used to write more concise code. The type of the generic parameter is inferred from the surrounding code. For example, in `List<String> c = new ArrayList<>()`, the type of the diamond operator is inferred to be `String`.

**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 and applies to both classes and interfaces. `<? super X>` is a lower-bounded wildcard.

## Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. Suppose that you have a collection of products for sale in a database and you need to display those products. The products are not unique. Which of the following collections classes in the `java.util` package best suits your needs for this scenario?
   A. `Arrays`
   B. `ArrayList`

C. `HashMap`
D. `HashSet`
E. `LinkedList`

2. Suppose that you need to work with a collection of elements that need to be sorted in their natural order, and each element has a unique text `id` that you want to use to store and retrieve the record. Which of the following collections classes in the `java.util` package best suits your needs for this scenario?
   A. `ArrayList`
   B. `HashMap`
   C. `HashSet`
   D. `TreeMap`
   E. `TreeSet`
   F. None of the above
3. Which of the following are true? (Choose all that apply.)

```
12: List<?> q = List.of("mouse", "parrot");
13: var v = List.of("mouse", "parrot");
14:
15: q.removeIf(String::isEmpty);
16: q.removeIf(s -> s.length() == 4);
17: v.removeIf(String::isEmpty);
18: v.removeIf(s -> s.length() == 4);
```

   A. This code compiles and runs without error.
   B. Exactly one of these lines contains a compiler error.
   C. Exactly two of these lines contain a compiler error.
   D. Exactly three of these lines contain a compiler error.
   E. Exactly four of these lines contain a compiler error.
   F. If any lines with compiler errors are removed, this code runs without throwing an exception.
   G. If all lines with compiler errors are removed, this code throws an exception.
4. What is the result of the following statements?

```
3:  var greetings = new LinkedList<String>();
4:  greetings.offer("hello");
5:  greetings.offer("hi");
```

```
6:  greetings.offer("ola");
7:  greetings.pop();
8:  greetings.peek();
9:  while (greetings.peek() != null)
10:    System.out.print(greetings.pop());
```

A. `hello`
B. `hellohi`
C. `hellohiola`
D. `hiola`
E. `ola`
F. The code does not compile.
G. An exception is thrown.

5. Which of these statements compile? (Choose all that apply.)

A. `HashSet<Number> hs = new HashSet<Integer>();`
B. `HashSet<? super ClassCastException> set = new HashSet<Exception>();`
C. `List<> list = new ArrayList<String>();`
D. `List<Object> values = new HashSet<Object>();`
E. `List<Object> objects = new ArrayList<? extends Object>();`
F. `Map<String, ? extends Number> hm = new HashMap<String, Integer>();`

6. What is the result of the following code?

```
1:  public class Hello<T> {
2:     T t;
3:     public Hello(T t) { this.t = t; }
4:     public String toString() { return t.toString(); }
5:     private <T> void println(T message) {
6:        System.out.print(t + "-" + message);
7:     }
8:     public static void main(String[] args) {
9:        new Hello<String>("hi").println(1);
10:       new Hello("hola").println(true);
11:    } }
```

A. `hi` followed by a runtime exception
B. `hi-1hola-true`
C. The first compiler error is on line 1.

D. The first compiler error is on line 4.

E. The first compiler error is on line 5.

F. The first compiler error is on line 9.

G. The first compiler error is on line 10.

7. Which of the following statements are true? (Choose all that apply.)

```
3:  var numbers = new HashSet<Number>();
4:  numbers.add(Integer.valueOf(86));
5:  numbers.add(75);
6:  numbers.add(Integer.valueOf(86));
7:  numbers.add(null);
8:  numbers.add(309L);
9:  Iterator iter = numbers.iterator();
10: while (iter.hasNext())
11:    System.out.print(iter.next());
```

A. The code compiles successfully.

B. The output is `8675null309`.

C. The output is `867586null309`.

D. The output is indeterminate.

E. There is a compiler error on line 3.

F. There is a compiler error on line 9.

G. An exception is thrown.

8. Which of the following can fill in the blank to print `[7, 5, 3]`? (Choose all that apply.)

```
3:  public class Platypus {
4:     String name;
5:     int beakLength;
6:
7:     // Assume getters/setters/constructors provided
8:
9:     public String toString() {return "" + beakLength;}
10:
11:    public static void main(String[] args) {
12:       Platypus p1 = new Platypus("Paula", 3);
13:       Platypus p2 = new Platypus("Peter", 5);
14:       Platypus p3 = new Platypus("Peter", 7);
15:
16:       List<Platypus> list = Arrays.asList(p1, p2, p3);
```

```
17:
18:       Collections.sort(list, Comparator.comparing      );
19:
20:       System.out.println(list);
21:    }
22: }
```

A.
```
(Platypus::getBeakLength)
```

B.
```
(Platypus::getBeakLength).reversed()
```

C.
```
(Platypus::getName)
   .thenComparing(Platypus::getBeakLength)
```

D.
```
(Platypus::getName)
   .thenComparing(
      Comparator.comparing(Platypus::getBeakLength)
   .reversed())
```

E.
```
(Platypus::getName)
   .thenComparingNumber(Platypus::getBeakLength)
   .reversed()
```

F.
```
(Platypus::getName)
   .thenComparingInt(Platypus::getBeakLength)
   .reversed()
```

G. None of the above

9. Which of the answer choices are valid given the following code? (Choose all that apply.)

```
Map<String, Double> map = new HashMap<>();
```

A. `map.add("pi", 3.14159);`

B. `map.add("e", 2L);`

C. `map.add("log(1)", new Double(0.0));`

D. `map.add('x', new Double(123.4));`

E. None of the above

10. What is the result of the following program?

```
3:  public class MyComparator implements Comparator<String> {
4:     public int compare(String a, String b) {
5:        return b.toLowerCase().compareTo(a.toLowerCase());
6:     }
7:     public static void main(String[] args) {
8:        String[] values = { "123", "Abb", "aab" };
9:        Arrays.sort(values, new MyComparator());
10:       for (var s: values)
11:          System.out.print(s + " ");
12:    }
13: }
```

A. `Abb aab 123`

B. `aab Abb 123`

C. `123 Abb aab`

D. `123 aab Abb`

E. The code does not compile.

F. A runtime exception is thrown.

11. What is the result of the following code?

```
3: var map = new HashMap<Integer, Integer>(10);
4: for (int i = 1; i <= 10; i++) {
5:    map.put(i, i * i);
6: }
7: System.out.println(map.get(4));
```

A. `16`

B. `25`

C. Compiler error on line 3.

D. Compiler error on line 5.

E. Compiler error on line 7.

F. A runtime exception is thrown.

12. Which of these statements can fill in the blank so that the `Helper` class compiles successfully? (Choose all that apply.)

```
2:  public class Helper {
3:     public static <U extends Exception>
4:        void printException(U u) {
5:
6:        System.out.println(u.getMessage());
7:     }
8:     public static void main(String[] args) {
9:        Helper._____________________________________;
10:    } }
```

A. `printException(new FileNotFoundException("A"))`
B. `printException(new Exception("B"))`
C. `<Throwable>printException(new Exception("C"))`
D. `<NullPointerException>printException(new NullPointerException ("D"))`
E. `printException(new Throwable("E"))`

13. Which of these statements can fill in the blank so that the `Wildcard` class compiles successfully? (Choose all that apply.)

```
3:  public class Wildcard {
4:     public void showSize(List<?> list) {
5:        System.out.println(list.size());
6:     }
7:     public static void main(String[] args) {
8:        Wildcard card = new Wildcard();
9:        _____________________________________;
10:       card.showSize(list);
11:    } }
```

A. `List<?> list = new HashSet <String>()`
B. `ArrayList<? super Date> list = new ArrayList<Date>()`
C. `List<?> list = new ArrayList<?>()`
D. `List<Exception> list = new LinkedList<java.io.IOException>()`
E. `ArrayList <? extends Number> list = new ArrayList <Integer>()`

F. None of the above

14. What is the result of the following program?

```
3:  public class Sorted
4:     implements Comparable<Sorted>, Comparator<Sorted> {
5:
6:     private int num;
7:     private String text;
8:
9:     // Assume getters/setters/constructors provided
10:
11:    public String toString() { return "" + num; }
12:    public int compareTo(Sorted s) {
13:       return text.compareTo(s.text);
14:    }
15:    public int compare(Sorted s1, Sorted s2) {
16:       return s1.num - s2.num;
17:    }
18:    public static void main(String[] args) {
19:       var s1 = new Sorted(88, "a");
20:       var s2 = new Sorted(55, "b");
21:       var t1 = new TreeSet<Sorted>();
22:       t1.add(s1); t1.add(s2);
23:       var t2 = new TreeSet<Sorted>(s1);
24:       t2.add(s1); t2.add(s2);
25:       System.out.println(t1 + " " + t2);
26:    } }
```

A. `[55, 88] [55, 88]`

B. `[55, 88] [88, 55]`

C. `[88, 55] [55, 88]`

D. `[88, 55] [88, 55]`

E. The code does not compile.

F. A runtime exception is thrown.

15. What is the result of the following code? (Choose all that apply.)

```
Comparator<Integer> c1 = (o1, o2) -> o2 - o1;
Comparator<Integer> c2 = Comparator.naturalOrder();
Comparator<Integer> c3 = Comparator.reverseOrder();

var list = Arrays.asList(5, 4, 7, 2);
```

```
Collections.sort(list,_________);
System.out.println(Collections.binarySearch(list, 2));
```

A. One or more of the comparators can fill in the blank so that the code prints `0`.
B. One or more of the comparators can fill in the blank so that the code prints `1`.
C. One or more of the comparators can fill in the blank so that the code prints `2`.
D. The result is undefined regardless of which comparator is used.
E. A runtime exception is thrown regardless of which comparator is used.
F. The code does not compile.

16. Which of the following statements are true? (Choose all that apply.)
A. `Comparable` is in the `java.util` package.
B. `Comparator` is in the `java.util` package.
C. `compare()` is in the `Comparable` interface.
D. `compare()` is in the `Comparator` interface.
E. `compare()` takes one method parameter.
F. `compare()` takes two method parameters.

17. Which options can fill in the blanks to make this code compile? (Choose all that apply.)

```
1: public class Generic _________{
2:    public static void main(String[] args) {
3:       Generic<String> g = new Generic _______();
4:       Generic<Object> g2 = new Generic();
5:    }
6: }
```

A. On line 1, fill in with `<>`.
B. On line 1, fill in with `<T>`.
C. On line 1, fill in with `<?>`.
D. On line 3, fill in with `<>`.
E. On line 3, fill in with `<T>`.
F. On line 3, fill in with `<?>`.

18. Which of the following lines can be inserted to make the code compile? (Choose all that apply.)

```
class W {}
class X extends W {}
class Y extends X {}
class Z<Y> {
// INSERT CODE HERE
}
```

A. `W w1 = new W();`
B. `W w2 = new X();`
C. `W w3 = new Y();`
D. `Y y1 = new W();`
E. `Y y2 = new X();`
F. `Y y1 = new Y();`

19. Which options are true of the following code? (Choose all that apply.)

```
3: _____________<Integer> q = new LinkedList<>();
4: q.add(10);
5: q.add(12);
6: q.remove(1);
7: System.out.print(q);
```

A. If we fill in the blank with `List`, the output is `[10]`.
B. If we fill in the blank with `List`, the output is `[10, 12]`.
C. If we fill in the blank with `Queue`, the output is `[10]`.
D. If we fill in the blank with `Queue`, the output is `[10, 12]`.
E. The code does not compile in either scenario.
F. A runtime exception is thrown.

20. What is the result of the following code?

```
4: Map m = new HashMap();
5: m.put(123, "456");
6: m.put("abc", "def");
7: System.out.println(m.contains("123"));
```

A. `false`
B. `true`
C. Compiler error on line 4
D. Compiler error on line 5

E. Compiler error on line 7

F. A runtime exception is thrown.

21. What is the result of the following code? (Choose all that apply.)

```
48: var map = Map.of(1,2, 3, 6);
49: var list = List.copyOf(map.entrySet());
50:
51: List<Integer> one = List.of(8, 16, 2);
52: var copy = List.copyOf(one);
53: var copyOfCopy = List.copyOf(copy);
54: var thirdCopy = new ArrayList<>(copyOfCopy);
55:
56: list.replaceAll(x -> x * 2);
57: one.replaceAll(x -> x * 2);
58: thirdCopy.replaceAll(x -> x * 2);
59:
60: System.out.println(thirdCopy);
```

A. One line fails to compile.

B. Two lines fail to compile.

C. Three lines fail to compile.

D. The code compiles but throws an exception at runtime.

E. If any lines with compiler errors are removed, the code throws an exception at runtime.

F. If any lines with compiler errors are removed, the code prints `[16, 32, 4]`.

G. The code compiles and prints `[16, 32, 4]` without any changes.

22. What code change is needed to make the method compile assuming there is no class named `T`?

```
public static T identity(T t) {
  return t;
}
```

A. Add `<T>` after the `public` keyword.

B. Add `<T>` after the `static` keyword.

C. Add `<T>` after `T`.

D. Add `<?>` after the `public` keyword.

E. Add `<?>` after the `static` keyword.

F. No change required. The code already compiles.

23. Which of the answer choices make sense to implement with a lambda? (Choose all that apply.)

A. `Comparable` interface

B. `Comparator` interface

C. `remove` method on a `Collection`

D. `removeAll` method on a `Collection`

E. `removeIf` method on a `Collection`

24. Which of the following compiles and prints out the entire set? (Choose all that apply.)

```
Set<?> set = Set.of("lion", "tiger", "bear");
var s = Set.copyOf(set);
s.forEach(_____________________________);
```

A. `() -> System.out.println(s)`

B. `s -> System.out.println(s)`

C. `(s) -> System.out.println(s)`

D. `System.out.println(s)`

E. `System::out::println`

F. `System.out::println`

G. None of the above

25. What is the result of the following?

```
var map = new HashMap<Integer, Integer>();
map.put(1, 10);
map.put(2, 20);
map.put(3, null);
map.merge(1, 3, (a,b) -> a + b);
map.merge(3, 3, (a,b) -> a + b);
System.out.println(map);
```

A. `{1=10, 2=20}`

B. `{1=10, 2=20, 3=null}`

C. `{1=10, 2=20, 3=3}`

D. `{1=13, 2=20}`

E. `{1=13, 2=20, 3=null}`

F. `{1=13, 2=20, 3=3}`

G. The code does not compile.

H. An exception is thrown.